

# Correctness of Concurrent Executions of Closed Nested Transactions in Transactional Memory Systems <sup>\*†</sup>

Sathya Peri <sup>‡</sup>  
sathya@iitp.ac.in

CSE Department, Indian Institute of Technology Patna, Patna, India

K.Vidyasankar  
vidya@mun.ca

Department of Computer Science, Memorial University, St John's, Canada

## Abstract

A generally agreed upon requirement for correctness of concurrent executions in Transactional Memory is that all transactions including the aborted ones read consistent values. *Opacity* is a commonly accepted correctness criterion that satisfies the above requirement. Our first contribution in this paper is extending the opacity definition for closed nested transactions. Secondly, we define a restricted class, again for closed nested transactions, that preserves conflicts. This is akin to conflict-serializable class for traditional database transactions. Our conflict definition is appropriate for optimistic executions which are most common in Software Transactional Memory (STM) systems. We show that membership in the new class can be checked in polynomial time. With opacity, an aborted transaction (considering only the read steps that were executed before aborting) may affect the consistency for the transactions that are executed subsequently. This property is not desirable in general and may be harmful for closed nested transactions in the sense that the abort of a sub-transaction may make committing its top-level transaction impossible. As our third contribution, we propose a correctness criterion that defines a class of schedules where aborted transactions do not affect consistency for other transactions. We define a conflict-preserving subclass of this class as well. Then we give the outline of a scheduler that implements this subclass. Both the class definitions and the conflict definition are new for nested transactions.

*Keywords:* STMs, Closed Nesting, Correctness Criterion, Conflict Notion

## 1 Introduction

In the recent years software transactional memory has garnered significant interest as an elegant alternative for developing concurrent code. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly [7, 17]. Traditionally locks have been used for developing parallel programs. But programming with locks has many disadvantages such as deadlocks, priority inversion etc.

---

<sup>\*</sup>This research is supported in part by the Natural Sciences and Engineering Research Council of Canada individual research grant 3182

<sup>†</sup>A preliminary version of this paper appeared in Twelfth International Conference on Distributed Computing and Networking (ICDCN 2011)

<sup>‡</sup>Corresponding Author

These disadvantages make it difficult to build scalable software systems. Importantly, lock based software components are difficult to compose i.e. building larger software systems using simpler software components [6]. Software transactions address many of the shortcomings of lock based systems. Specifically, software transactions provide a very promising approach for composing software components [6].

A (memory) transaction is an unit of code in execution in memory. A software transactional memory system (STM) ensures that a transaction appears either to execute atomically (even in presence of other concurrent transactions) or to never have executed at all. If a transaction executes to completion then it is *committed* and its effects are visible to other transactions. Otherwise it is *aborted* and none of its effects are visible to other transactions. Thus the values written by a live (incomplete) transaction to the memory are not visible to other transactions. To explain this concept, consider two transactions  $t_1, t_2$  accessing a data-item, say  $x$ , which is initialized to 0. Let the sequence of operations be:  $w_1(x, 5)r_2(x)c_1c_2$  where  $c_1, c_2$  refer to the commit operations of transactions  $t_1, t_2$  respectively. Here the value that  $t_2$  reads for  $x$  is 0 since at the time when  $t_2$  reads  $x$ ,  $t_1$  has not yet committed. Thus its write is not yet visible to  $t_2$ .

To achieve this effect, a commonly used approach by software transactions is optimistic synchronization (term used in [6]). In this approach, transactions have a local log where they record the values read and written in the course of its execution. When the transaction completes, it validates the contents of its log. If the log contributes to a consistent view of the memory, then the transaction updates the memory with the contents of the log. If not it aborts.

A STM system implements the log described above by having one global buffer for each data-item and one local buffer for each transaction accessing that data item. In the example described above, a global buffer is created for  $x$ . Any write to  $x$  by  $t_1$  is performed in the local buffer. When  $t_1$  commits, the value in the local buffer is transferred to the global buffer. Then  $t_1$ 's write values can be viewed by others. Hence until  $t_1$  commits, its write operations are not visible to  $t_2$ .

Composing simple transactions to build a larger transaction is an extremely useful property which forms the basis of modular programming. In STMs this can be achieved through nesting of transactions. A transaction is called nested if it invokes another transaction as a part of its execution. Nested transactions can broadly be classified as: *closed* and *open*. Consider a transaction  $t_P$  which has a sub-transaction  $t_S$ . In closed nesting when the sub-transaction  $t_S$  commits its effects are visible to  $t_P$  (and the siblings of  $t_S$ ) but not to other transactions. On the other hand in open nesting the effects of the transaction  $t_S$  are visible to other transactions immediately after it commits without waiting for its parent transaction  $t_P$  to commit. However when  $t_P$  aborts then  $t_S$  is also aborted. In this paper we focus only on closed nested transactions.

To achieve atomicity, the above discussed notion of multiple buffers extends naturally to closed nested transactions. When a sub-transaction is invoked, new buffers are created for all the data-items it writes to. The contents of the buffers are merged with its parent's buffers when the sub-transaction commits. Thus if the sub-transaction writes any value to any data-item, that value will not be visible to its parent until the sub-transaction commits.

When (nested or non-nested) transactions accessing common data-items execute concurrently it is imperative that they execute correctly. A commonly accepted correctness requirement for concurrent executions in STM systems is that all transactions including aborted ones read consistent values. The values resulting from any serial execution of transactions are assumed to be consistent. Guerraoui and Kapalka [5] captured this requirement as *opacity* by requiring, for any concurrent execution, a single equivalent serial execution of all committed and aborted transactions, considering only the read steps of aborted ones. An implementation of opacity for non-nested transactions has been given in [9].

The correctness criterion used in traditional databases is serializability [14, 18]. According to serializability an interleaving execution of committed transactions is correct if it is equivalent to some serial

execution of the same set of transactions. But serializability concerns itself only with the events of committed transactions. Any execution that satisfies serializability ensures that all committed transactions read consistent values. It does not require that the aborted transactions read consistent values. As pointed out in [5] this is acceptable in the context of databases which are executed in highly controlled environments. But in the context of STMs, it is imperative that even the operations of aborted transactions see consistent values. Otherwise it could have several undesirable effects such as ‘divide by zero’ error, crash failure or even infinite loops [5, 9].

On the other hand, the recent understanding (Doherty et al [3], Imbs et al [8]) is that opacity is too strong a correctness criterion for STMs. Weaker notions have been proposed: (i) The requirement of a single equivalent serial schedule is replaced by allowing possibly different equivalent serial schedules, one for all committed transactions and one for each aborted transaction, and these schedules need not be compatible; and (ii) the effects, namely, the read steps, of aborted transactions should not affect the consistency of the transactions executed subsequently. The first point refines the consistency notion for aborted transactions. The second point is a desirable property for transactions in general and a critical point for nested transactions, where, otherwise, the effects of an aborted sub-transaction may prohibit committing the entire top-level transaction. The above proposals in the literature have been made for non-nested transactions.

In this paper, we extend the opacity definition for closed nested transactions. We define two notions and corresponding classes of schedules: *Closed Nested Opacity (CNO)* and *Abort-Shielded Consistency (ASC)*. In the first notion, read steps of aborted (sub-)transactions are included as in Guerraoui and Kapalka [5, 9]. In the second, they are discarded. These extensions turn out to be nontrivial due to the fact that an aborted sub-transaction may have some committed descendants and similarly some committed ancestors.

Checking opacity, like general serializability (for instance, view-serializability), cannot be done efficiently. Very much like restricted classes of serializability allowing polynomial membership test, and facilitating online scheduling, restricted classes of opacity can also be defined. We define such classes along the lines of conflict-serializability for database transactions: *Conflict-Preserving Closed Nested Opacity (CP-CNO)* and *Conflict-Preserving Abort-Shielded Consistency (CP-ASC)*. Our conflict notion is tailored for closed-nested transactions. We give an algorithm for checking the membership in CP-CNO (which can be easily modified for CP-ASC) and a scheduler for CP-ASC (which can be easily modified for CP-CNO). Both use serialization graphs similar to those in [16].

We note that all online schedulers (implementing 2PL, timestamp, optimistic approaches, etc.) for database transactions allow only subclasses of conflict-serializable schedules. We believe similarly that all STM schedulers can only allow subclasses of conflict-preserving schedules satisfying opacity or any of its variants. Such schedulers are likely to use mechanisms simpler than serialization graphs as in the database area. An example is the scheduler described by Imbs and Raynal [9].

In the context of nested transactions there have been many implementations of nested transactions in the past few years [2, 13, 12, 1, 11, 10]. In [5], the authors discuss extending opacity to nested transactions. But none of them provide a precise correctness criteria for nested software transactional memory system that can be efficiently verified.

To summarize, in this paper we present two classes of correctness criteria for closed nested transactions and describe subsets of these classes that can be efficiently verified.

**Roadmap:** In Section 2, we describe our model and background. In Section 3, we define CNO, CP-CNO and give an algorithm for polynomial membership test. In Section 4, we present ASC and CP-ASC. In Section 5 we discuss about some variations to the definitions presented and Section 6 concludes this paper. The following table shows the important definitions and theorems of this paper.

Definitions/Theorems	Explanation	Page Number
Definition 4	Formally defines CNO	16
Definition 5	Formally defines CP-CNO	18
Theorem 14	Shows that CP-CNO is a subset of CNO	19
Theorem 30	Proves the graph characterization of CP-CNO	29
Definition 6	Formally defines ASC	38
Lemma 34	Shows that CNO is a subset of ASC	38

## 2 Background and System Model

A transaction is a piece of code in execution. In the course of its execution a nested transaction may perform read/write operations on memory and invoke other transactions (also referred to as sub-transactions). We refer to these as *operations* of the transaction. A sub-transaction (of a transaction) could further invoke other transactions as a part of its execution. Thus a computation involving nested transactions constitutes a *computation tree*. The nodes of this tree are read and write operations, and transactions. The operations of a transaction can be viewed as its children. The operations are classified as: *simple-memory operations* and *transaction operations* or just *transactions*. Simple-memory operations are read or write operations on memory and have no children. Thus in the computation tree all the leaves are simple-memory operations.

In addition to memory operations, a transaction also contains a *commit* or *abort* operation. If a transaction  $t_X$  executes successfully to completion, it terminates with a commit operation  $c_X$ . Otherwise it aborts with the operation  $a_X$ . Abort and commit operations are called *terminal operations*. By default, all the simple-memory operations always commit.

Consider a closed-nested transaction  $t_P$ . All the writes by  $t_P$  are performed on its local buffer. When  $t_P$  commits, the contents of its local buffers are merged with the buffer of its parent. Thus any peer transaction of  $t_P$  can read the values written by  $t_P$  only after it commits. If  $t_P$  aborts then its local write values are not merged with its parent's buffers. Thus, none of the writes of an aborted transaction ever become visible to other transactions.

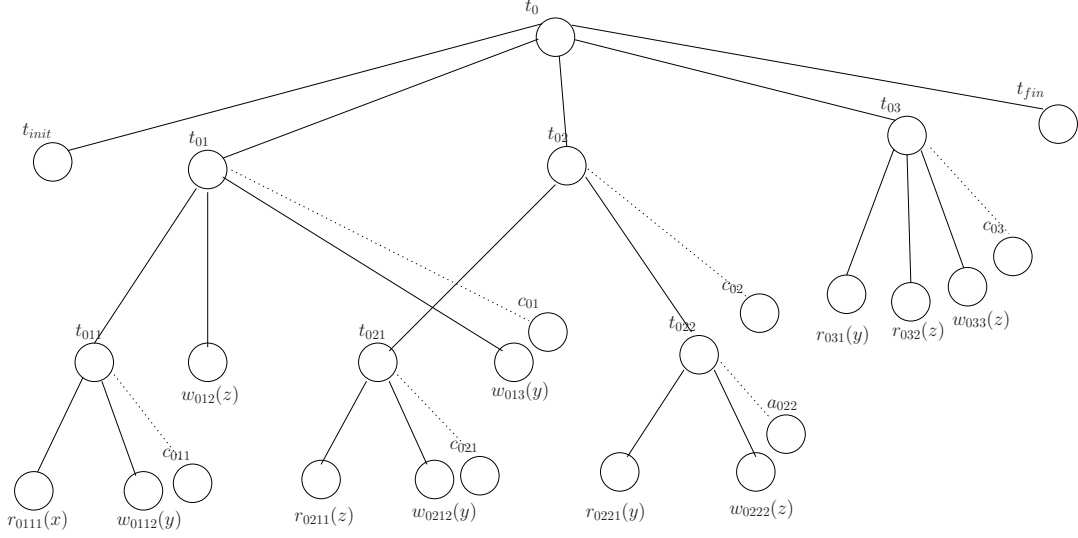
We assume that there exists a hypothetical root transaction of the computation tree, denoted as  $t_0$ , which invokes all the other transactions. On system initialization, we assume that there exists a child transaction of  $t_0$ ,  $t_{init}$ , which initializes all the buffers of  $t_0$  with non- $\perp$  values. Similarly we also assume that there exists a child transaction of  $t_0$ ,  $t_{fin}$ , which reads the contents of  $t_0$ 's buffers when the computation terminates.

### 2.1 Schedules

All transactions and simple-memory operations are nodes of the computation tree. We denote them as  $n_{id}$ . An *id* is concatenation of digits and uniquely identifies a transaction/operation. When we are specifically referring to a transaction we denote it as  $t_X$ . For a transaction with *id* as  $t_X$  having  $k$  children, we name the child operations as  $n_{X1}, n_{X2}, \dots, n_{Xk}$ . If a child (for example  $n_{X1}$ ) is a simple-memory operation reading or writing data-item  $y$  then we denote it as  $r_{X1}(y)$  or  $w_{X1}(y)$  and also as  $sm_{X1}(y)$ .

A sample computation tree is shown here. We show each transaction followed by all its operations. In Figure 1 we show the computation tree for this schedule. As indicated earlier we denote the root transaction as  $t_0$ :

**Example 1**  $t_0 : \{t_{init}, t_{01}, t_{02}, t_{03}, t_{fin}\},$   
 $t_{01} : \{t_{011}, sm_{012} = w_{012}(z), sm_{013} = w_{013}(y), c_{01}\},$   
 $t_{011} : \{sm_{0111} = r_{0111}(x), sm_{0112} = w_{0112}(y), c_{011}\},$



**Figure 1: Computation tree for Example 1**

$$\begin{aligned}
t_{02} &: \{t_{021}, t_{022}, c_{02}\}, \\
t_{021} &: \{sm_{0211} = r_{0211}(z), sm_{0212} = w_{0212}(y), c_{021}\}, \\
t_{022} &: \{sm_{0221} = r_{0221}(y), sm_{0222} = w_{0222}(z), a_{022}\}, \\
t_{03} &: \{sm_{031} = r_{031}(y), sm_{032} = r_{032}(z), sm_{033} = w_{033}(z), c_{03}\}
\end{aligned}$$

A *schedule* is a real time execution of the leaves of a computation tree. The events of a schedule are memory operations and terminal operations of transactions in the computation. The events of a schedule  $S$  are totally ordered. A schedule is represented by the tuple  $\langle evts, nodes, ord \rangle$ , where  $evts$  is the set of all events in the schedule,  $nodes$  is the set of all the nodes (transactions and simple-memory operations) present in the computation and  $ord$  is a function that totally orders all the events. In the context of a schedule we denote an event of a schedule as  $e_i$ . Thus all the leaf nodes in the tree are referred to as events in the context of schedules. A schedule for the computation tree in Example 1 can be represented as:

**Example 2**

$$S1 : r_{0111}(x)w_{0112}(y)c_{011}w_{012}(z)r_{0211}(z)w_{0212}(y)c_{021}w_{013}(y)c_{01}r_{0221}(y)w_{0222}(z)a_{022}c_{02}r_{031}(y)r_{032}(z)w_{033}(z)c_{03}$$

For a closed nested transaction, all its write operations are visible to other transactions only after it commits. Here  $w_{0212}(y)$  occurs before  $w_{013}(y)$ . When  $t_{01}$  commits, it writes  $w_{013}(y)$  in  $t_0$ 's buffer. But  $t_{02}$  commits after  $t_{01}$  commits. When  $t_{02}$  commits it overwrites  $t_0$ 's  $y$  buffer with  $w_{0212}(y)$ .

To model these effects clearly, we augment a schedule with extra write operations. Prior to the commit event of a transaction, write operations are added to the schedule to represent the merging of its local buffers with its parent's buffers. We call these writes as *commit-write* operations. To every data buffer a committed transaction writes to (i.e. values written by a child or a descendent that has not aborted), there exists a commit-write operation. This write is the latest value on the data buffer. For example consider a transaction  $t_X$  consisting of operations  $w_{X1}(y)w_{X2}(z)w_{X3}(y)$  which it executes in this order and commits. Then in the schedule there is a commit-write operation for  $y$  and a commit-write for  $z$ .

In the above example  $t_X$  writes to data-item  $y$  twice. So its local data-buffer will hold the most recently written value. In this case the buffer holds the write of  $w_{X3}(y)$ . The commit-write operation for  $y$  writes the latest write operation i.e.  $w_{X3}(y)$ . We denote the commit-write for  $y$  as  $w_X^{X3}(y)$  and for  $z$  as  $w_X^{X2}(z)$ . The superscript provides the information about which child write operation this commit-write corresponds to. Since the local write buffers of an aborted transaction are not merged with its parent's buffer there are no commit-write operations corresponding to an aborted transaction. Using this notation we re-write the schedule in Example 2 as follows:

**Example 3**

$S2 : r_{0111}(x)w_{0112}(y)w_{011}^{0112}(y)c_{011}w_{012}(z)r_{0211}(z)w_{0212}(y)w_{021}^{0212}(y)c_{021}w_{013}(y)w_{01}^{012}(z)w_{01}^{013}(y)c_{01}r_{0221}(y)w_{0222}(z)a_{022}w_{02}^{021}(y)c_{02}r_{031}(y)r_{032}(z)w_{033}(z)w_{03}^{033}(z)c_{03}$

In the original computation tree only the leaves could write. With this augmentation of transactions even non-leaf nodes corresponding to committed transactions write with commit-write operations. For sake of brevity, we do not represent commit-writes in the computation tree. We assume that all the schedules we deal with are augmented with commit-writes.

It must be observed that a transaction's commit-write operation writes in its parent's buffers. For instance  $t_{021}$ 's commit-write  $w_{02}^{021}(y)$  writes in  $t_0$ 's  $y$  buffer (and not in  $t_{02}$ 's buffer). We denote the set of commit-writes of a committed transaction as *commit-set*. As opposed to commit-write we denote a simple memory write operation as a *simple-memory write*.

In our model a schedule has the complete information about the computation tree. Thus given a schedule we can obtain the entire computation tree from the subscripts of the events in it. Now consider two schedules  $S1$  and  $S2$ . If the sets of events in these schedules are the same then the computation trees represented by these schedules are the same. This is true irrespective of the ordering of the events in the schedules. The following property states it.

**Property 1** *Consider two schedules  $S1$  and  $S2$ . If the sets of events of the schedules are the same then the computation trees represented by the schedules are also the same and vice-versa. Formally,  $\langle S1, S2 : (S1.evts = S2.evts) \Leftrightarrow (\text{the computation trees of } S1 \text{ and } S2 \text{ are the same}) \rangle$*

Collectively, we refer to simple-memory operations and commit-write operations as memory operations. Since simple-memory operations are committed by default the commit-write notion can be extended to any tree node. Thus for any node  $n_X$  in a computation tree represented by a schedule  $S$ , we define

$$S.cwrite(n_X) = \begin{cases} n_X\text{'s commit-set} & n_X \text{ is a committed transaction} \\ nil & n_X \text{ is an aborted transaction} \\ n_X & n_X \text{ is a simple-memory write} \\ nil & n_X \text{ is a read operation} \end{cases}$$

With the introduction of commit-write operations we extend the definition of an operation, denoted as  $o_X$ , to represent either a transaction or a commit-write operation or a simple-memory operation. When we refer to a node on the computation tree, denoted as  $n_X$ , it is either a transaction or a simple-memory operation. Thus a node is also an operation. But an operation referring to a commit-write operation of a transaction is not a node since it is not part of the computation tree. We denote a memory operation (either commit-write or simple-memory operation) as  $m_X(y)$  or just  $m_X$  if the data-item is not important to the context.



We define two kinds of transactions: nested and non-nested. A non-nested transaction has only simple-memory operations as its children. A nested transaction has one or more nested or non-nested transactions and zero or more simple memory operations as its children.

## 2.2 Function Definitions

In this section we describe the functions used for describing our algorithm. All the functions pertain to the computation tree represented by a schedule  $S$ .

We define a function *holder* for an operation as:

$$S.holder(o_X) = \begin{cases} t_X & o_X \text{ is a commit-write belonging to } t_X, \\ o_X & o_X \text{ is a node of the tree} \end{cases}$$

The  $S.holder(o_X)$  is same as  $o_X$  when it is a transaction or a simple-memory operation. For any  $o_X$ , its holder maps it onto a node in the computation tree and thus will be denoted by  $n_X$ . In  $S2$  of Example 3,  $S2.holder(w_{021}^{0212})$  is  $t_{021}$ .

For any operation  $o_X$ , we define  $S.level(o_X)$  as the distance of  $S.holder(o_X)$  in the tree from the root. From this definition,  $t_0$  is at level 0. The level of a transaction and all its commit-write operations are the same. For instance in Example 3,  $S2.level(w_{021}^{0212}) = S2.level(t_{021}) = 2$ .

For a given tree node  $n_X$  (a transaction or a simple-memory operation) in the computation tree represented by the schedule  $S$ , we define:  $S.parent(n_X)$  as the parent of  $n_X$  on the tree,  $S.children(n_X)$  as children of  $n_X$  on the tree,  $S.desc(n_X)$  as the set of descendants of  $n_X$  on the tree and  $S.ansc(n_X)$  as the set of ancestors of  $n_X$  on the tree.

These functions can be extended to any operation  $o_X$  (including commit-write operation of transactions) by defining them for  $S.holder(o_X)$  over the tree. Thus by this extension the parent of a commit-write,  $m_X$ , of a transaction  $t_X$  is  $t_X$ 's parent in the tree. Similarly  $m_X$ 's children are  $t_X$ 's children. For instance in  $S2$  of Example 3,  $S2.parent(w_{021}^{0212}) = t_{02}$  and  $S2.children(w_{021}^{0212}) = \{r_{0211}(z), w_{0212}(y)\}$ . But it must be noted that  $S2.parent(r_{0211}(z))$  is  $t_{021}$  and not  $w_{021}^{0212}$ . Similarly these arguments can be extended to descendants and ancestors.

Consider two operations  $o_X, o_Y$ , in the computation tree represented by a schedule  $S$ . We define  $S.lca(o_X, o_Y)$  as the least common ancestor of  $S.holder(o_X)$  and  $S.holder(o_Y)$  in the computation tree of  $S$ .

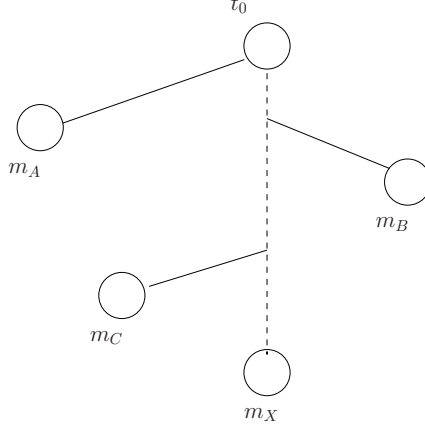
Next we define *dSet* function to be associated with every operation in the schedule  $S$ .

### Definition 1 (dSet)

$$S.dSet(o_X) = \begin{cases} o_X \cup \left( \bigcup_{n_Y \in S.children(o_X)} S.dSet(n_Y) \right) \cup S.cwrite(o_X) & o_X \text{ is a transaction,} \\ o_X & o_X \text{ is a simple-memory operation,} \\ S.dSet(S.holder(o_X)) & o_X \text{ is a commit-write} \end{cases}$$

Thus for a transaction  $t_X$  this function comprises of itself, its descendants, its commit-writes and all its descendants's commit-writes. By this definition we get that for any operation  $o_X$ ,  $S.dSet(o_X) = S.dSet(S.holder(o_X))$ . In Example 3,  $S2.dSet(t_{02}) = S2.dSet(w_{02}^{021}(y)) = \{t_{02}, r_{0211}(z), w_{0212}(y), w_{021}^{0212}(y), t_{021}, r_{0221}(y), w_{0222}(z), t_{022}, w_{02}^{021}(y)\}$

We have the following properties which follow from the definition of *dSet*:



**Figure 2:** This figures illustrates `optVis`. The dashed line represents the set of ancestors of  $m_X$ .

**Property 2** In the computation tree represented by a schedule  $S$ , for any operation  $o_X$  belonging to  $o_Y$ 's  $dSet$ , the level of  $o_X$  is greater than or equal to  $o_Y$ 's level. Formally,  
 $\langle S : (o_X \in S.dSet(o_Y)) \Rightarrow (S.level(o_X) \geq S.level(o_Y)) \rangle$

**Property 3** In the computation tree represented by a schedule  $S$ , if an operation  $o_X$  belongs to  $o_Y$ 's  $dSet$  and  $o_X, o_Y$  are at the same level then the holders of  $o_X, o_Y$  are the same. Formally,  
 $\langle S : (o_X \in S.dSet(o_Y)) \wedge (S.level(o_X) = S.level(o_Y)) \Rightarrow (S.holder(o_X) = S.holder(o_Y)) \rangle$

Next we define a peer function for an operation  $o_X$  in a schedule  $S$ :

$$S.peers(o_X) = \{o_Y | (S.holder(o_X) \neq S.holder(o_Y)) \wedge (S.parent(o_X) = S.parent(o_Y))\}$$

A transaction and all the elements of its commit-set are not peers of each other even though they all have the same parent. It is useful to view a transaction and all the elements of its commit-set as a single fused super node in the tree. From this definition we get that  $(o_X \in S.peers(o_Y)) \Rightarrow (o_Y \in S.peers(o_X))$  but  $(o_X \notin S.peers(o_X))$ . Consider two memory operations  $m_X(z), m_Y(z)$  operating on the same data-item. If they are peers, having the same parent say  $t_P$ , then they have access to the same data buffer  $z$  belonging to  $t_P$ .

Next we define a very useful function `optVis` on two operations  $o_X, o_Y$  in a schedule  $S$ , denoted as  $S.optVis(o_Y, o_X)$ . We will explain the significance of this function through the course of this document.

**Definition 2 (optVis)**

$$S.optVis(o_Y, o_X) = \begin{cases} true & o_Y \in (S.peers(o_X) \cup S.peers(S.ansc(o_X))) \\ false & otherwise \end{cases}$$

One can see that `optVis` function is not symmetrical. That is,  $S.optVis(o_Y, o_X)$  does not imply  $S.optVis(o_X, o_Y)$ . If  $S.optVis(o_Y, o_X)$  is true then we say that  $o_Y$  is `optVis` to  $o_X$  in  $S$ . It must also be noted that by the definition if  $(o_X \in S.dSet(o_Y))$  then  $S.optVis(o_Y, o_X)$  is false. As a result, for any commit-write of a transaction  $t_Y$ , say  $w_Y$ ,  $S.optVis(w_Y, t_Y)$  is false. It can also be seen that if  $S.optVis(o_Y, o_X)$  then the  $S.holder(o_Y)$  is not an ancestor of  $o_X$ .



Figure 2 illustrates optVis. Here the dashed line represents the set of ancestors of  $m_X$ . The operations  $m_A, m_B, m_C$  are peers of  $m_X$ 's ancestors. Hence they all are optVis to  $m_X$ .

In  $S2$  of Example 3, we have that  $S2.optVis(t_{01}, t_{02}) = S2.optVis(t_{02}, t_{03}) = S2.optVis(t_{03}, t_{01}) = true$  because  $t_{01}, t_{02}, t_{03}$  are peers of each other. Now looking at some subtle examples:  $S2.optVis(w_{01}^{013}(y), w_{03}^{033}(z))$  is true because  $w_{01}^{013}(y), w_{03}^{033}(z)$  are peers.  $S2.optVis(w_{01}^{012}(z), r_{0211}(z))$  is true as  $w_{01}^{012}(z)$  is a peer of  $t_{02}$  which is an ancestor of  $r_{0211}(z)$ . Similarly  $S2.optVis(t_{01}, t_{022})$  is true. But  $S2.optVis(r_{0211}(z), w_{01}^{012}(z))$  and  $S2.optVis(t_{022}, t_{01})$  are false. Also  $S2.optVis(w_{01}^{013}(y), w_{0112}(y))$  is false as  $w_{0112}(y)$  is in  $t_{01}$ 's dSet and  $w_{01}^{013}(y)$  is a commit-write of  $t_{01}$ . Similarly  $S2.optVis(w_{03}^{033}(z), r_{032}(z))$  is false. Next we define some properties and lemmas about optVis.

**Property 4** *In a schedule  $S$  if a memory operation (commit-write/simple-memory operation)  $m_Y$  is optVis to another memory operation  $m_X$  then  $m_X$ 's holder is a descendent of parent of  $m_Y$ . Formally*  
 $\langle S : S.optVis(m_Y, m_X) \Rightarrow (S.holder(m_X) \in S.desc(S.parent(m_Y))) \rangle$

**Property 5** *Consider two write operations  $w_Y, w_Z$  and a read operation  $r_X$  in a schedule  $S$ . If both  $w_Y, w_Z$  are optVis to  $r_X$  and are at the same level then  $w_Y, w_Z$  have the same parent. Formally,*  
 $(S.optVis(w_Y, r_X) \wedge S.optVis(w_Z, r_X) \wedge (S.level(w_Y) = S.level(w_Z))) \Rightarrow (S.parent(w_Y) = S.parent(w_Z))$

**Lemma 6** *Consider two schedules  $S1$  and  $S2$  such that both of them have the same set of events. Suppose for two events  $o_Y$  and  $o_X$ ,  $o_Y$  is optVis to  $o_X$  in  $S1$ . Then  $o_Y$  is optVis to  $o_X$  in  $S2$  as well. Formally,*  
 $\langle S1, S2 : \{o_X, o_Y\} \in S1.evts : (S1.evts = S2.evts) \wedge (S1.optVis(o_Y, o_X)) \Rightarrow (S2.optVis(o_Y, o_X)) \rangle$

**Proof:** Since the events of  $S1$  and  $S2$  are the same, from Property 1, we get that the computation trees of  $S1$  and  $S2$  are the same. In  $S1$ ,  $o_Y$  is optVis to  $o_X$ . This implies that  $o_Y$  is either a peer of  $o_X$  or a peer of an ancestor of  $o_X$  in the computation tree of  $S1$ . Since the computation tree of  $S2$  is the same as that of  $S1$ ,  $o_Y$  is either a peer of  $o_X$  or a peer of an ancestor of  $o_X$  in the computation tree of  $S2$  as well. Hence  $o_Y$  is optVis to  $o_X$  in  $S2$  also. Thus we have  $S2.optVis(o_Y, o_X)$ .  $\square$

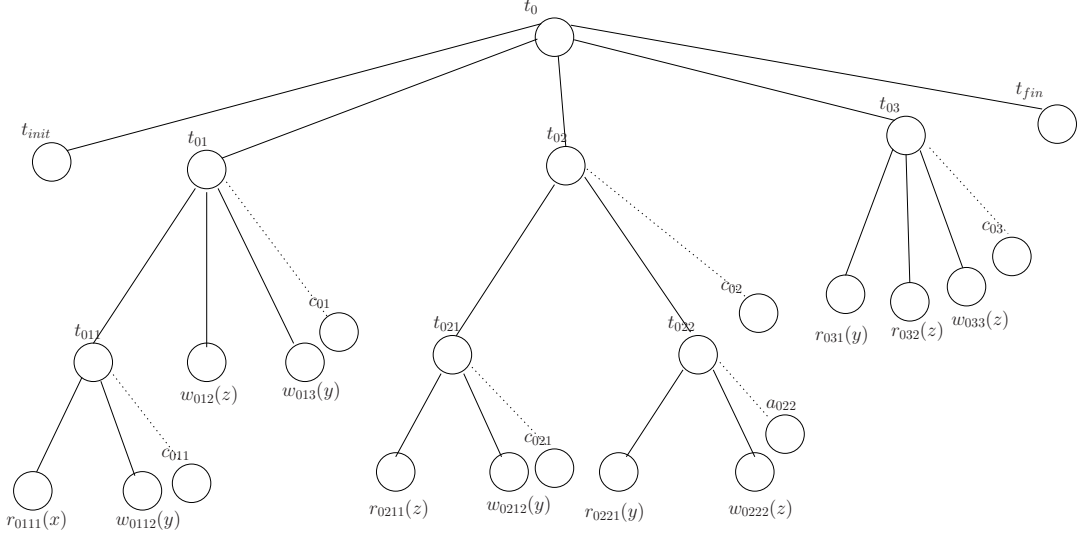
### 2.3 Serial Schedules for Closed Nested Transactions

In this section we talk about serial schedules in the context of nested transactions.

**Schedule Partial Order:** A schedule totally orders all the events of a transaction. Further it partially orders all the transactions and simple-memory operations. For a schedule  $S$  and a transaction  $t_X$  in it, we define  $S.t_X.first$  as the first operation of  $t_X$  that executes according to  $S$ . Similarly we define  $S.t_X.last$  as the last operation of  $t_X$  (i.e., a terminal operation) to execute according to  $S$ . For a simple-memory operation,  $S.m_X.first = S.m_X.last$ . With these definitions we can define a partial order on all the nodes in the computation tree represented by the schedule:  $(n_X <_S n_Y) \equiv (S.n_X.last < S.n_Y.first)$

We call this order as the *schedule-partial-order*. It must be noted that all the memory operations having the same parent are totally ordered.

**Serial Schedules:** For non-nested transactions a serial schedule is a schedule in which all the transactions execute serially (as the name suggests) without any interleaving. Serial schedules are very useful because their executions are easy to verify since there is no interleaving. For a closed nested STM system we define a serial schedule as follows:



**Figure 3: Computation tree for a serial schedule of  $S2$**

**Definition 3** A schedule  $SS$  is called serial if for every transaction  $t_X$  in  $SS$ , the children (both transactions and simple-memory operations) of  $SS$  are totally ordered. Formally,  
 $\langle \forall t_X \in SS.nodes : \{n_Y, n_Z\} \subseteq SS.children(t_X) : (n_Y <_{SS} n_Z) \vee (n_Z <_{SS} n_Y) \rangle$

A serial schedule with the same events of the schedule  $S2$  is as follows:  $r_{0111}(x)w_{0112}(y)w_{0111}^{0112}(y)c_{011}w_{012}(z)w_{013}(y)w_{01}^{012}(z)w_{01}^{013}(y)c_{01}r_{0211}(z)w_{0212}(y)w_{021}^{0212}(y)c_{021}r_{0221}(y)w_{0222}(z)a_{022}w_{02}^{021}(y)c_{02}r_{031}(y)r_{032}(z)w_{033}(z)w_{03}^{033}(z)c_{03}$ . The computation tree for this serial schedule is shown in Figure 3. From the definition of a serial schedule we get the following property:

**Property 7** Consider two peer nodes,  $n_X, n_Y$  in a serial schedule  $SS$ . Let  $m_R$  be a memory operation belonging to  $n_X$ 's  $dSet$  and  $m_S$  be a memory operation belonging to  $n_Y$ 's  $dSet$ . If  $m_R$  occurs before  $m_S$  in  $SS$ , then all the memory operations in  $n_X$ 's  $dSet$  occur before all the memory operations of  $n_Y$ 's  $dSet$ . Formally,

$$\langle \{n_X, n_Y\} \in SS.nodes : (m_R \in SS.dSet(n_X)) \wedge (m_S \in SS.dSet(n_Y)) : (SS.parent(n_X) = SS.parent(n_Y)) \wedge (SS.is\ serial) \wedge (SS.ord(m_R) < SS.ord(m_S)) \Rightarrow (\forall m_P, \forall m_Q : (m_P \in SS.dSet(n_X)) \wedge (m_Q \in SS.dSet(n_Y)) : (SS.ord(m_P) < SS.ord(m_Q))) \rangle$$

## 2.4 Writes for Read Operations and Well-Formedness

In an earlier sub-section we described how write operations are performed by transactions. Now let us understand how read operations are performed. In our model, for non-nested transactions there exists only a single version for each data-item  $z$ . Thus, each read operation in non-nested transaction reads the latest value in the buffer of  $z$ . This implies that in a schedule  $S$  consisting of non-nested transactions, a read operation  $r_X(z)$  reads the value written by the previous closest commit-write on  $z$ . We call such a write as the *lastWrite* of  $r_X(z)$ <sup>1</sup> and denote it as  $S.lastWrite(r_X(z))$ .

<sup>1</sup>This term is inspired from [12]

Now coming to the case of nested transactions, a transaction wanting to read a data-item, say  $z$ , (unlike write) has access to  $z$  data buffers of all its ancestors (apart from its own). But it does not have access to its children's buffers. Hence, the lastWrite could potentially be the most recent previous write in any one of these buffers. Thus, we must identify rules to precisely define lastWrite of a read operation for nested transactions.

To understand this, we consider the serial (nested) schedule of  $S_2$  shown in Figure 3. Consider the read operation  $r_{0211}(z)$  of transaction  $t_{021}$ . Transaction  $t_{021}$  has access to its own buffers, the buffers of its parent  $t_{02}$  and the buffers of root  $t_0$ . In this case,  $r_{0211}(z)$  reads from the  $z$  buffer with the value written by commit-write  $w_{01}^{012}(z)$ . Next, consider the read operation  $r_{0221}(y)$  of transaction  $t_{022}$ . Similar to the above case,  $t_{022}$  has access to its own buffers, the buffers of its parent  $t_{02}$  and the buffers of root  $t_0$ . Thus  $r_{0221}(y)$  has the option of reading the value written by commit-write  $w_{021}^{0212}(y)$  onto  $t_{02}$ 's  $y$  buffer or by the commit-write  $w_{01}^{013}(y)$  onto  $t_0$ 's  $y$  buffer. In our model,  $r_{0221}(y)$  reads the value written by  $w_{021}^{0212}(y)$  onto  $t_{02}$ 's buffer.

Having illustrated with examples from a serial schedule, we define the properties that a lastWrite of a read operation in a schedule (which is not necessarily serial) must satisfy:

1. The lastWrite  $w_Y$  should occur prior to the read operation  $r_X$  in the schedule.
2. The lastWrite  $w_Y$  should be a commit-write belonging to a committed transaction or a simple-memory operation. Since the read operation can access the  $z$  data buffers of all its ancestors, the commit-write on  $z$  should be a peer of  $r_X(z)$  or a peer of an ancestor of  $r_X(z)$ , i.e.,  $S.optVis(w_Y, r_X)$  should be true.
3. The read operation  $r_X(z)$  accesses  $z$  buffers starting from that of its own transaction. It then accesses its ancestor's  $z$  buffer in the decreasing order of level. It reads from the first buffer which has a non- $\perp$  value in it. Thus the lastWrite  $w_Y$  is such that the difference between its level and  $r_X$ 's level is the smallest.
4. If there are multiple writes satisfying the above conditions then among these writes the lastWrite  $w_Y$  is the closest to  $r_X$  in the schedule  $S$ .

As mentioned earlier when a new sub-transaction is invoked (by a parent transaction), the sub-transaction creates a separate set of buffers for each data-item it accesses. On creation these buffers are initialized with  $\perp$ .

It can be seen that applying these rules to non-nested transactions, the lastWrite for a read  $r_X(z)$  boils down to the previous closest commit-write or a simple-memory operation.

A nested transaction  $t_N$  wanting to read  $z$  can satisfy the above mentioned properties of lastWrite of  $r_N(z)$  by following a simple procedure:  $t_N$  first reads its local  $z$  buffer. If the value read from its buffer is  $\perp$  then it reads from its parent's  $z$  buffer. If that is also  $\perp$ , it then reads the buffer of the parent of the parent and so on. It reads the  $z$  buffers in this way until it reads a non- $\perp$  value. Since  $t_0$ 's buffers have been initialized,  $t_N$  will eventually read a non- $\perp$  value.

Next we will formally describe the notion of lastWrite in a sequence of steps. We consider the following schedule to describe our definitions. The computation tree for this schedule is in Figure 4.

#### Example 4

*Computation Tree:*

$$t_0 : \{t_{init}, t_{01}, t_{02}, t_{03}, t_{fin}\},$$

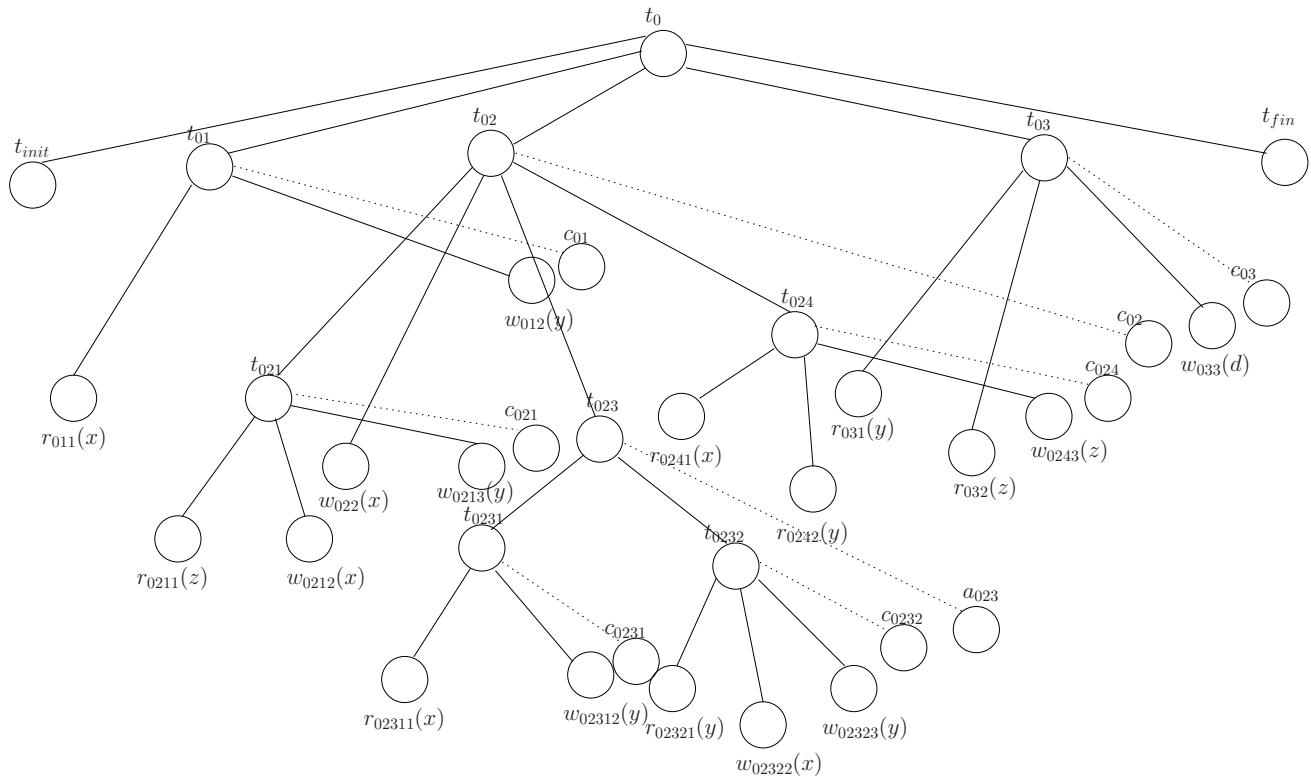


Figure 4: Computation tree for Example 4

$t_{01} : \{sm_{011} = r_{011}(x), sm_{012} = w_{012}(y), c_{01}\},$   
 $t_{02} : \{t_{021}, sm_{022} = w_{022}(x), t_{023}, t_{024}, c_{02}\},$   
 $t_{021} : \{sm_{0211} = r_{0211}(z), sm_{0212} = w_{0212}(x), sm_{0213} = w_{0213}(y), c_{021}\},$   
 $t_{023} : \{t_{0231}, t_{0232}, a_{023}\},$   
 $t_{0231} : \{sm_{02311} = r_{02311}(x), sm_{02312} = w_{02312}(y), c_{0231}\},$   
 $t_{0232} : \{sm_{02321} = r_{02321}(y), sm_{02322} = w_{02322}(x), sm_{02323} = w_{02323}(y), c_{0232}\},$   
 $t_{024} : \{sm_{0241} = r_{0241}(x), sm_{0242} = r_{0242}(y), sm_{0243} = w_{0243}(z), c_{024}\},$   
 $t_{03} : \{sm_{031} = r_{031}(y), sm_{032} = r_{032}(z), sm_{033} = w_{033}(d), c_{03}\},$

*Schedule:*

$S3 : r_{011}(x)r_{0211}(z)w_{0212}(x)w_{022}(x)r_{02311}(x)w_{0213}(y)w_{021}^{0212}(x)w_{021}^{0213}(y)c_{021}w_{012}(y)w_{01}^{012}(y)c_{01}w_{02312}(y)$   
 $w_{0231}^{02312}(y)c_{0231}r_{02321}(y)r_{0241}(x)w_{02322}(x)r_{0242}(y)r_{031}(y)w_{02323}(y)w_{0232}^{02322}(x)w_{0232}^{02323}(y)c_{0232}r_{032}(z)a_{023}$   
 $w_{0243}(z)w_{024}^{0243}(z)c_{024}w_{02}^{021}(x)w_{02}^{021}(y)w_{02}^{024}(z)c_{02}w_{033}(d)w_{03}^{033}(d)c_{03}$

It must be noted that in the schedule  $S3$  transaction  $t_{023}$  is aborted. But both its child transactions  $t_{0231}, t_{0232}$  are committed.

For two memory operations in a schedule we define two kinds of distances. We define  $schDist$  as  $S.schDist(m_X, m_Y) = |S.ord(m_X) - S.ord(m_Y)|$ .

Next we define  $levDist$  as  $S.levDist(m_X, m_Y) = |level(m_X) - level(m_Y)|$ . For a memory operation  $m_X(y)$  in  $S$ , we define the following sets:

$$S.prevW(m_X(y)) = \{w_Y(y) | (w_Y(y) \in S.evts) \wedge (S.ord(w_Y(y)) < S.ord(m_X(y)))\}$$

As the name suggests the set  $prevW$  consists of all the  $y$  writes that happen before  $m_X(y)$  in  $S$  irrespective of whether they are simple write or commit-write operations.

$$S.prevVisW(m_X(y)) = \{w_Y(y) | (w_Y(y) \in S.prevW(m_X(y))) \wedge (S.optVis(w_Y(y), m_X(y)))\}$$

This set consists of all the  $y$  writes that occur before  $m_X(y)$  and are  $optVis$  to  $m_X(y)$ . Since the transaction  $t_{init}$  is a child of  $t_0$ ,  $t_{init}$  is  $optVis$  to every other operation in the computation. Hence the set  $prevVisW$  of every memory operation will contain a write by  $t_{init}$ . As a result, the  $prevVisW$  of every memory operation has at least one element. For instance in the schedule  $S3$  mentioned in Example 4,

$$S3.prevVisW(r_{0241}(x)) = \{w_{init}(x), w_{021}^{0212}(x), w_{022}(x)\}$$

$$S3.prevVisW(r_{0242}(y)) = \{w_{init}(y), w_{021}^{0213}(y), w_{01}^{012}(y)\}$$

$$S3.prevVisW(r_{02321}(y)) = \{w_{init}(y), w_{021}^{0213}(y), w_{01}^{012}(y), w_{0231}^{02312}(y)\}$$

Next we define a set having all the writes that occur before a memory operation, are  $optVis$  to it and are closest to it in terms of level.

$$S.prevCloseSet(m_X(y)) = \{w_Y(y) | (w_Y(y) \in S.prevVisW(r_X(y))) \wedge (S.levDist(w_Y(y), m_X(y)) \text{ is smallest})\}$$

For instance, for the writes of schedule  $S3$  in Example 4 mentioned above,

$$S3.prevCloseSet(r_{0241}(x)) = \{w_{021}^{0212}(x), w_{022}(x)\}$$

$$S3.prevCloseSet(r_{0242}(y)) = \{w_{021}^{0213}(y)\}$$

$$S3.prevCloseSet(r_{02321}(y)) = \{w_{0231}^{02312}(y)\}$$

Having defined these sets, we define the  $lastWrite$  for a read operation in a schedule as the closest write

operation from the `prevCloseSet` set. Formally,

$$S.lastWrite(m_X(y)) = \{w_Y(y) \mid (w_Y(y) \in S.prevCloseSet(m_X(y))) \wedge (S.schDist(m_X(y), w_Y(y)) \text{ is minimum})\}$$

Since the set `prevVisW` has at least one element, `lastWrite` is never nil. In the worst case a read operation will read the values written by  $t_{init}$ . The lastWrites for all the reads in  $S3$  of Example 4 are as follows:

$$\{r_{011}(x) : w_{init}(x), r_{0211}(z) : w_{init}(z), r_{02311}(x) : w_{022}(x), r_{02321}(y) : w_{0231}^{02312}(y), r_{0241}(x) : w_{021}^{0212}(x), r_{0242}(y) : w_{021}^{0213}(y), r_{031}(y) : w_{01}^{012}(y), r_{032}(z) : w_{init}(z)\}$$

An important requirement of a STM is that no transaction reads from an aborted transaction. Intuitively this implies that the lastWrite of no read operation belongs to an aborted transaction's dSet. Consider the read  $r_{02321}(y)$ . Its lastWrite is  $w_{0231}^{02312}(y)$  which belongs to  $t_{023}$ 's dSet. Transaction  $t_{023}$  is aborted. In this case it might seem that the read  $r_{02321}(y)$  is reading from an aborted transaction. But  $w_{0231}^{02312}(y)$  actually belongs to  $t_{0231}$ 's dSet which is a committed transaction. Further  $r_{02321}(y)$  also belongs to  $t_{023}$ . Thus the properties that we want of aborted transactions have not been violated. We have the following property and lemma which formalizes this notion:

**Property 8** Consider a schedule  $S$  which has a read  $r_X$ . Let the lastWrite of  $r_X$  be  $w_Y$ . Then the holder of  $w_Y$  cannot be an aborted transaction. Formally,

$$\langle S : r_X \in S.evts : (w_Y = S.lastWrite(r_X)) \Rightarrow (S.holder(w_Y) \text{ is not aborted}) \rangle$$

**Lemma 9** Consider a schedule  $S$  which has a read  $r_X$ . Let the lastWrite of  $r_X$  be  $w_Y$ . If an ancestor of  $w_Y$ , say  $t_A$ , is aborted then  $r_X$  is in  $t_A$ 's dSet. Formally,

$$\langle S : r_X \in S.evts, t_A \in S.nodes : (w_Y = S.lastWrite(r_X)) \wedge (t_A \in S.ansc(w_Y)) \wedge (t_A \text{ is aborted}) \Rightarrow (r_X \in S.dSet(t_A)) \rangle$$

**Proof:** Let the parent of  $w_Y$  be  $t_P$ . From Property 4, we get that  $t_P$  is an ancestor of  $r_X$ . Hence, any ancestor of  $w_Y$  is an ancestor of  $r_X$ . This implies that  $t_A$  is an ancestor of  $r_X$ . Thus,  $r_X$  is in the dSet of  $t_A$ .  $\square$

Informally this lemma implies that no transaction outside an aborted transaction reads from it. Consider the read operation  $r_{0242}(y)$  in  $S3$  of Example 4. Its lastWrite is  $w_{021}^{0213}(y)$ . But in  $S3$  there is a write  $w_{01}^{012}(y)$  which is `optVis` to  $r_{0242}(y)$  and occurs before it. Moreover  $w_{01}^{012}$  is closer to  $r_{0242}(y)$  in `schDist` than  $w_{021}^{0213}(y)$  i.e.  $S3.schDist(r_{0242}(y), w_{021}^{0213}(y)) > S3.schDist(r_{0242}(y), w_{01}^{012}(y))$ . So intuitively it might seem that  $w_{01}^{012}(y)$  should be the lastWrite. But  $w_{021}^{0213}(y)$  is closer to  $r_{0242}(y)$  in terms of level than  $w_{01}^{012}$  (condition 3 of the properties required by `lastWrite`) i.e.  $S3.levDist(r_{0242}(y), w_{021}^{0213}(y)) < S3.levDist(r_{0242}(y), w_{01}^{012}(y))$ . Hence  $w_{021}^{0213}(y)$  is the lastWrite. The following two properties describe this notion formally,

**Property 10** Consider a schedule  $S$  with memory operations  $w_Z, w_Y, r_X$  such that  $w_Z$  occurs prior to  $w_Y$  in  $S$ ,  $w_Z$  is `optVis` to  $r_X$  and  $w_Y$  is the lastWrite of  $r_Y$ . Then  $w_Z$ 's level should be less than or equal to  $w_Y$ 's level. Formally,

$$\langle S : \{w_Z, w_Y, r_X\} \in S.evts : (S.ord(w_Z) < S.ord(r_X)) \wedge S.optVis(w_Z, r_X) \wedge (w_Y = S.lastWrite(r_X)) \Rightarrow (S.level(w_Z) \leq S.level(w_Y)) \rangle$$

**Property 11** Consider a schedule  $S$  with memory operations  $w_Z, w_Y, r_X$  such that  $w_Z$ 's level is same as  $w_Y$ 's level,  $w_Z$  occurs prior to  $r_X$  in  $S$ ,  $w_Z$  is `optVis` to  $r_X$  and  $w_Y$  is the lastWrite of  $r_X$ . Then  $w_Z$  also



occurs prior to  $w_Y$  in  $S$ . Formally,

$$\langle S : \{w_Z, w_Y, r_X\} \in S.evts : (S.level(w_Z) = S.level(w_Y)) \wedge (S.ord(w_Z) < S.ord(r_X)) \wedge S.optVis(w_Z, r_X) \wedge (w_Y = S.lastWrite(r_X)) \Rightarrow (S.ord(w_Z) < S.ord(w_Y) < S.ord(r_X)) \rangle$$

We would like to make a note about the definition of `optVis`. Consider a read operation  $r_X(z)$  and a committed transaction  $t_Y$  in a schedule  $S$ . Let  $r_X$  be in  $t_Y$ 's `dSet`. Then by our convention all the commit-writes of  $t_Y$  occur after  $r_X$  has executed in the  $S$ . Thus no commit-write of  $t_Y$  can be the `lastWrite` of  $r_X$ . Due to this property we defined `optVis` such that any write  $w_Y$  is not `optVis` to  $r_X$  if  $r_X$  is contained in  $t_Y$ 's `dSet`. Formally,

$$\langle S : \{r_X, w_Y, t_Y\} \in S.evts : (r_X \in S.dSet(t_Y)) \wedge (w_Y \in t_Y\text{'s commit-set}) \Rightarrow (S.optVis(w_Y, r_X) = false) \rangle$$

For a node  $n_P$  with a read operation  $r_X$  in its `dSet`, the read is said to be an *external-read* of  $n_P$  if its `lastWrite` is not in  $n_P$ 's `dSet`. For instance,  $r_{0241}(x)$  is an external-read of  $t_{024}$  since its `lastWrite`  $w_{021}^{0212}(x)$  is not in  $t_{024}$ 's `dSet`. The read  $r_{02321}(y)$  is not an external-read of the transaction  $t_{023}$  since its `lastWrite`  $w_{0231}^{02312}(y)$  belongs to  $t_{023}$ 's `dSet`. From this definition we get that every read operation is an external-read of itself. Thus,  $r_{0241}(x)$  is an external-read of itself. It can be seen that a nested transaction interacts with its peers through external-reads and commit-writes. Thus, a nested transaction can be treated as a non-nested transaction consisting only of its external-reads and commit-writes. The external-reads and commit-writes of a transaction constitute its *extOpsSet*.

A schedule is called *well-formed* if it satisfies: (1) Validity of Transaction limits: After a transaction executes a terminal operation no operation (memory or terminal) belonging to it can execute; and (2) Validity of Read Operations: Every read operation reads the value written by its `lastWrite` operation.

We assume that all the schedules we deal with are well-formed.

### 3 Conflict Preserving Closed Nested Opacity

In this section, we (1) define opacity for closed nested transactions, represented by a class of schedules *CNO*, (2) present a new conflict notion *optConf* for closed nested transactions (3) define *CP-CNO*, a subclass of *CNO* based on *optConf* and then (4) present an algorithm for verifying the membership of *CP-CNO* in polynomial time.

#### 3.1 Closed Nested Opacity

A STM system allows interleaving between transactions to efficiently utilize the system resources. But the STM system should also ensure that the interleaving transactions execute in correct manner. In the context of traditional databases the correctness criterion for the execution of concurrent transactions is *serializability* [19]. Serializability ensures that the execution of all the committed transactions corresponds to a serial execution. But serializability does not specify the correctness of aborted transactions. In STM systems where transactions execute in memory it is imperative that all transactions including aborted transactions execute correctly. Incorrect execution of aborted transactions could result in the STM system entering into an inconsistent state. This could result in many errors such as crash failures, division-by-zero etc. as described in Section 1.

To address this shortcoming Guerraoui and Kapalka [5] came up with the notion of *opacity*. A schedule, consisting of an execution of transactions, is said to be *opaque* if there is an equivalent serial schedule such that it respects the original schedule's schedule-partial-order and the `lastWrite` for every read operation (including the reads of aborted transactions) in the serial schedule is the same as in the original schedule.

To effectively capture this notion, Imbs and Raynal [9] treat all the aborted transactions in a given schedule as read-only transactions. Then in the resulting schedule they try to find an equivalent serial schedule satisfying the above mentioned conditions. We extend this notion to nested transactions. That is, in our characterization, all aborted sub-transactions are viewed as read-only transactions. We define a class of schedules called as Closed Nested Opacity or CNO as follows

**Definition 4** A schedule  $S$  belongs to Closed Nested Opacity (CNO) class if there exists a serial schedule  $SS$  such that:

1. *Event Equivalence*: The events of  $S$  and  $SS$  are the same. Formally,  $\langle\langle S.evts = SS.evts \rangle\rangle$
2. *schedule-partial-order Equivalence*: For any two nodes  $n_Y, n_Z$  that are peers in the computation tree represented by  $S$  if  $n_Y$  occurs before  $n_Z$  in  $S$  then  $n_Y$  occurs before  $n_Z$  in  $SS$  as well. Formally,  $\langle t_X : \{n_Y, n_Z\} \subseteq S.children(t_X) : (n_Y <_S n_Z) \Rightarrow (n_Y <_{SS} n_Z) \rangle$
3. *lastWrite Equivalence*: For all read operations the lastWrites in  $S$  and  $SS$  are the same. Formally,  $\langle S, SS : \forall r_X : S.lastWrite(r_X) = SS.lastWrite(r_X) \rangle$

Even though the definition of CNO is similar to opacity, the condition lastWrite equivalence captures the intricacies of nested transactions. This class ensures that the reads of all the transactions including all the sub-transactions of aborted transactions read consistent values. We denote this equivalence between a schedule  $S$  and a serial schedule  $SS$  as  $S \approx_o SS$ .

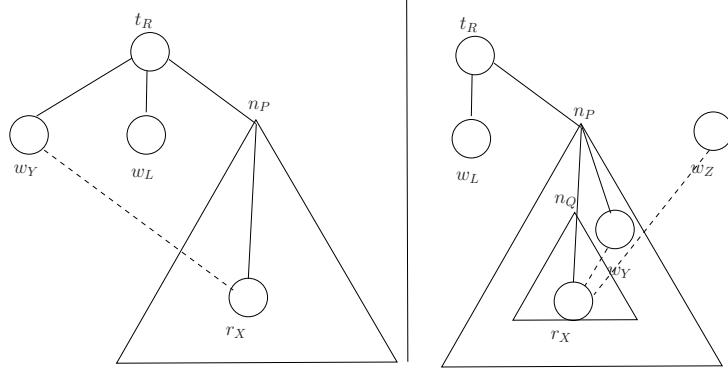
### 3.2 Conflict Notion: optConf

Checking opacity, like general serializability (for instance, view-serializability) cannot be done efficiently. Restricted classes of serializability (like conflict-serializability) have been defined based on conflicts which allow polynomial membership test, and facilitate online scheduling. Along the same lines, we define a subclass of CNO, CP-CNO.

This subclass is based on the notion of conflicts. Two memory operations operating on the same data-item are said to be in conflict if one of them is a write operation (and the other is either a read or write operation). We extend the notion of conflicts to closed nested transactions. We call this conflict notion as *optConf* (conflict for optimistic executions). It is tailored for closed-nested nature of transactions. This notion is similar to the idea of conflicts presented in [4] for non-nested transactions. In this section we present *Conflict Preserving Closed Nested Opacity* or *CP-CNO* a subclass of CNO based on optConf notion for closed nested transactions.

Consider a schedule  $S$  and a serial schedule  $SS$  with the same set of events as  $S$ . We show that, if the set of optConflicts between the events in  $S$  are also in  $SS$ , then the lastWrite for every read is also the same in  $S$  and  $SS$ . It must be noted that since the set of events (and transactions) are the same in  $S$  and  $SS$ , from Property 1 we get that their computation trees are also the same. As a result if an operation  $o_X$  is at level  $l_X$  in  $S$ , then its level in  $SS$  is also  $l_X$ .

The conflict notion optConf is defined only between memory operations in extOpsSets (defined in Section 2.4) of two peer nodes. As explained earlier, a node (or transaction) interacts with its peer nodes through its extOpsSet. Consider two peer nodes  $n_A, n_B$ . For two memory operations  $m_X, m_Y$  in the extOpsSets of  $n_A, n_B$ ,  $S.optConf(m_X, m_Y)$  is true if  $m_X$  occurs before  $m_Y$  in  $S$  and one of the following conditions hold:



**Figure 5: Example illustrating w-r and r-w conflicts**

1. w-r optConf:  $m_X$  is a commit-write  $w_X$  of  $n_A$  and  $m_Y$  is an external-read  $r_Y$  of  $n_B$  or
2. r-w optConf:  $m_X$  is an external-read  $r_X$  in  $n_A$ ,  $m_Y$  is a commit-write  $w_Y$  of  $n_B$  or
3. w-w optConf:  $m_X$  is a commit-write  $w_X$  of  $n_A$  and  $m_Y$  is a commit-write  $w_Y$  of  $n_B$ .

Figure 5 illustrates the conflicts for a read  $r_X$ . Here  $w_L$  and  $n_P$  are peers with  $r_X$  in  $n_P$ 's dSet and the lastWrite of  $r_X$  is  $w_L$ . In the figure on the left,  $w_Y$  is a peer of  $w_L$  and  $n_P$ . This figure illustrates w-r conflict between  $w_Y$  and  $r_X$ . The dotted line shows the conflict.

The figure on the right of Figure 5 illustrates r-w conflict. In this figure,  $w_Y$  belongs to  $n_P$ 's dSet and is a peer of  $n_Q$ . The commit-write  $w_Z$  is a peer of  $n_P$ . The read  $r_X$  is in  $n_Q$ 's dSet and in  $n_P$ 's dSet with  $n_P$  being an ancestor of  $n_Q$ . Since  $r_X$ 's lastWrite is not in  $n_Q$ 's dSet and also not in  $n_P$ 's dSet, it is an external-read of both  $n_Q$  and  $n_P$ . Hence  $r_X$  is in r-w optConf with both  $w_Y$  and  $w_Z$ .

Next, we will motivate the reason for defining the conflicts in this manner. Consider a read  $r_X(d)$  in a schedule  $S$  with lastWrite as  $w_L(d)$ . Let  $w_A(d)$  be an arbitrary write in  $S$  that is optVis to  $r_X(d)$ . Let their levels be  $l_X, l_L, l_A$  respectively. From optVis definition we get that,  $l_L \leq l_X$  and  $l_A \leq l_X$  and these relationships hold in  $SS$  as well (since the set of events in  $S$  and  $SS$  are the same). The conflicts are defined such that  $w_A$  does not become  $r_X$ 's lastWrite in any conflict equivalent serial schedule  $SS$ . The following paragraphs explain this.

For  $w_L$  to be the lastWrite of  $r_X$  in  $SS$ ,  $w_L$  must occur before  $r_X$  in  $SS$  as well. This is ensured by w-r optConf. Next, let us analyse the motive of r-w optConf. From the definition of lastWrite, we get that if  $l_A < l_L$  (i.e.  $w_A$  is closer to the root than  $w_L$ ) then  $w_A$  can never be the lastWrite of  $r_X$  in  $SS$ . Hence, it suffices to define r-w conflict only between the read  $r_X$  and any such  $w_A$  whose level  $l_A$  is greater than or equal to  $l_L$ . We do not need to consider conflicts between read and writes that are at level smaller than its lastWrite (i.e. closer to the root than the lastWrite).

Consider the case that  $l_A \geq l_L$ . Consider two peer nodes  $n_P, n_Q$  (which are at the same level in the tree since they are peers). Let  $r_X$  be in  $n_P$ 's dSet and  $w_A$  in  $n_Q$ 's dSet. Also, let  $r_X$  occur before  $w_A$  in  $S$ . Since  $w_A$  is optVis to  $r_X$ ,  $w_A$  must be  $n_Q$ 's commit-write (if  $n_Q$  is a simple-memory operation then it is same as  $w_A$ ). As a result, the levels of  $n_P, n_Q$  and  $w_A$  are the same. From our assumption, we have that  $w_A$ 's level is greater than or equal to  $l_L$ . Hence,  $n_P$ 's level is greater than or equal to  $l_L$  as well. From Property 2 we get that,  $r_X$ 's lastWrite  $w_L$  cannot be in  $n_P$ 's dSet. As a result,  $r_X$  is an external-read of  $n_P$ . Thus by

defining r-w conflict between such  $r_X$  and  $w_A$  we ensure that  $w_A$  can never be  $r_X$ 's lastWrite in any conflict equivalent serial schedule  $SS$ .

Now consider the case that the write  $w_A$  occurs before  $w_L$  and  $r_X$  in  $S$ . Let  $w_A$ 's level  $l_A$  be same as  $l_L$ . Combining this with the observation that  $w_A$  and  $w_L$  are optVis to  $r_X$ , we get that  $w_A$  and  $w_L$  are peers. It must be noted that w-r conflict ensures that  $w_A$  occurs before  $r_X$  in  $SS$ . But it is possible that  $w_A$  occurs between  $w_L$  and  $r_X$  in  $SS$ . Then  $w_A$  becomes  $r_X$ 's lastWrite in  $SS$ . The w-w conflict between  $w_A$  and  $w_L$  ensures that  $w_A$  occurs before  $w_L$  in  $SS$  as well. Thus all the three conflicts ensure that  $w_L$  is  $r_X$ 's lastWrite in  $SS$  as well.

The set of conflicts for the schedule  $S3$  mentioned in Example 4 are:

$$\{(r_{011}(x), w_{02}^{021}(x)), (r_{0211}(z), w_{024}^{0243}(z)), (w_{022}(x), w_{021}^{0212}(x)), (w_{022}(x), r_{02311}(x)), (r_{02311}(x), w_{021}^{0212}(x)), (r_{02311}(x), w_{0232}^{02322}(x)), (w_{021}^{0213}(y), r_{0242}(y)), (w_{01}^{012}(y), r_{031}(y)), (w_{01}^{012}(y), w_{02}^{021}(y)), (r_{02311}(x), w_{021}^{0212}(x)), (w_{0231}^{02312}(y), r_{02321}(y)), (w_{0231}^{02312}(y), w_{0232}^{02323}(y)), (w_{021}^{0212}(x), r_{0241}(x)), (w_{022}(x), r_{0241}(x)), (r_{031}(y), w_{02}^{021}(y)), (r_{032}(z), w_{02}^{024}(z))\}$$

The conflicts involving  $t_{init}$  and  $t_{fin}$  are not shown here. Now, we describe a property about w-r conflict and a lemma about r-w conflict,

**Property 12** *If the lastWrite of read  $r_X$  in  $S$  is  $w_Y$  then  $w_Y$  and  $r_X$  are in w-r optConf. Formally,*  
 $\langle (w_Y = S.lastWrite(r_X)) \Rightarrow (S.optConf(w_Y, r_X)) \rangle$

**Lemma 13** *Consider a write  $w_A$  and a read  $r_X$  in a schedule  $S$ . Let  $r_X$ 's lastWrite be  $w_L$ . Let the levels of  $r_X, w_A, w_L$  be  $l_X, l_A, l_L$  respectively. If  $l_L$  is less than or equal to  $l_A$  and  $w_A$  is optVis to  $r_X$  and  $r_X$  occurs before  $w_A$  in  $S$  then  $S.optConf(r_X, w_A)$  is true. Formally,*

$$(w_L = S.lastWrite(r_X)) \wedge (l_L \leq l_A) \wedge (S.optVis(w_A, r_X)) \wedge (S.ord(r_X) < S.ord(w_A)) \Rightarrow (S.optConf(r_X, w_A))$$

**Proof:** Let holder of  $w_A$  be  $n_A$  (which is same as  $w_A$ , if it is a simple-write). Since  $w_A$  is optVis to  $r_X$ , there is a peer  $n_B$  of  $n_A$  such that  $r_X$  is in  $n_B$ 's dSet. Since  $n_A, n_B$  are peers we get that  $level(w_A) = level(n_A) = level(n_B) = l_A$ . Here we have two cases depending on the levels of  $w_L$  and  $w_A$ .

Case 1  $l_L < l_A$ : This case implies that  $l_L < level(n_B)$ . Combining this with the contrapositive of Property 2, we get that  $w_L$  is not in  $n_B$ 's dSet. But  $r_X$  is in  $n_B$ 's dSet. Hence  $r_X$  is an external-read of  $n_B$ .

Case 2  $l_L = l_A$ : This case implies that  $l_L = level(n_B)$ . Consider the case that  $w_L$  is in  $n_B$ 's dSet. Then from Property 3, we get that holder of  $w_L$  is same as  $n_B$ 's holder. This is possible only when  $w_L$  is  $n_B$ 's commit-write. Since  $w_L$  is lastWrite of  $r_X$ , it occurs before  $r_X$  in  $S$ . This implies that  $w_L$  is not a commit-write of  $n_B$ . This is possible only when  $w_L$  is not in  $n_B$ 's dSet. Hence  $r_X$  is an external-read of  $n_B$ .

Thus in both the cases, we get that  $r_X$  is an external-read of  $n_B$ . From our assumptions we have that  $n_A, n_B$  are peers,  $w_A$  is a commit-write of  $n_A$ , and we are given that  $(S.ord(r_X) < S.ord(w_A))$ . These are the conditions of r-w conflict. Hence,  $S.optConf(r_X, w_A)$  is true.  $\square$

Based on this conflict definition, we define a class of schedules called as *Conflict Preserving Closed Nested Opacity* or *CP-CNO*.

**Definition 5** *A schedule  $S$  belongs to CP-CNO class if there exists a serial schedule  $SS$  such that:*

1. *Event Equivalence: The events of  $S$  and  $SS$  are the same. Formally,*  
 $\langle (S.evts = SS.evts) \rangle$
2. *schedule-partial-order Equivalence: For any two nodes  $n_Y, n_Z$  that are peers in the computation tree represented by  $S$  if  $n_Y$  occurs before  $n_Z$  in  $S$  then  $n_Y$  occurs before  $n_Z$  in  $SS$  as well. Formally,*  
 $\langle t_X : \{n_Y, n_Z\} \subseteq S.children(t_X) : (n_Y <_S n_Z) \Rightarrow (n_Y <_{SS} n_Z) \rangle$
3. *optConf Implication: if two memory operations in  $S$  are in optConf then they are also in optConf in  $SS$ . Formally,*

$$\langle \forall m_Y, \forall m_Z : \{m_Y, m_Z\} \subseteq S.evts : (S.optConf(m_Y, m_Z) \Rightarrow SS.optConf(m_Y, m_Z)) \rangle$$

We denote this equivalence to such a serial schedule as  $(S \approx_{oc} SS)$ . As we can see, the class CP-CNO is different from CNO only in condition 3. We prove this equivalence also ensures that lastWrites are the same i.e. class CP-CNO is a subset of CNO.

**Theorem 14** *If a schedule  $S$  is in the class CP-CNO then it is also in CNO. Formally,*  
 $\langle (S \in CP-CNO) \Rightarrow (S \in CNO) \rangle$

**Proof:** Since  $S \in CP-CNO$ , we know that there exists a serial schedule  $SS$  such that  $S \approx_{oc} SS$ . We will prove that the lastWrite for every read operation in  $SS$  is same as in  $S$ . We will prove this using contradiction. Consider a read  $r_X$ . Let  $(w_Y = S.lastWrite(r_X)) \neq (w_Z = SS.lastWrite(r_X))$ . Let  $S.parent(w_Y) = t_P$  and  $S.parent(w_Z) = t_Q$ . Since  $w_Y$  is the lastWrite of  $r_X$  in  $S$ , from the definition of optConf and Property 12, we get that  $S.optConf(w_Y, r_X)$  is true which also implies  $SS.optConf(w_Y, r_X)$  is true. Thus from the definition of optConf we get that  $w_Y$  occurs prior to  $r_X$  in  $SS$ . Formally,

$$\begin{aligned} \langle (w_Y = S.lastWrite(r_X)) \xrightarrow{\text{Property 12}} S.optConf(w_Y, r_X) \xrightarrow{S \approx_{oc} SS} \\ SS.optConf(w_Y, r_X) \xrightarrow{\text{optConf definition}} (SS.ord(w_Y) < SS.ord(r_X)) \rangle \quad (1) \end{aligned}$$

From the definition of lastWrite we have that

$$(w_Y = S.lastWrite(r_X)) \Rightarrow S.optVis(w_Y, r_X) \xrightarrow[\text{Lemma 6}]{S.evts=SS.evts} SS.optVis(w_Y, r_X) \quad (2)$$

$$(w_Z = SS.lastWrite(r_X)) \Rightarrow SS.optVis(w_Z, r_X) \xrightarrow[\text{Lemma 6}]{S.evts=SS.evts} S.optVis(w_Z, r_X) \quad (3)$$

Consider Eqn (1) and Eqn (2). We have that  $w_Y$  occurs prior to  $r_X$  in  $SS$  and  $SS.optVis(w_Y, r_X)$ . Further we have that  $w_Z$  is the lastWrite of  $r_X$  in  $SS$ . Combining these with Property 10 we get that  $SS.level(w_Z)$  is greater than or equal to  $SS.level(w_Y)$ . Formally,

$$\begin{aligned} \langle (SS.ord(w_Y) < SS.ord(r_X)) \wedge SS.optVis(w_Y, r_X) \wedge (w_Z = SS.lastWrite(r_X)) \xrightarrow{\text{Property 10}} \\ (SS.level(w_Z) \geq SS.level(w_Y)) \xrightarrow{SS.evts=S.evts} (S.level(w_Z) \geq S.level(w_Y)) \rangle \quad (4) \end{aligned}$$

Now we have two cases based on the positions of  $w_Z, r_X$  in  $S$ .

Case 1  $S.ord(w_Z) < S.ord(r_X)$ : Here  $w_Z$  also occurs before  $r_X$  in  $S$ . Similar to the argument of Eqn (4), combining Eqn (3) with this case we get that level of  $w_Y$  in  $S$  and  $SS$  is greater than equal to  $w_Z$ 's level,

$$\begin{aligned} & \langle (S.ord(w_Z) < S.ord(r_X)) \wedge S.optVis(w_Z, r_X) \wedge (w_Y = S.lastWrite(r_X)) \xrightarrow{\text{Property 10}} \\ & (S.level(w_Y) \geq S.level(w_Z)) \xrightarrow{SS.evts=S.evts} (SS.level(w_Y) \geq SS.level(w_Z)) \rangle \quad (5) \end{aligned}$$

Combining Eqn (4) with Eqn (5) we get that level of  $w_Y$  in  $S$  and  $SS$  is equal to  $w_Z$ 's level,

$$\begin{aligned} & (S.level(w_Z) \geq S.level(w_Y)) \wedge (S.level(w_Y) \geq S.level(w_Z)) \Rightarrow (S.level(w_Z) = S.level(w_Y)) \\ & \xrightarrow{SS.evts=S.evts} (SS.level(w_Z) = SS.level(w_Y)) \quad (6) \end{aligned}$$

This gives us that the levels are the same. Combining this result with the information of  $S$  i.e.  $w_Z$  occurs prior to  $r_X$  in  $S$ ,  $w_Z$  is optVis to  $r_X$  and  $w_Y$  is the lastWrite of  $r_X$  in  $S$  and Property 11 we get that  $w_Z$  occurs prior to  $w_Y$  in  $S$ . Formally,

$$\begin{aligned} & \langle (S.level(w_Z) = S.level(w_Y)) \wedge (S.ord(w_Z) < S.ord(r_X) \wedge S.optVis(w_Z, r_X)) \wedge \\ & (w_Y = S.lastWrite(r_X)) \xrightarrow{\text{Property 11}} (S.ord(w_Z) < S.ord(w_Y)) \rangle \quad (7) \end{aligned}$$

Similarly combining Eqn (6) with the information about  $SS$ , we get that  $w_Y$  occurs prior to  $w_Z$  in  $SS$ .

$$\begin{aligned} & \langle (SS.level(w_Z) = SS.level(w_Y)) \wedge (SS.ord(w_Y) < SS.ord(r_X) \wedge SS.optVis(w_Y, r_X)) \wedge \\ & (w_Z = SS.lastWrite(r_X)) \xrightarrow{\text{Property 11}} (SS.ord(w_Y) < SS.ord(w_Z)) \rangle \quad (8) \end{aligned}$$

From Eqn (2) we have that  $w_Y$  is optVis to  $r_X$  in  $S$  and from Eqn (3) we have that  $w_Z$  is optVis to  $r_X$  in  $S$ . In Eqn (6) we obtained that level of  $w_Z$  is same as  $w_Y$ 's level in  $S$ . Combining these results with Property 5 we get that parent of  $w_Z$  is same as  $w_Y$  in  $S$ ,

$$\begin{aligned} & \langle S.optVis(w_Y, r_X) \wedge S.optVis(w_Z, r_X) \wedge (S.level(w_Z) = S.level(w_Y)) \\ & \xrightarrow{\text{Property 5}} (S.parent(w_Y) = S.parent(w_Z)) \rangle \quad (9) \end{aligned}$$

Combining Eqn (7), which states that  $w_Z$  occurs before  $w_Y$  in  $S$ , with the result obtained just above in Eqn (9) we get that  $w_Z$  is in optConf with  $w_Y$  in  $S$ . From  $S \approx_{oc} SS$ , we get that this is also true in  $SS$ . Hence  $w_Z$  should also occur prior to  $w_Y$  in  $SS$ ,

$$\begin{aligned} & \langle (S.parent(w_Y) = S.parent(w_Z)) \wedge (S.ord(w_Z) < S.ord(w_Y)) \xrightarrow[\text{definition}]{optConf} (S.optConf(w_Z, w_Y)) \\ & \xrightarrow{S \approx_{oc} SS} (S.optConf(w_Z, w_Y)) \xrightarrow[\text{definition}]{optConf} ((SS.ord(w_Z) < SS.ord(w_Y))) \rangle \quad (10) \end{aligned}$$



But this result contradicts with Eqn (8) which states that  $w_Y$  should occur prior to  $w_Z$  in  $SS$ . Hence this case is not possible.

Case 2  $S.ord(r_X) < S.ord(w_Z)$ : In this case  $r_X$  occurs before  $w_Z$  in  $S$ .

Eqn (3) states  $w_Z$  is optVis to  $r_X$  in  $S$ . From Eqn (4) we have that level of  $w_Z$  is greater than or equal to level of  $w_Y$  which is the lastWrite of  $r_X$  in  $S$ . Combining all these with the current case we obtain that  $r_X, w_Z$  are in optConf in  $S$ . From  $S \approx_{oc} SS$ , we get that this is also true in  $SS$ . Hence  $w_Z$  should occur after  $r_X$  in  $SS$ ,

$$\begin{aligned} & (S.ord(r_X) < S.ord(w_Z)) \wedge (S.level(w_Z) \geq S.level(w_Y)) \wedge S.optVis(w_Z, r_X) \wedge \\ & (w_Y = S.lastWrite(r_X)) \xrightarrow{\text{Lemma 13}} (S.optConf(r_X, w_Z)) \xrightarrow{S \approx_{oc} SS} \\ & (SS.optConf(r_X, w_Z)) \xrightarrow[\text{definition}]{optConf} (SS.ord(r_X) < SS.ord(w_Z)) \end{aligned} \quad (11)$$

Thus  $w_Z$  cannot be lastWrite of  $r_X$  in  $SS$  which again is a contradiction. Hence this case is also not possible and rules out all cases.

This implies that  $(w_Z \neq SS.lastWrite(r_X))$ . □

Next we give an example of a schedule which is in CNO but not in CP-CNO. Consider the following computation tree and schedule:

**Example 5 Computation Tree:**

$$\begin{aligned} t_0 & : \{t_{init}, t_{01}, t_{02}, t_{03}, t_{fin}\}, \\ t_{01} & : \{sm_{011} = r_{011}(x), sm_{012} = w_{012}(y), c_{01}\}, \\ t_{02} & : \{sm_{021} = r_{021}(y), sm_{022} = w_{022}(y), c_{02}\}, \\ t_{03} & : \{sm_{031} = r_{031}(z), sm_{032} = w_{032}(y), c_{03}\} \end{aligned}$$

*Schedule:*

$$S4 : r_{011}(x)r_{021}(y)w_{012}(y)w_{01}^{012}(y)c_{01}w_{022}(y)w_{02}^{022}(y)c_{02}r_{031}(z)w_{032}(y)w_{03}^{032}(y)c_{03}$$

Figure 6 shows the computation tree corresponding to  $S4$ . An equivalent opaque serial schedule is:

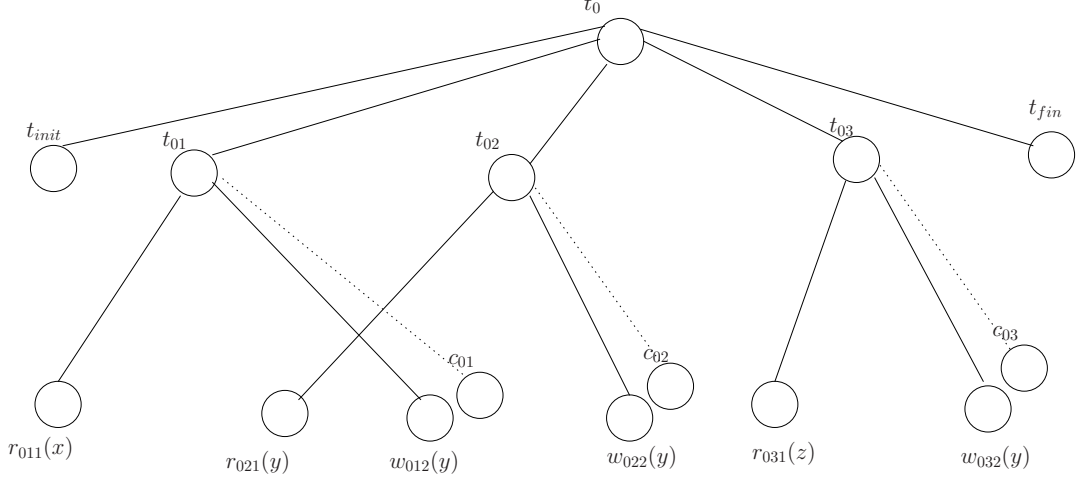
$$S5 : r_{021}(y)w_{022}(y)w_{02}^{022}(y)c_{02}r_{011}(x)w_{012}(y)w_{01}^{012}(y)c_{01}r_{031}(z)w_{032}(y)w_{03}^{032}(y)c_{03}$$

The set of optConfs in  $S4$ :

$$\{(r_{021}(y), w_{01}^{012}(y)), (r_{021}(y), w_{022}(y)), (r_{021}(y), w_{03}^{032}(y)), (w_{01}^{012}(y), w_{02}^{022}(y)), (w_{01}^{012}(y), w_{03}^{032}(y)), (w_{02}^{022}(y), w_{03}^{032}(y))\}$$

But there is no optConf equivalent serial schedule for this example. In the next section we will show this using the graph construction algorithm. This shows that  $CP-CNO \subset CNO$ .

In many of the existing STM systems proposed (for non-nested transactions), whenever a conflict is detected between a read and a write operation of two transactions, one of the transactions is aborted [9]. It can be verified that the set of schedules accepted by such a system is a subclass of CP-CNO. By defining optConf only between external-reads and commit-writes as opposed to any arbitrary read and write, the class CP-CNO is as non-restrictive as possible.



**Figure 6: The computation tree for Example 5**

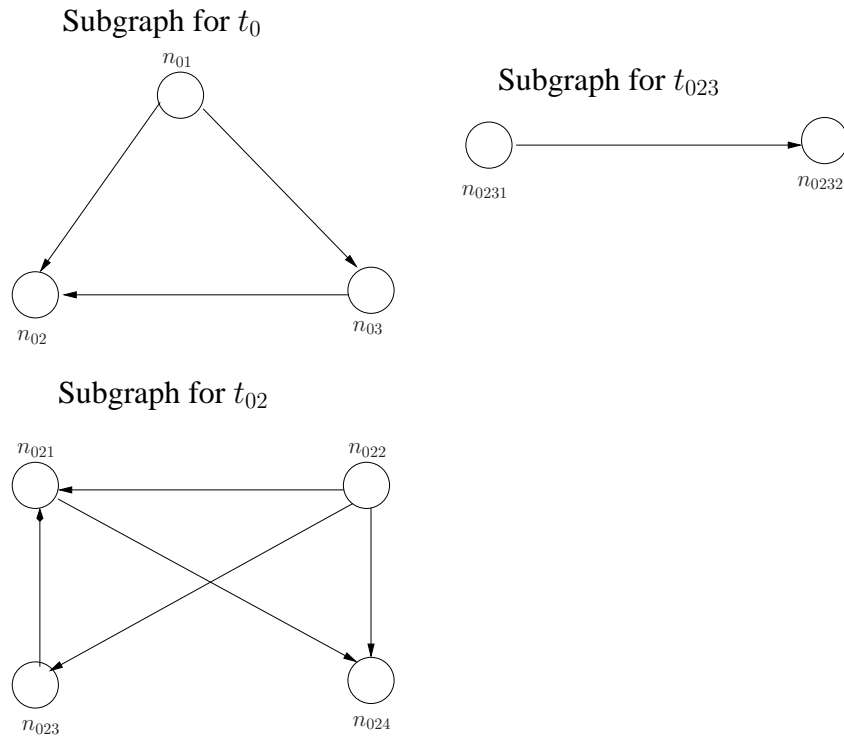
### 3.3 Algorithm

In this subsection, we describe the algorithm for testing the membership of the class CP-CNO in polynomial time. Our algorithm is based on the graph construction algorithm by Resende and Abbadi [16]. For a schedule  $S$ , the algorithm computes a *conflict graph* (also referred as *serialization graph*) based on *optConfs*, denoted as  $S.optConfGraph$ , and checks for the acyclicity of the graph constructed. We call this as *optConfGraphCons algorithm*. The graph  $S.optConfGraph$  is constructed as follows: (1) Vertices: It comprises of all the nodes in the computation tree. The vertex for a node  $n_X$  is denoted as  $v_X$ . (2) Edges: Consider each transaction  $t_X$  starting from  $t_0$ . For each pair of children  $n_P, n_Q$ , (other than  $t_{init}$  and  $t_{fin}$ ) in  $S.children(t_X)$  we add an edge from vertex  $v_P$  (corresponding to  $n_P$ ) to vertex  $v_Q$  (corresponding to  $n_Q$ ) as follows:

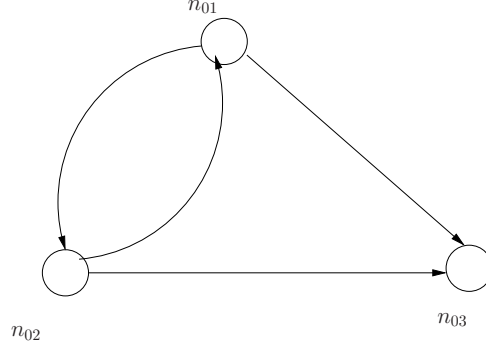
1. Completion edges: if  $n_P <_S n_Q$
2. Conflict edges: For any two memory operations,  $m_Y, m_Z$  such that  $m_Y$  is in  $n_P$ 's dSet and  $m_Z$  is in  $n_Q$ 's dSet, an edge from  $n_P$  to  $n_Q$  if  $S.optConf(m_Y, m_Z)$  is true.

Then the algorithm checks for the acyclicity of the graph  $S.optConfGraph$  constructed. Since the position of the transactions  $t_{init}$  and  $t_{fin}$  are fixed in the tree and in any schedule, we do not consider them in our graph construction algorithm. It must be noted that in our graph construction all the edges are between vertices corresponding to peer nodes. There are no edges between vertices that correspond to nodes of different levels. Thus the graph constructed consists of disjoint subgraphs. Applying this algorithm on the schedule of  $S3$  of Example 4 we get the graph shown in Figure 7. In Figure 8 we show the serialization graph for the schedule  $S4$  of Example 5. As one can see this graph has a cycle caused by the conflicts:  $(w_{01}^{012}(y), w_{02}^{022}(y))$  and  $(r_{021}(y), w_{01}^{012}(y))$ . Hence this schedule is not in CP-CNO.

We use this graph characterization to determine if a schedule  $S$  is in CP-CNO or not. We show in Theorem 30 that a schedule  $S$  is in CP-CNO if and only if the serialization graph constructed is acyclic. We prove this theorem using a series of lemmas and properties. The properties and lemmas from Property 15



**Figure 7:** The serialization graph for the schedule in Example 4. Only the subgraphs of nested transactions are shown here.



**Figure 8:** The serialization graph for the schedule in Example 5. Only the subgraph of the nested transaction  $t_0$  is show here.

until Lemma 20 show that if  $S$  is in CP-CNO then  $S.optConfGraph$  is acyclic (forward direction of the theorem).

**Proposition 15** Consider a graph  $g1$ , which is a subgraph of another graph  $g2$ . If  $g1$  is cyclic then  $g2$  is also cyclic. Formally,

$$\langle (g1 \subseteq g2) \wedge (g1 \text{ is cyclic}) \Rightarrow (g2 \text{ is cyclic}) \rangle$$

From graph theory we get the above property. Next we get the following property and lemmas from  $optConfGraphCons$  algorithm.

**Property 16** Consider a schedule  $S$ , and the corresponding graph,  $S.optConfGraph$ , constructed by  $optConfGraphCons$  algorithm. Let it contain two vertices  $v_R$  (corresponding to the tree node  $n_R$ ) and  $v_S$  (corresponding to the tree node  $n_S$ ). If there is an edge from  $v_R$  to  $v_S$  then the tree nodes  $n_R$  and  $n_S$  have the same parent.

**Lemma 17** Consider a serial schedule  $SS$ , its serialization graph  $SS.optConfGraph$  constructed using  $optConfGraphCons$  algorithm. Let it contain two vertices  $v_R$  (corresponding to the tree node  $n_R$ ) and  $v_S$  (corresponding to the tree node  $n_S$ ). If there is an edge from  $v_R$  to  $v_S$  then the last event of  $n_R$  in  $SS$  occurs before the first event of  $n_S$  in  $SS$ .

**Proof:** From the construction of  $SS.optConfGraph$  as observed in Property 16 we have that there is a transaction  $t_P$  which is the parent of  $n_R$  and  $n_S$ . Now we have two cases depending on the type of edge connecting from  $v_R$  to  $v_S$ .

- Completion edge: From the definition of completion edge, we directly get that  $SS.ord(SS.n_R.last) < SS.ord(SS.n_S.first)$ .
- Conflict edge: From the definition of conflict edge, we have that,
 
$$\langle (\exists m_X, m_Y : (m_X \in SS.dSet(n_R)) \wedge (m_Y \in SS.dSet(n_S)) \wedge (SS.parent(n_R) = SS.parent(n_S)) \wedge (SS.optConf(m_X, m_Y))) \xrightarrow[\text{definition}]{optConf} ((\exists m_X, m_Y : (m_X \in SS.dSet(n_R)) \wedge (m_Y \in SS.dSet(n_S)) \wedge (SS.parent(n_R) = SS.parent(n_S)) \wedge (SS.ord(m_X) < SS.ord(m_Y))) \rangle$$
 Applying Property 7 on this result we get that  $SS.ord(SS.n_R.last) < SS.ord(SS.n_S.first)$

□

**Lemma 18** For a serial schedule  $SS$ ,  $SS.optConfGraph$  is acyclic.

**Proof:** We will prove this using contradiction. Let  $SS.optConfGraph$  be cyclic. Let a cycle in  $SS.optConfGraph$  be composed of  $k$  vertices,  $v_{X_1} \rightarrow v_{X_2} \rightarrow \dots \rightarrow v_{X_k} \rightarrow v_{X_1}$ . From Lemma 17 we get that,  
 $(SS.ord(SS.n_{X_1}.last) < SS.ord(SS.n_{X_2}.first) < SS.ord(SS.n_{X_2}.last) < SS.ord(SS.n_{X_3}.first) < \dots < SS.ord(SS.n_{X_1}.first) < SS.ord(SS.n_{X_1}.last)) \Rightarrow (SS.ord(SS.n_{X_1}.last) < SS.ord(SS.n_{X_1}.last))$   
This is not possible. Hence  $SS.optConfGraph$  cannot be cyclic. □

**Lemma 19** Consider a schedule  $S$  and a serial schedule  $SS$  such that  $(S \approx_{oc} SS)$ . Then  $S.optConfGraph$  is a subgraph of  $SS.optConfGraph$ . Formally,  
 $\langle (S \approx_{oc} SS) \wedge (SS \text{ is serial}) \Rightarrow (S.optConfGraph \subseteq SS.optConfGraph) \rangle$

**Proof:** To prove this we have to show that, if  $(S \approx_{oc} SS)$  then  $(S.optConfGraph.v = SS.optConfGraph.v) \wedge (S.optConfGraph.e \subseteq SS.optConfGraph.e)$ .

From `optConfGraphCons` algorithm we get that every vertex in the graph corresponds to a computation tree node. Since  $(S \approx_{oc} SS)$ , the set of events and transactions of  $S$  are the same as  $SS$ . Hence we get  $(S.optConfGraph.v = SS.optConfGraph.v)$ .

Coming to the edges, any edge in  $S.optConfGraph$  corresponds to either a completion or conflict edge between peer nodes in  $S$ . From  $\approx_{oc}$  equivalence we get that these relationships also exist in  $SS$ . Hence these edges also exist in  $SS.optConfGraph$ . Thus we have  $(S.optConfGraph.e \subseteq SS.optConfGraph.e)$ . This implies  $(S.optConfGraph \subseteq SS.optConfGraph)$ . □

**Lemma 20** Let  $S$  be a schedule for which there is a serial schedule  $SS$  such that  $(S \approx_{oc} SS)$ . Then  $S.optConfGraph$  is acyclic. Formally,  
 $\langle (S \approx_{oc} SS) \wedge (SS \text{ is serial}) \Rightarrow (S.optConfGraph \text{ is acyclic}) \rangle$

**Proof:** We will prove this using contradiction. Let  $S.optConfGraph$  be cyclic. We have,

$$\begin{aligned}
& (S \approx_{oc} SS) \wedge (SS \text{ is serial}) \wedge (S.optConfGraph \text{ is cyclic}) \\
\Rightarrow & \{ \text{Lemma 19} \} \\
& (S.optConfGraph \subseteq SS.optConfGraph) \wedge (SS \text{ is serial}) \wedge (S.optConfGraph \text{ is cyclic}) \\
\Rightarrow & \{ \text{Property 15} \} \\
& (SS \text{ is serial}) \wedge (SS.optConfGraph \text{ is cyclic}) \\
\Rightarrow & \{ \text{contrapositive of Lemma 18} \} \\
& (SS \text{ is serial}) \wedge (SS \text{ is not serial})
\end{aligned}$$

Here we have a contradiction. Hence  $S.optConfGraph$  is acyclic.  $\square$

Next we show that for a given schedule  $S$ , if its serialization graph is acyclic then  $S$  is in CP-CNO (the reverse direction of Theorem 30). This is shown in properties, lemmas Lemma 21 to Lemma 29.

We give an algorithm for generating conflict preserving serial schedule from  $S.optConfGraph$  if it is acyclic. We call this *expander algorithm*. The expander algorithm separates the disjoint sub-graphs of  $S.optConfGraph$ . For a transaction  $t_X$ , a subgraph denoted as  $g_X$  is constructed by taking all the nodes corresponding to  $t_X$ 's children nodes and the edges between them. To construct the final schedule the expander algorithm works with  $xschedules$ . A  $xschedule$  is like a normal schedule but also has transaction operations in its event set. Similar to a schedule all the events in a  $xschedule$  are totally ordered. When a  $xschedule$  has no transaction operations in it, then it is same as a normal schedule. For a transaction  $t_X$ , a subgraph denoted as  $g_X = t_X.subGraph(S)$  is constructed as follows:

Initialize a  $xschedule$   $XS$  to  $t_0$

- 1 Parse the  $xschedule$   $XS$ . Perform the following actions when each of the following is encountered:
  - 1.1 Transaction  $t_N$ : Replace this transaction with all its child operations, followed by  $t_N$ 's commit-write set and  $t_N$ 's terminal operation. The order of  $t_N$ 's children is given by a topological sort obtained from the graph  $g_N = t_N.subGraph(S)$ .
  - 1.2 Memory and Terminal operations: Nothing needs to be done.
- 2 Repeat the above step until the serial schedule  $XS$  contains only memory and terminal operations.

When the expander algorithm starts, the  $xschedule$   $XS$  has only one transaction  $t_0$  in it. Then expander algorithm recursively replaces any transaction operation in  $XS$  with its children, its commit-write operations and its terminal operation until  $XS$  has no more transactions in it. We denote the various changes in the  $xschedule$  by subscripting  $XS$ . The expander algorithm starts with  $XS_0$ , working through  $XS_1$ ,  $XS_2$  and so on until it reaches the final schedule  $XS_f$ . We denote the final schedule  $XS_f$  as  $RS$  (resultant schedule).

The topological sorts of the various subgraphs obtained by applying this algorithm on  $S3.optConfGraph$  of Example 4:

$g_0 : t_{01}t_{03}t_{02}$   
 $g_{01} : t_{011}t_{012}$   
 $g_{02} : t_{022}t_{023}t_{021}t_{024}$   
 $g_{021} : t_{0211}t_{0212}t_{0213}$   
 $g_{023} : t_{0231}t_{0232}$   
 $g_{0231} : t_{02311}t_{02312}$   
 $g_{0232} : t_{02321}t_{02322}t_{02323}$   
 $g_{024} : t_{0241}t_{0242}t_{0243}$   
 $g_{03} : t_{031}t_{032}t_{033}$

The resultant schedule is:

$S6 : r_{011}(x)w_{012}(y)w_{01}^{012}(y)c_{01}r_{031}(y)r_{032}(z)w_{033}(d)w_{03}^{033}(d)c_{03}w_{022}(x)r_{02311}(x)w_{02312}(y)w_{0231}^{02312}(y)c_{0231}$   
 $r_{02321}(y)w_{02322}(x)w_{02323}(y)w_{0232}^{02322}(x)w_{0232}^{02323}(y)c_{0232}a_{023}r_{0211}(z)w_{0212}(x)w_{0213}(y)w_{021}^{0212}(x)w_{021}^{0213}(y)c_{021}$   
 $r_{0241}(x)r_{0242}(y)w_{0243}(z)w_{024}^{0243}(z)w_{02}^{021}(x)w_{02}^{021}(y)w_{02}^{024}(z)c_{02}$

Now we will prove if  $S.optConfGraph$  is acyclic then the resultant schedule  $RS$  obtained is serial and  $optConf$  equivalent to the original schedule  $S$ .



**Lemma 21** Consider a  $XS_i$  that has two nodes  $n_P, n_Q$  such that  $n_P$  occurs before  $n_Q$ . Then in  $XS_f(RS)$ , the last event of  $n_P$ ,  $XS_f.n_P.last$  occurs before the first event of  $n_Q$ ,  $XS_f.n_Q.first$ . Formally,  $\langle (\{n_P, n_Q\} \subseteq XS_i.evts : XS_i.ord(n_P) < XS_i.ord(n_Q)) \Rightarrow (XS_f.ord(XS_f.n_P.last) < XS_f.ord(XS_f.n_Q.first)) \rangle$

**Proof:** This is can be easily proved using induction on the distance between  $n_P, n_Q$  in  $XS_i$ :  $\delta = |XS_i.ord(n_P) - XS_i.ord(n_Q)|$ .  $\square$

One can see the following property about  $RS$ .

**Property 22** Consider a schedule  $S$  such that  $S.optConfGraph$  is acyclic. Then the resultant schedule  $RS$  satisfies validity of transaction limits i.e. after a transaction terminates no operation (memory or terminal) belonging to it should execute

In the next lemma we describe the relationship between edges in a graph of  $S.optConfGraph$  and the resultant schedule  $RS$ .

**Lemma 23** Consider a schedule  $S$  with the graph  $S.optConfGraph$  being acyclic. Let there be two vertices in  $v_P, v_Q$  in it corresponding to tree nodes  $n_P, n_Q$ . If there is an edge from  $v_P$  to  $v_Q$  then in  $RS$  the last event of  $n_P$  occurs before the first event of  $n_Q$ . Formally,  $\langle v_P, v_Q \subseteq S.optConfGraph.v, e_i \in S.optConfGraph.e : (e_i \text{ connects } v_P \text{ to } v_Q) \Rightarrow (RS.ord(RS.n_P.last) < RS.ord(RS.n_Q.first)) \rangle$

**Proof:** From our construction of  $S.optConfGraph$ , we get that  $n_P, n_Q$  are peers. Let these nodes be children of a transaction  $t_N$  in the computation tree. Let the subgraph corresponding to  $t_N$  be  $g_N = t_N.subGraph(S)$ . When the expander algorithm encounters  $t_N$  in some xschedule  $XS_j$  and parses, it replaces  $t_N$  by all its children, followed by  $t_N$ 's commit-write set and  $t_N$ 's terminal operation. The ordering among the child nodes is given by topological sort of  $g_N$ .

Since there is an edge from  $v_P$  to  $v_Q$  in  $S.optConfGraph$ , the expander algorithm ensures that  $v_P$  occurs before  $v_Q$  in the topological sort of  $g_N$ . Hence in  $XS_{(j+1)}$ , the expander algorithm places  $n_P$  before  $n_Q$ . Combining this result with Lemma 21, we get that in  $RS$  the last event of  $n_P$  occurs before the first event of  $n_Q$ .  $\square$

Next we show that  $RS$  satisfies each of the conditions mentioned in the definition of  $CP-CNO$ .

**Property 24** If  $S.optConfGraph$  is acyclic then  $RS$  contains the same events as  $S$ . Formally,  $\langle (S.optConfGraph \text{ is acyclic}) \Rightarrow (S.evts = RS.evts) \rangle$

This property directly follows from the observation that the expander algorithm does not alter the computation tree. It only alters the schedule of the memory operations.

**Property 25** If  $S.optConfGraph$  is acyclic then  $RS$  is serial.

This property follows directly from the working of expander algorithm.

**Lemma 26** Consider a schedule  $S$  such that  $S.optConfGraph$  is acyclic. Let  $t_X$  be a transaction in  $S$  with children  $n_P$  and  $n_Q$ . If  $n_P$  occurs before  $n_Q$  in  $S$  then  $n_P$  also occurs before  $n_Q$  in  $RS$ . Formally,  $\langle S : t_X \in S.nodes, \{n_P, n_Q\} \subseteq S.children(t_X) : ((S.optConfGraph \text{ is acyclic}) \wedge (n_P <_S n_Q)) \Rightarrow (n_P <_{RS} n_Q) \rangle$

**Proof:** From the construction of  $g = S.optConfGraph$  we can see that it contains two vertices  $v_P$  (corresponding to  $n_P$ ) and  $v_Q$  (corresponding to  $n_Q$ ). If  $(n_P <_S n_Q)$  then in  $g$  there is an edge from  $v_P$  (the vertex corresponding to  $n_P$ ) to  $v_Q$  (the vertex corresponding to  $n_Q$ ). Combining this with Lemma 23 we get that  $(RS.ord(RS.n_P.last) < RS.ord(RS.n_Q.first))$  which implies that  $(n_P <_{RS} n_Q)$ .  $\square$

**Lemma 27** Consider a schedule  $S$  with two memory operations  $m_X, m_Y$  such that  $S.optConf(m_X, m_Y)$  is true. If  $S.optConfGraph$  is acyclic then in  $RS$ ,  $m_X$  occurs before  $m_Y$ . Formally,  
 $(S.optConfGraph \text{ is acyclic}) \wedge S.optConf(m_X, m_Y) \Rightarrow (RS.ord(m_X) < RS.ord(m_Y))$

**Proof:** From the definition of  $optConf$ , we get that there exist two peer nodes  $n_P, n_Q$  such that  $m_X$  is in  $n_P$ 's  $dSet$  and  $m_Y$  is in  $n_Q$ 's  $dSet$ . From the construction of  $g = S.optConfGraph$  we can see that it contains two vertices  $v_P$  (corresponding to  $n_P$ ) and  $v_Q$  (corresponding to  $n_Q$ ) and there is an edge between  $v_P$  and  $v_Q$ . Now the argument is similar to the proof of Lemma 26. Due to the presence of an edge, from Lemma 23 we get that  $(RS.ord(RS.n_P.last) < RS.ord(RS.n_Q.first))$ . Hence,  $m_X$  occurs before  $m_Y$  in  $RS$ .  $\square$

**Lemma 28** Consider a schedule  $S$  such that  $S.optConfGraph$  is acyclic. Then the lastWrite for every read operation in  $S$  is the same as in  $RS$ . Formally,  
 $\langle S.optConfGraph \Rightarrow (\forall r_X \in S.evts : (S.lastWrite(r_X) = RS.lastWrite(r_X))) \rangle$

**Proof:** The proof is very similar to Theorem 14.  $\square$

**Lemma 29** Consider a schedule  $S$  with two memory operations  $m_X, m_Y$  such that  $S.optConf(m_X, m_Y)$  is true. If  $S.optConfGraph$  is acyclic then  $RS.optConf(m_X, m_Y)$  is true as well. Formally,  
 $\langle (S.optConfGraph \text{ is acyclic}) \wedge S.optConf(m_X, m_Y) \Rightarrow RS.optConf(m_X, m_Y) \rangle$

**Proof:** From Property 24, we get that all the events in  $RS$  is same as  $S$ . Thus their computation trees are the same. Further from Lemma 28, we get that all the lastWrites for every read are same in  $S$  and  $RS$ . Let us consider each case of conflict:

- $m_X = w_X, m_Y = r_Y$ : This case implies that there exist two peer nodes  $n_P, n_Q$  such that  $w_X$  is  $n_P$ 's commit-write and  $r_Y$  is  $n_Q$ 's external-read in  $S$ . Since the computation trees of  $S$  and  $RS$  are the same and the lastWrites for every read are the same we have that  $w_X$  is  $n_P$ 's commit-write and  $r_Y$  is  $n_Q$ 's external-read in  $RS$  as well. From Lemma 27, we get that  $w_X$  occurs before  $r_Y$  in  $RS$ . These are the conditions for  $w_X$  and  $r_Y$  to be in  $optConf$  in  $RS$ . Hence,  $w_X, r_Y$  are in  $optConf$  in  $RS$  as well.
- $m_X = r_X, m_Y = w_Y$ : The argument is the same as above.
- $m_X = w_X, m_Y = w_Y$ : Here,  $w_X$  and  $w_Y$  are peers in  $S$ . Since the computation trees of  $S$  and  $RS$  are the same,  $w_X$  and  $w_Y$  are peers in  $RS$  as well. From Lemma 27, we get that  $w_X$  occurs before  $w_Y$  in  $RS$ . Hence,  $w_X, w_Y$  are in  $optConf$  in  $RS$  as well.

Thus, in all the cases we get that  $RS.optConf(m_X, m_Y)$  is true.  $\square$

Finally we have,

**Theorem 30** A schedule  $S$  is in CP-CNO iff if the conflict graph  $S.optConfGraph$  is acyclic. Formally,  $(S \in CP-CNO) \Leftrightarrow (S.optConfGraph \text{ is acyclic})$

**Proof:** We will prove each direction.

$(\Rightarrow) (S \in CP-CNO) \Rightarrow (S.optConfGraph \text{ is acyclic}):$

Here we have that,

$(S \in CP-CNO) \xrightarrow[\text{definition}]{CP-CNO} ((S \approx_{oc} SS) \wedge (SS \text{ is serial})) \xrightarrow{\text{Lemma 20}} (S.optConfGraph \text{ is acyclic})$

$(\Leftarrow) (S.optConfGraph \text{ is acyclic}) \Rightarrow (S \in CP-CNO):$

Since  $S.optConfGraph$  is acyclic, the expander algorithm generates a schedule  $RS$ . From Property 25 we get that  $RS$  is serial. Next we will prove each of the conditions required by the definition of CP-CNO.

- Event Equivalence: From Property 24 we get that,  $((S.evts = RS.evts) \wedge (S.nodes = SS.nodes))$ .
- schedule-partial-order Equivalence: From Lemma 26, we get that,  $\langle S : t_X \in S.nodes, \{n_P, n_Q\} \subseteq S.children(t_X) : (n_P <_S n_Q) \Rightarrow (n_P <_{RS} n_Q) \rangle$
- optConf Implication: From Lemma 29, we get that,  $\langle S : (\{m_X, m_Y\} \subseteq S.evts) : (S.optConf(m_X, m_Y)) \wedge (S.optConfGraph \text{ is acyclic}) \Rightarrow (RS.optConf(m_X, m_Y)) \rangle$ .

This proves all the requirements for CP-CNO.

□

This proof shows that checking the membership of CP-CNO can be done in polynomial time since checking for acyclicity can be done in polynomial time.

## 4 Extensions to Closed Nested Opacity

In the previous section we developed a polynomial time verifiable characterization of CP-CNO, a subclass of CNO (Theorem 30). In this section we will develop some extensions to CNO.

### 4.1 Drawback of CNO

Given a schedule with aborted transactions, opacity specifies that the read operations of aborted transactions also read consistent values. To ensure that no transaction reads from an aborted transaction, aborted transactions are treated as read-only transactions. A given schedule is said to be opaque if there exists a serial schedule equivalent to it. In this way the current specification of opacity ensures that the reads of all transactions (including aborted transactions) are consistent and the writes of aborted transactions are hidden from other transactions. Class CNO extends opacity to nested transactions, treating aborted sub-transactions in the same manner.

Consider the following transactions,

<b>Transaction 1</b> $t_{01}$	<b>Transaction 3</b> $t_{022}$
1: read $y$	1: read $y$
2: write $y$	2: read $z$
3: write $z$	3: write $d$
<b>Transaction 2</b> $t_{02}$	<b>Transaction 4</b> $t_{023}$
1: read $d$	1: read $p$
2: invoke $t_{022}$	2: read $z$
3: invoke $t_{023}$	3: write $z$

For this example, let the transactions  $t_{01}, t_{02}$  execute in an interleaved manner. Let the following partial schedule represent the execution of these transactions. In this partial schedule, transactions do not have terminal operations. A STM scheduler on receiving these events in this order will decide to abort or commit the transactions.

**Example 6**  $t_0 : \{t_{init}, t_{01}, t_{02}, t_{fin}\}$ ,  
 $t_{01} : \{sm_{011} = r_{011}(y), sm_{012} = w_{012}(y), sm_{013} = w_{013}(z)\}$ ,  
 $t_{02} : \{sm_{021} = r_{021}(d), t_{022}, t_{023}\}$ ,  
 $t_{022} : \{sm_{0221} = r_{0221}(y), sm_{0222} = r_{0222}(z), sm_{0223} = w_{0223}(d)\}$ ,  
 $t_{023} : \{sm_{0231} = r_{0231}(p), sm_{0232} = r_{0232}(z), sm_{0233} = w_{0233}(z)\}$ ,

*Schedule:*

$S7 : r_{011}(y)r_{021}(d)w_{012}(y)r_{0221}(y)w_{013}(z)r_{0222}(z)w_{0223}(d)r_{0231}(p)r_{0232}(z)w_{0233}(z)$

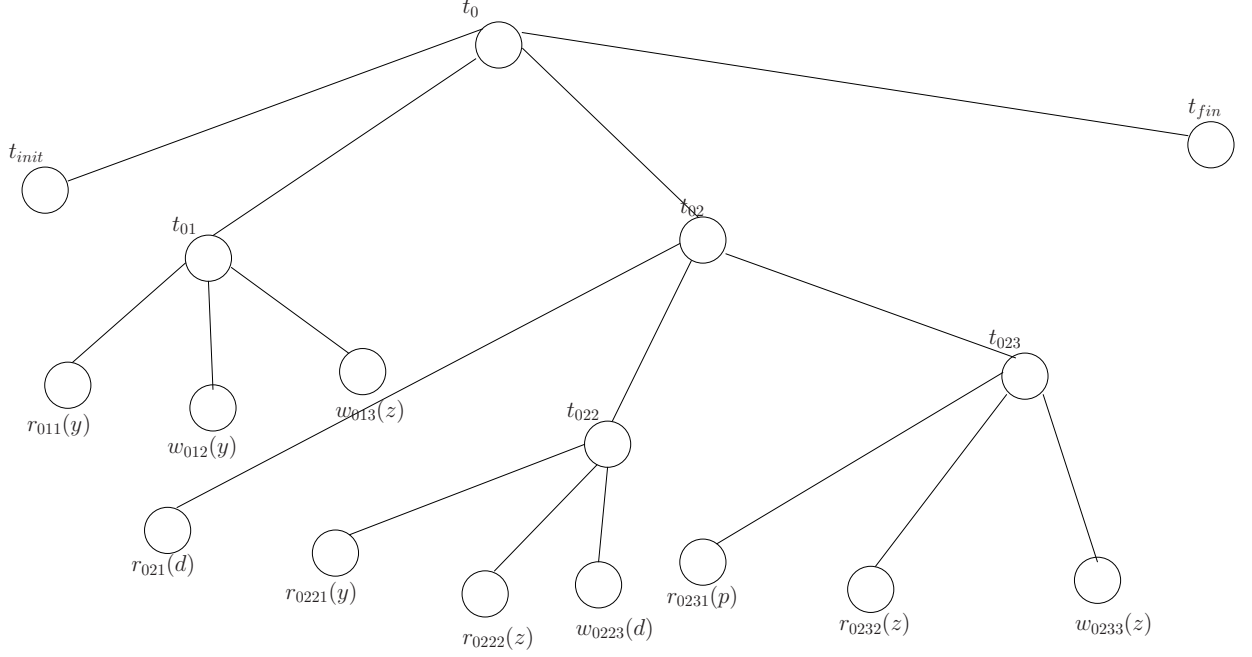
Consider a scheduler  $H$  based on the class CP-CNO (and hence the class CNO) which schedules these events. The scheduler is invoked on-demand basis. When a transaction wishes to perform a read operation or wishes to commit, it invokes the scheduler. On being invoked, the scheduler looks at the current operation (either read or commit operation) with the history of events already executed. Using all these events it constructs a serialization graph based on optConf and checks for acyclicity. If the graph is acyclic, then the scheduler allows the current operation to execute. Otherwise it does not allow the operation to execute and aborts the corresponding transaction.

In the given schedule, the scheduler allows all the events till the transaction  $t_{01}$  commits. None of these events form a conflict cycle. Thus the schedule of the events are:

$r_{011}(y)r_{021}(d)w_{012}(y)r_{0221}(y)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}$

Then, let the next event to be executed be  $r_{0222}(z)$  belonging to  $t_{022}$ . In between the reads of the variables  $y$  and  $z$  by the transaction  $t_{022}$ , the transaction  $t_{01}$  updates these variables. Thus a conflict cycle is formed between the transactions  $t_{01}$  and  $t_{02}$  in the serialization graph and hence this schedule is not in CP-CNO. As a result the scheduler will abort the transaction  $t_{022}$ . Further it can be verified that this schedule is not in the class CNO as well.

Next the events of the transaction  $t_{023}$  execute as shown in the schedule. The read operation  $r_{0231}(p)$  is allowed by the scheduler. The next event to execute is a read operation  $r_{0232}(z)$ . But the scheduler  $H$  will not allow this event to execute as it causes a conflict cycle. It can be seen that the read operation  $r_{0221}(y)$  by the transaction  $t_{022}$  has been performed before the transaction  $t_{01}$  commits. But the read  $r_{0232}(z)$  is performed after  $t_{01}$  commits and the lastWrite of  $r_{0232}(z)$  is  $w_{01}^{012}(z)$ . Due to these operations, a cycle is



**Figure 9: The tree for Example 6**

formed in the serialization graph between the nodes  $t_{01}$  and  $t_{02}$ . Even though the transaction  $t_{022}$  has been aborted, its read operation still causes  $t_{023}$  to abort. Thus, this schedule of events  $S7$  is not in CP-CNO. This schedule is not in CNO as well. For this schedule to be accepted by the scheduler, no sub-transaction of  $t_{02}$  starting after  $t_{022}$  has aborted can read any of the data-items written by  $t_{01}$ . In the worst case all the sub-transactions of  $t_{02}$  starting after  $t_{022}$  could abort due to  $r_{0221}(y)$  which effectively implies that  $t_{02}$  is aborted as well. This shows that the read of an aborted sub-transaction can cause its top-level transaction to abort.

Thus with CNO, an aborted sub-transaction can severely restrict the concurrency of nested transactions. An aborted transaction affects the transactions that follow it. But ideally we would want an aborted transaction to have no affect on the transactions that follow it. To address this shortcoming, we formulate a new correctness criterion called *Abort-Shielded Consistency* or ASC. This criterion is based on the notion of sub-schedules. Next, we will describe a few notations which we will later use to describe the correctness criterion.

## 4.2 Notations

For a transaction  $t_X$  in  $S$  we denote the terminal operations of all the sub-transactions in  $t_X$ 's dSet by terminal operation or  $S.termOp(t_X)$ . We denote  $S.schOps(t_X)$  as the set of operations in  $S.dSet(t_X)$  which are also present in  $S.evts$  along with the set of terminal operations. Formally,  $S.schOps(t_X) = (S.dSet(t_X) \cap S.evts) \cup S.termOp(t_X)$ .

We define two functions for a commit-write operation. If  $w_X$  is a commit-write operation in  $S$ , then  $S.orgWrite(w_X(d))$  denotes the original simple-write of  $w_X(d)$ . Let the holder of the commit-write  $w_X$  be  $n_X$ . Then function  $S.baseWrite(w_X(d))$  denotes the corresponding commit-write or simple-write on  $d$  in

the child transaction of  $n_X$ . For example in  $S7$  of Example 6, for the commit-write  $w_{02}^{022}(d)$ , the baseWrite is  $w_{022}^{0223}(d)$  and the orgWrite is  $w_{0223}(d)$ . For the commit-write  $w_{022}^{0223}(d)$ , the baseWrite and orgWrite are  $w_{0223}(d)$ . Thus the orgWrite is always a simple-write whereas the baseWrite can be either a commit-write or a simple-write.

We define a few notations based on aborted transactions in a schedule. Consider a schedule  $S$ , with a transaction  $t_X$ . We denote  $S.abort(t_X)$  as the set of all aborted transactions in  $t_X$ 's dSet. If  $t_X$  is an aborted transaction then  $S.abort(t_X)$  contains  $t_X$  as well. For  $t_X$ , we define  $S.prune(t_X)$  as all the events in the schOps of  $t_X$  after removing the events from all the aborted transactions in  $t_X$ 's dSet. Formally, 
$$S.prune(t_X) = \{S.schOps(t_X) - (\bigcup_{t_A \in S.abort(t_X)} S.schOps(t_A))\}$$

Intuitively this function denotes the schOps remaining in  $t_X$  after pruning all the aborted transactions from it. If  $t_X$  has no aborted transaction in its dSet then  $S.prune(t_X)$  is same as  $S.schOps(t_X)$ . If  $t_X$  is an aborted transaction then  $S.prune(t_X)$  is nil. Also for a schedule,  $S.prune(t_0)$  denotes the schedule events with only the committed transactions and no aborted transaction.

To capture all the pruned descendants of an aborted transaction we define chrnPruned (children-pruned) function. For a transaction  $t_X$  (either committed or aborted),

$$S.chrnPruned(t_X) = \{ \bigcup_{t_Y \in S.children(t_X)} S.prune(t_Y) \cup S.cwrite(t_Y) \}$$

It must be noted that for a committed transaction  $t_X$ ,  $S.prune(t_X)$  is same as  $S.chrnPruned(t_X)$ .

For a node  $n_P$ , its *anscTermSet* denoted as  $S.anscTermSet(n_P)$  is the set of terminal operations of all its ancestors in the schedule. We denote a node as a *committed node* if it is either a committed transaction or a simple-memory operation.

### 4.3 Sub-Schedules

Now we will formally define the notion of sub-schedules. Given a well-formed schedule  $S$  a sub-schedule  $subS$  should satisfy:

- $subS.evts \subseteq S.evts$
- $subS.ord \subseteq S.ord$

It was observed earlier that the events of schedule  $S$  constitute a computation tree with the events being the leaves. The same property applies to  $subS$  as well.

Consider a sub-schedule  $subS$  of a schedule  $S$ . Since the events in  $subS$  could be a random subset of events of a  $S$ , it may not signify anything. For  $subS$  to be meaningful it must be well-formed. The conditions of well-formedness defined in Section 2.4 for schedules also apply to sub-schedules. The set of events of the sub-schedule is a subset of the events in  $S$ , a well-formed schedule. Since the order of events in the sub-schedule  $subS$  is same as the order of the events in the  $S$ , after a transaction terminates no operation belonging to it executes. This is the condition (1), validity of transaction limits, of the well-formedness requirement.

A read operation in a sub-schedule is valid if it reads its lastWrite value of  $S$  which is condition (2) of well-formedness. Thus for any read operation in a sub-schedule, its lastWrite in  $S$  should also be in  $subS$ . In addition to this, for any memory operation  $m_X$  in  $subS$ , all the memory operations that *affect*  $m_X$  in  $S$  should also be in  $subS$ . This requirement is called *causality* of events. We say that  $subS$  is *causally complete* w.r.t  $m_X$  if it contains all the events that affect  $m_X$  in  $S$ . Now we define a few functions to formally define the affects relationship. First, we define a function *isUseful* between two memory operations. This function



defines when one memory operation is useful to another memory operation. It is similar to *immediately-useful-to* relation of [18]. For two memory operations  $m_Y, m_X$  in  $S$ , it is denoted as  $S.isUseful(m_Y, m_X)$ :

1.  $m_Y = w_Y, m_X = r_X$ :  $w_Y$  is the lastWrite of  $r_X$  then  $S.isUseful(w_Y, r_X)$  is true
2.  $m_Y = r_Y, m_X = w_X$ , where  $w_X$  is a simple-write: If there exists a node  $n_P$  such that  $n_P$  is optVis to  $w_X$ ,  $n_Q$  is a peer of  $n_P$  with  $w_X$  is in  $n_Q$ 's dSet,  $n_P$  occurred before  $n_Q$  in  $S$ ,  $r_Y$  is in the pruned set of  $n_P$  and  $r_Y$  is an external-read of  $n_P$  then  $S.isUseful(r_Y, w_X)$  is true. Formally,
 
$$\langle \exists n_P, \exists n_Q : (n_P, n_Q \text{ are peers}) \wedge (n_P <_S n_Q) \wedge (r_Y \in S.prune(n_P)) \wedge (S.lastWrite(r_Y) \notin S.dSet(n_P)) \wedge (w_X \in S.dSet(n_Q)) \Rightarrow S.isUseful(r_Y, w_X) \rangle$$
3.  $m_Y = r_Y, m_X = w_X$ , where  $w_X$  is a commit-write: Let  $S.orgWrite(w_X)$  be  $w_Z$ , the corresponding simple write. Then  $S.isUseful(r_Y, w_X)$  is true when  $S.isUseful(r_Y, w_Z)$  is true

A read operation's lastWrite affects the read operation. Hence it is useful to the read operation. Consider a simple-write operation,  $w_X$ , and a read operation  $r_Y$ . If the read is its peer and occurs before it in the schedule then  $r_Y$  affects  $w_X$ . Consider another scenario. Let  $n_P$  and  $n_Q$  be two peer nodes such that  $n_P$  occurs before  $n_Q$  in  $S$ . Let  $w_X$  be in  $n_Q$ 's dSet. Hence  $n_P$  is optVis to  $w_X$ . If  $r_Y$  is in the pruned set of  $n_P$  and is an external-read of  $n_P$  then it affects  $w_X$ . The same idea can be extended to  $w_X$  if it is a commit-write. Hence  $r_Y$  is useful to  $w_X$ . Thus, from the definition of isUseful we get that if  $m_Y$  is useful to  $m_X$ , then  $m_Y$  occurs before  $m_X$  in  $S$ .

In the schedule  $S7$ ,  $S7.isUseful(w_{01}^{013}(z), r_{0232}(z))$  is true, since  $w_{01}^{013}(z)$  is the lastWrite of  $r_{0232}(z)$ . Then  $S7.isUseful(r_{011}(y), w_{013}(z))$  and  $S7.isUseful(r_{011}(y), w_{01}^{013}(z))$  are true since  $r_{011} <_{S7} w_{013}$ ,  $w_{013}$  is the simple-write for  $w_{01}^{013}$  and  $r_{011}(d)$  being a simple-memory operation is an external-read of itself.

Based on isUseful function, for a given memory operation  $m_X$  in  $S$ , we define the set *usefulMemOps* which consists of all memory operations that are useful to  $m_X$ ,
$$S.usefulMemOps(m_X) = \{m_Y | S.isUseful(m_Y, m_X)\}$$

Next based on the notion of usefulMemOps, we identify a set of nodes that are useful to a memory operation  $m_X$  in  $S$ . It consists of all the nodes that are optVis to  $m_X$  and have at least one memory operation in their pruned sets which is useful to  $m_X$ . We call this set as *usefulNodes*. Formally,
$$S.usefulNodes(m_X) = \{n_Y | S.optVis(n_Y, m_X) \wedge (S.prune(n_Y) \cap S.usefulMemOps(m_X) \neq nil)\}$$
It can be verified that any node in the usefulNodes set of a memory operation  $m_X$  terminates before  $m_X$  in the schedule.

Next we define a function *transUsefulNodes* that computes all nodes that are directly and transitively useful to a memory operation  $m_X$ . This is recursively defined and uses usefulNodes as the base case.

$$S.transUsefulNodes(m_X) = (S.usefulNodes(m_X)) \cup \left( \bigcup_{n_Y \in S.usefulNodes(m_X) \wedge m_Y \in S.prune(n_Y)} S.transUsefulNodes(m_Y) \right)$$

Thus any node that is useful to a memory operation is also transitively useful to it. It must also be noted that if a transaction is aborted, then it cannot be useful or transitively useful to any memory operation. Thus we have the lemma,

**Lemma 31** *If a node  $n_Z$  is useful to some memory operation  $m_X$  in  $S$ , then the node  $n_Z$  is a committed node. Formally,*

$$\langle n_Z \in S.transUsefulNodes(m_X) \Rightarrow n_Z \text{ is a committed node} \rangle$$



**Proof:** This can be proved using induction over the schDist of the last event of  $n_Z$  from  $m_X$ . The base case of the induction is when  $n_Z$  is in the usefulNodes set of  $m_X$ .  $\square$

The notion committed node is defined in Section 4.2. Similar to usefulNodes, it can be proved that all the nodes in the set transUsefulNodes terminate before  $n_X$  in  $S$ .

Using the set transUsefulNodes we will construct the set usefulSchEvs for a memory operation  $m_X$ . It consists of all the pruned operations from all the nodes  $m_Y$  that are transitively useful to  $n_X$ . Formally,  $S.usefulSchEvs(m_X) = \{(\bigcup_{n_Y \in S.transUsefulNodes(m_X)} S.prune(n_Y))\}$

Using usefulSchEvs we can formally define affects relationship. A memory operation  $m_Y$  affects another memory operation  $m_X$  if  $m_Y$  is in the usefulSchEvs set of  $m_X$ . Formally,

$$S.affects(m_Y, m_X) = \begin{cases} true & (m_Y \in S.usefulSchEvs(m_X)) \\ false & otherwise \end{cases}$$

Having formally defined the affects function, we state the requirements for the well-formedness of any sub-schedule:

1. Causality Completeness: For any memory operation  $m_X$  present in a sub-schedule  $subS$  of a schedule  $S$ , the sub-schedule should also contain all the memory operations that affect  $m_X$ . Formally,  $\langle m_X \in subS.evts \Rightarrow S.usefulSchEvs(m_X) \in subS.evts \rangle$

Consider a sub-schedule  $subS$  of a schedule  $S$  that is causally complete. With this definition of causality completeness, we get that if a commit-write operation  $w_X$  is in  $subS$  then the baseWrite of  $w_X$ ,  $S.baseWrite(w_X)$  is also in  $subS$ . If  $w_X$ 's baseWrite is another commit-write then its baseWrite is also in  $subS$ . Following the baseWrites recursively which terminates in  $w_X$ 's orgWrite, we get that it is also included in  $subS$ .

Having described the usefulSchEvs w.r.t a memory operation, we next extend this notion to transactions as well (and nodes). Consider a node  $n_X$  in a schedule. For this node we define usefulSchEvs as the union of usefulSchEvs of all the memory operations in the pruned set of  $n_X$ . Formally,  $S.usefulSchEvs(n_X) = \{(\bigcup_{m_Y \in S.chrnPruned(n_X)} S.usefulSchEvs(m_Y))\}$

In addition to causality, we also require that for every transaction present in a sub-schedule, its terminal operation is also present in it. This clearly indicates when a transaction completes.

2. Transaction termination: Consider a sub-schedule  $subS$  of a schedule  $S$ . If the  $subS$  contains events from a transaction  $t_X$  then it also contains the terminal operation of  $t_X$  in its set of events. Formally,  $\langle t_X \in subS.evts \Rightarrow S.termOp(t_X) \in subS.evts \rangle$

It must be noted that by this characterization a transaction in a well-formed sub-schedule can have its commit operation but none of its commit-write operations in the sub-schedule. The sub-schedule still satisfies all the requirements of well-formedness. From causality completeness, we get the following lemma on sub-schedules.

**Lemma 32** Consider a schedule  $S$  in CNO. Let  $subS$  be a sub-schedule of  $S$  that is causally complete. Then, there exists a serial sub-schedule  $ssubS$  that:

1. Sub-Schedule Event Equivalence: The events of  $subS$  and  $ssubS$  are the same.

2. *Schedule-Partial-Order Equivalence*: For any two nodes  $n_Y, n_Z$  that are peers in the computation tree represented by  $subS$  if  $n_Y$  occurs before  $n_Z$  in  $subS$  then  $n_Y$  occurs before  $n_Z$  in  $ssubS$  as well.
3. *LastWrite Equivalence*: For all read operations the lastWrites in  $subS$  and  $ssubS$  are the same.

**Proof:** These properties follow directly from the definition of CNO. Since  $S$  is in CNO, we get that there exists a serial schedule  $SS$  which has the same set of events as  $S$ . Removing all the events from the serial schedule that are not in  $subS$ , we get that the resulting sub-schedule, denoted as  $subSS$ , has the same set of events as  $subS$  and is serial. Further it can be seen that schedule-partial-order in  $subS$  is same as  $subSS$ . This proves the conditions 1 and 2 above.

It must be noted that since  $S$  is in CNO, the lastWrites for every read in  $S$  and  $SS$  are the same. Since  $subS$  is causally complete the lastWrite for every read operation of  $subS$  is also in  $subS$ . Similarly the lastWrite for every read operation of  $subSS$  is also in  $subSS$ . Further from the construction of  $subSS$ , we get that the lastWrite of every read in  $subSS$  is same as in  $subS$ . This proves the condition 3 above. Hence, the lemma follows.  $\square$

#### 4.4 Abort Shielded Consistency

In Section 4.1 we observed how an aborted transaction can affect the transactions following it. But ideally we want an aborted transaction to have no effect on the transactions that follow it. By looking for a single serial schedule involving all transactions, opacity limits concurrency. In this section, we present a class of schedules *Abort-Shielded Consistency* or *ASC*, which define a correctness criterion in which an aborted transaction does not affect the transactions that follow it. Then we present *Conflict Preserving Abort-Shielded Consistency* or *CP-ASC*, a subset of *ASC* based on *optConf*. The membership of *CP-ASC* can be tested in polynomial time. Using *CP-ASC*, we give the design of a scheduler *CP-ASC-Sched* for scheduling interleaving nested transactions.

We consider the following schedule  $S_9$  for illustrating this class.

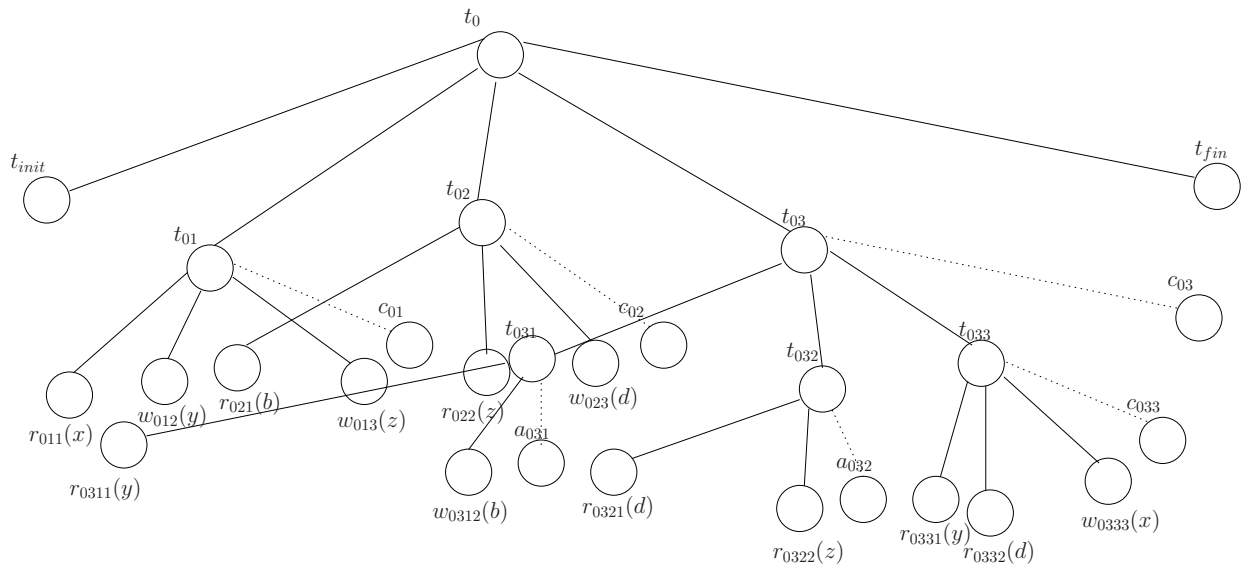
**Example 7**  $t_0 : \{t_{init}, t_{01}, t_{02}, t_{03}, t_{fin}\}$ ,  
 $t_{01} : \{sm_{011} = r_{011}(x), sm_{012} = w_{012}(y), sm_{013} = w_{013}(z), c_{01}\}$ ,  
 $t_{02} : \{sm_{021} = r_{021}(b), sm_{022} = r_{022}(z), sm_{023} = w_{023}(d), c_{02}\}$ ,  
 $t_{03} : \{t_{031}, t_{032}, t_{033}, c_{03}\}$ ,  
 $t_{031} : \{sm_{0311} = r_{0311}(y), sm_{0312} = w_{0312}(b), a_{031}\}$ ,  
 $t_{032} : \{sm_{0321} = r_{0321}(d), sm_{0322} = r_{0322}(z), a_{032}\}$ ,  
 $t_{033} : \{sm_{0331} = r_{0331}(y), sm_{0332} = r_{0332}(d), sm_{0333} = w_{0333}(x), c_{033}\}$ ,

*Schedule:*

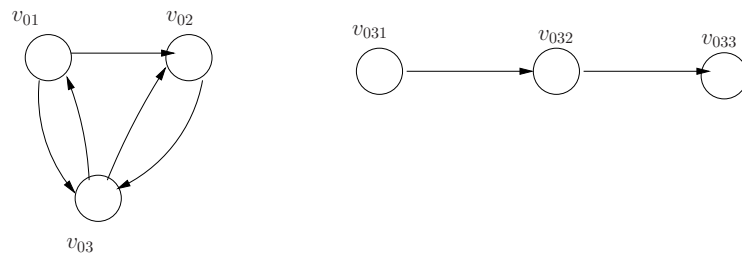
$S_9 : r_{011}(x)r_{0311}(y)w_{012}(y)r_{021}(b)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{022}(z)w_{0312}(b)a_{031}r_{0321}(d)w_{023}(d)w_{02}^{023}(d)c_{02}r_{0322}(z)a_{032}r_{0331}(y)r_{0332}(d)w_{0333}(x)w_{033}^{0333}(x)c_{033}w_{03}^{033}(x)c_{03}$

In this schedule,  $r_{0311}(y)$  reads from  $t_{init}$ , whereas  $w_{01}^{012}(y)$  of  $t_{01}$  writes in  $y$ . But  $r_{0322}(z)$  reads from  $w_{01}^{013}(z)$  of  $t_{01}$ . Thus between two external-reads of  $t_{03}$ , we have  $t_{01}$ 's updates. Hence there is no serial schedule equivalent to it. As a result it is not in CNO. The *optConf* serialization graph for this schedule is shown in Figure 11 which shows that  $S_9$  is not in *CP-CNO*.

Consider a schedule  $S$  with an aborted transaction  $t_A$ . If the aborted transaction should not affect the transactions following it, then  $t_A$  should be dropped from the schedule while considering the correctness of



**Figure 10: Computation tree for Example 7**



**Figure 11: The graph shows the CP-CNO conflict graph for  $S_9$**

the remaining transactions. Generalizing this idea to all aborted transactions, we construct a sub-schedule which consists of events only from committed transactions (sub-transactions) and no event from any aborted transaction. It does not contain events from committed transactions that are sub-transactions of aborted transactions as well. Thus, the sub-schedule consists of all the events from  $S.prune(t_0)$ . For simplicity we will denote this sub-schedule as  $commitSubSch_0$ . Then we check for the correctness of  $commitSubSch_0$ . This idea is similar to verifying the consistency of committed transactions in virtual worlds consistency [8].

As explained in [5], it is necessary that each aborted transaction  $t_A$  also reads consistent values. To ensure this, we construct a sub-schedule of  $S$  denoted as  $pprefSubSch_A$  (pruned prefix sub-schedule) for  $t_A$ . For this, we consider the prefix of all the events until  $t_A$ 's abort operation. From this prefix we construct the sub-schedule by removing (1) events from transactions that aborted earlier and (2) events of any aborted sub-transaction of  $t_A$ . Thus, the sub-schedule consists of events from transactions that committed before  $t_A$  and events from live transactions, i.e., transactions that have not yet terminated when  $t_A$  is aborted. The ordering among the events is same as in the original schedule  $S$ .

Finally, for every live transaction which including the ancestors of  $t_A$ , we add a commit operation after  $t_A$ 's abort operation to the sub-schedule. But we do not add the commit-writes for these transactions. The ordering among the commit operations is such that an ancestor's commit operation is added only after all its children's commit operations (which are also ancestors of  $t_A$ ) have been added. This ensures that well-formedness of the sub-schedule is maintained. By adding the commit operations, we ensure that all the transactions in the sub-schedule have a terminal operation. Then we look for the correctness of this sub-schedule. In  $S_9$ , for the aborted transaction  $t_{031}$ ,  $pprefSubSch_{031}$  is:

$$r_{011}(x)r_{0311}(y)w_{012}(y)r_{021}(b)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{022}(z)w_{0312}(b)a_{031}c_{02}c_{03}.$$

Similarly the sub-schedules for every aborted transaction can be constructed.

The set of  $pprefSubSch$ s for the schedule  $S_9$  are,

$$\begin{aligned} commitSubSch_0 &= r_{011}(x)w_{012}(y)r_{021}(b)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{022}(z)w_{023}(d)w_{02}^{023}(d)c_{02}r_{0331}(y) \\ & r_{0332}(d)w_{0333}(x)w_{033}^{0333}(x)c_{033}w_{03}^{033}(x)c_{03} \\ pprefSubSch_{031} &= r_{011}(x)r_{0311}(y)w_{012}(y)r_{021}(b)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{022}(z)w_{0312}(b)a_{031}c_{02}c_{03} \\ pprefSubSch_{032} &= r_{011}(x)w_{012}(y)r_{021}(b)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{022}(z)r_{0321}(d)w_{023}(d)w_{02}^{023}(d)c_{02} \\ & r_{0322}(z)a_{032}c_{03} \end{aligned}$$

From the definition of  $pprefSubSch$  we can prove that  $pprefSubSch$ s are causally complete stated in the following lemma,

**Lemma 33** *For every aborted transaction  $t_A$  in  $S$ , the sub-schedule  $pprefSubSch_A$  is causally complete.*

**Proof:** This follows from the construction of  $pprefSubSch$ . The sub-schedule  $pprefSubSch_A$  contains events either from transactions that committed before  $t_A$  or transactions that have not yet terminated. Thus, all the events that affects  $t_A$  are in  $pprefSubSch_A$ . Hence it is causally complete.  $\square$

For a schedule  $S$ , we define a set of well-formed sub-schedules denoted as  $subSchSet$ . It consists of the following sub-schedules:

1. The sub-schedule  $commitSubSch_0$  is in  $subSchSet$ . Formally,  $\langle commitSubSch_0 \in subSchSet \rangle$
2. For every aborted transaction  $t_A$  in  $S$ , there exists a  $pprefSubSch$ ,  $pprefSubSch_A$  in  $subSchSet$ ,  $\langle \forall t_A : pprefSubSch_A \in subSchSet \rangle$

Using  $subSchSet$ , we define a class of schedules, *Abort-Shielded Consistency* or *ASC* as:

**Definition 6** A schedule  $S$  belongs to ASC class if for every sub-schedule  $subS$  in the set  $subSchSet$  of  $S$ , there exists a serial sub-schedule  $sSubS$  such that:

1. *Sub-Schedule Event Equivalence*: The events of  $subS$  and  $sSubS$  are the same. Formally,  $\langle subS.evts = sSubS.evts \rangle$
2. *schedule-partial-order Equivalence*: For any two nodes  $n_Y, n_Z$  that are peers in the computation tree represented by  $subS$  if  $n_Y$  occurs before  $n_Z$  in  $subS$  then  $n_Y$  occurs before  $n_Z$  in  $sSubS$  as well. Formally,  $\langle t_X : \{n_Y, n_Z\} \subseteq subS.children(t_X) : (n_Y <_{subS} n_Z) \Rightarrow (n_Y <_{sSubS} n_Z) \rangle$
3. *lastWrite Equivalence*: For all read operations the lastWrites in  $subS$  and  $sSubS$  are the same. Formally,  $\langle \forall r_X \in subS : subS.lastWrite(r_X) = sSubS.lastWrite(r_X) \rangle$

Similarly using  $pprefSubSch$  we define an extension to CP-CNO, *Conflict Preserving Abort Shielded Consistency* or *CP-ASC*. It differs from the definition of the class ASC only in the case 3 as:

3. *optConf Implication*: If two memory operations in  $subS$  are in  $optConf$  then they are also in  $optConf$  in  $sSubS$ . Formally,

$$\langle \forall m_Y, \forall m_Z : \{m_Y, m_Z\} \subseteq subS.evts : (subS.optConf(m_Y, m_Z) \Rightarrow sSubS.optConf(m_Y, m_Z)) \rangle$$

For this class, we get the following lemmas

**Lemma 34** If a schedule  $S$  is in CNO then it is also in ASC. Formally  $\langle CNO \subset ASC \rangle$

**Proof:** Consider a schedule  $S$  in CNO. Then, from the definition of CNO we get that there exists a serial schedule  $SS$  such that the lastWrites of  $S$  and  $SS$  are the same. Thus for any sub-schedule  $subS$  of  $S$  that is causally complete, there exists a serial sub-schedule  $sSubS$  that is serial and has the same lastWrites of  $subS$ , by Lemma 32.

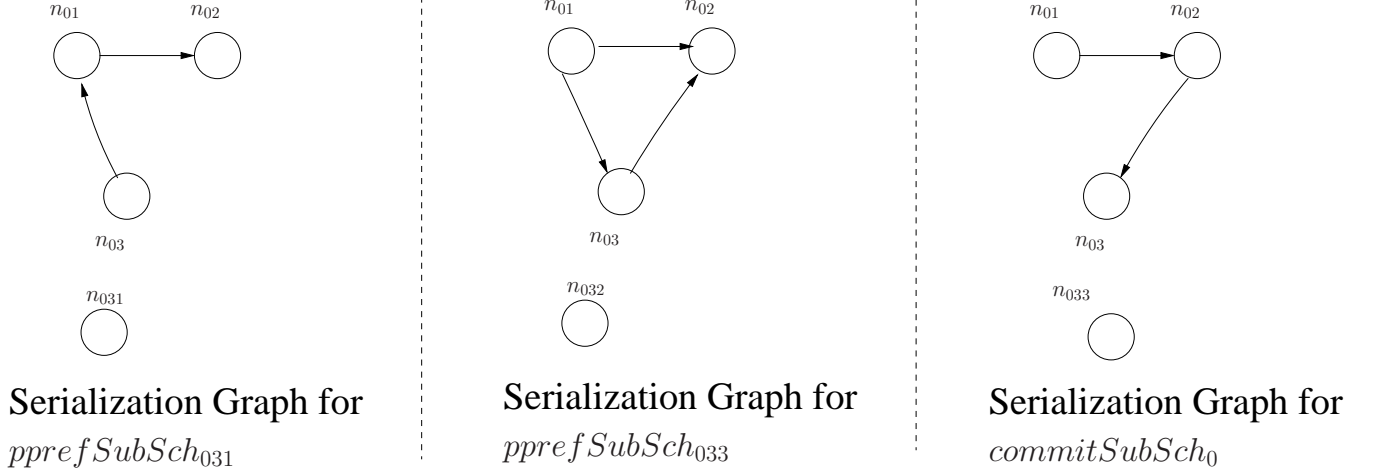
To prove that  $S$  is also in ASC, we have to prove that,

- For  $commitSubSch_0$ , there exists a serial sub-schedule, namely  $commitSerSubSchSS_0$ : It must be noted that  $commitSubSch_0$  is a sub-schedule of  $S$  and is causally complete. Since  $S$  is in CNO, from Lemma 32 we get that there exists a serial sub-schedule  $commitSerSubSchSS_0$ .
- The sub-schedule  $pprefSubSch_A$  for every aborted transaction,  $t_A$  has an equivalent serial sub-schedule: From Lemma 33 we get that the sub-schedule  $pprefSubSch_A$  is causally complete. Hence the reasoning for this case is same as the above case.

This completes the proof. □

It can be verified that schedule  $S_9$  is in ASC but not in CNO. Hence, the class CNO is a strict subset of ASC.

**Lemma 35** If a schedule  $S$  is in CP-ASC then it is also in ASC. Formally  $\langle CP-ASC \subset ASC \rangle$



**Figure 12: The serialization graphs for the schedule in Example 7. This shows that this schedule is in CP-ASC**

**Proof:** The proof is similar to Theorem 14. □

It can be seen that verifying whether  $S$  is in CP-ASC or not can be done in polynomial time. From the schedule  $S$ , the sub-schedules  $commitSubSch_0$  and  $pprefSubSch_A$  for every aborted transaction  $t_A$  are constructed. Then for each sub-schedule, the serialization graph is constructed using `optConfGraphCons` algorithm based on `optConf`. If all the graphs constructed are acyclic, then the schedule  $S$  is in CP-ASC. The equivalent serial sub-schedules for the sub-schedules is constructed from the graphs using the expander algorithm.

For the schedule  $S_9$  of Example 7, the set of serial *pprefSerSubSchs* are as follows where  $commitSerSubSchSS_0$  is the serial sub-schedule corresponding to  $commitSubSch_0$ ,

$$commitSerSubSchSS_0 = t_{01}t_{02}t_{03} = r_{011}(x)w_{012}(y)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{021}(b)r_{022}(z)w_{023}(d)w_{02}^{023}(d)c_{02}r_{0331}(y)r_{0332}(d)w_{0333}(x)w_{033}^{0333}(x)c_{033}w_{03}^{033}(x)c_{03}$$

$$pprefSerSubSch_{031} = t_{03}t_{01}t_{02} = r_{0311}(y)w_{0312}(b)a_{031}c_{03}r_{011}(x)w_{012}(y)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{021}(b)r_{022}(z)c_{02}$$

$$pprefSerSubSch_{032} = t_{01}t_{03}t_{02} = r_{011}(x)w_{012}(y)w_{013}(z)w_{01}^{012}(y)w_{01}^{013}(z)c_{01}r_{0321}(d)r_{0322}(z)a_{032}c_{03}r_{021}(b)r_{022}(z)w_{023}(d)w_{02}^{023}(d)c_{02}$$

The CP-ASC serialization graphs are shown in Figure 12.

#### 4.5 CP-ASC-Sched: A scheduler based on CP-ASC

In this section we give the outline of a scheduler, called as CP-ASC-Sched (CP-ASC Scheduler) which implements the class CP-ASC. When a transaction wants to read, write or commit, it sends the request to the scheduler CP-ASC-Sched. The scheduler on receiving a request from a transaction, checks with the previously committed and live transactions to see if the request maintains the consistency. If it does, then CP-ASC-Sched allows the request to proceed. Otherwise it does not allow the request to proceed and aborts the corresponding transaction. Consistency is checked by adding the appropriate conflict edges in the

conflict graph and checking for its acyclicity.

The scheduler maintains a conflict graph for each transaction  $t_P$ , denoted as  $G_P$ . The scheduler CP-ASC-Sched implements CP-ASC using `optConfGraphCons` algorithm (described in Section 3.3) as follows:

1. On receiving a request from a transaction  $t_P$  to invoke a new transaction  $t_S$ , a node  $v_S$  is created in  $G_P$ . Then CP-ASC-Sched adds completion edges from all the peers of  $t_S$  that have terminated earlier to  $v_S$ .
2. On receiving a read request  $r_X(d)$  from a transaction  $t_P$ , CP-ASC-Sched creates a node  $v_X$  for  $r_X$  in  $G_P$  and adds completion edges from all the peer nodes of  $r_X$  that completed before it. Let  $r_X$ 's last-Write be  $w_L$ ,  $w_L$  be a commit-write of a node  $n_L$  (either a transaction or simple-memory operation) and  $w_L$ 's parent be  $t_Q$  ( $t_Q$  is same as  $t_P$  if  $w_L$  is a peer of  $r_X$ ). Also let  $n_K$  be a peer node of  $n_L$  in whose dSet is the read  $r_X$  is contained. Then CP-ASC-Sched adds a w-r conflict edge from  $v_L$  to  $v_K$  in  $G_Q$ . Then, the read  $r_X$  is stored as an external-read in all its ancestors starting from  $t_P$  ending at  $n_K$ .
3. On receiving a write request  $w_Y(d)$  from a transaction  $t_P$ , CP-ASC-Sched adds a node  $v_Y$  in the graph. Then it adds completion edges from all the peers of  $w_Y$  that have completed before it. For any peer node  $n_Z$  of  $w_Y$  that has an external-read  $r_X(d)$ , a r-w conflict edge is added from  $v_Z$  to  $v_Y$  in  $G_P$ . Similarly for any peer node  $n_T$  that has a commit-write  $w_T(d)$  a w-w conflict edge is added from  $v_T$  to  $v_Y$ .
4. Transaction  $t_P$  on receiving a request to commit from a transaction  $t_X$ , CP-ASC-Sched adds r-w and w-w conflict edges w.r.t the commit-writes of  $t_X$  (similar to step 3). It adds these edges between  $n_X$  and its corresponding peers in  $G_P$ .

After adding the edges, CP-ASC-Sched checks if these edges form a cycle in  $G_P$ . If no cycle is formed, then the requested action of the transaction is permitted. Otherwise, the requested action is not permitted. The corresponding transaction  $t_P$  (or  $t_X$ ) and all its live descendant transactions are aborted (the status of committed sub-transactions of the aborted transaction remain unchanged). The vertex and edges of  $t_P$  are removed from the graph. All the reads in  $t_P$ 's dSet that are stored as external-reads in its ancestors are removed. In this way, an aborted transaction does not affect any other transaction that follows it. With this implementation, we get that any schedule accepted by CP-ASC-Sched is also in CP-ASC.

We note that the scheduler can be implemented in a completely distributed manner. The different components of the graph can be maintained by different processes. It is not necessary for any single process to have the global information. A more detailed version of this paper can be found in [15].

## 5 Discussion

### 5.1 A Simpler Conflict Notion

Having described the idea of `optConf`, in this subsection we will discuss a variant to the conflict notion. As discussed earlier (in Section 2.4), a read operation can read from the value written by a write operation only if the write is `optVis` to the read. Based on this observation, one can come up with a simpler notion of conflict between any arbitrary read and a write operation based only on `optVis`. This conflict notion does not concern if a given read operation is an external-read or not. We call such a conflict as *vConf*.

For two memory operations  $m_X, m_Y$  in the dSets of peers  $n_A, n_B$ ,  $S.vConf(m_X(d), m_Y(d))$  is true if  $m_X$  occurs before  $m_Y$  in  $S$  and one of the following conditions holds:



1. r-w conflict:  $m_X$  is a read  $r_X$  (and not necessarily an external-read) in  $n_A$ 's dSet,  $m_Y$  is a commit-write  $w_Y$  of  $n_B$  or
2. w-r conflict:  $m_X$  is a commit-write  $w_X$  of  $n_A$  and  $m_Y$  is a read  $r_Y$  in  $n_B$ 's dSet or
3. w-w conflict:  $m_X$  is a commit-write  $w_X$  of  $n_A$  and  $m_Y$  is a commit-write  $w_Y$  of  $n_B$ .

Based on this conflict definition we can define a class of schedules called as *Visible Conflict Preserving Closed Nested Opacity* or *VCP-CNO*. It is very similar to CP-CNO and differs only in condition 3 of CP-CNO definition, the conflict implication. It is defined as below:

vConf Implication: if two memory operations in  $S$  are in vConf then they are also in vConf in  $SS$ . Formally,

$$\langle \forall m_Y, \forall m_Z : \{m_Y, m_Z\} \subseteq S.evts : (S.vConf(m_Y, m_Z) \Rightarrow SS.vConf(m_Y, m_Z)) \rangle$$

We denote this equivalence to such a serial schedule as  $(S \approx_{vc} SS)$ . From the definitions of vConf and optConf we get that in a given schedule  $S$ , if two memory operations  $m_X, m_Y$  are in optConf then they are also in vConf i.e.  $S.optConf(m_Y, m_Z) \Rightarrow S.vConf(m_Y, m_Z)$ . From this one can prove that if any schedule  $S$  is in VCP-CNO then it is also in CP-CNO i.e.  $(S \in VCP-CNO) \Rightarrow (S \in CP-CNO)$ . But the class VCP-CNO is not as generic as CP-CNO. There are some schedules which are in CP-CNO but not in VCP-CNO. The following example illustrates it.

**Example 8 Computation Tree:**

$$\begin{aligned} t_0 &: \{t_{init}, t_{01}, t_{02}, t_{03}, t_{fin}\}, \\ t_{01} &: \{r_{011}(x), w_{012}(y), c_{01}\}, \\ t_{02} &: \{r_{021}(d), w_{022}(x), w_{023}(y), c_{02}\}, \\ t_{03} &: \{t_{031}, t_{032}, sm_{033} = w_{033}(z), c_{03}\}, \\ t_{031} &: \{sm_{0311} = r_{0311}(z), sm_{0312} = w_{0312}(y), c_{031}\}, \\ t_{032} &: \{sm_{0321} = r_{0321}(y), sm_{0322}(x) = w_{0322}(x), c_{032}\} \end{aligned}$$

*Schedule:*

$$S_{10} : r_{011}(x)r_{021}(d)w_{012}(y)r_{0311}(z)w_{012}^{01}(y)c_{01}w_{0312}(y)w_{031}^{0312}(y)c_{031}w_{022}(x)r_{0321}(y)w_{0322}(x)w_{032}^{0322}(x) \\ c_{032}w_{023}(y)w_{02}^{022}(x)w_{02}^{023}(y)c_{02}w_{033}(z)w_{03}^{031}(y)w_{03}^{032}(x)w_{03}^{033}(z)c_{03}$$

The corresponding computation tree is shown in Figure 13. The lastWrites for all the reads in the above example are:

$$\{r_{011}(x) : w_{init}(x), r_{021}(d) : w_{init}(d), r_{0311}(z) : w_{init}(z), r_{0321}(y) : w_{031}^{0312}(y)\}$$

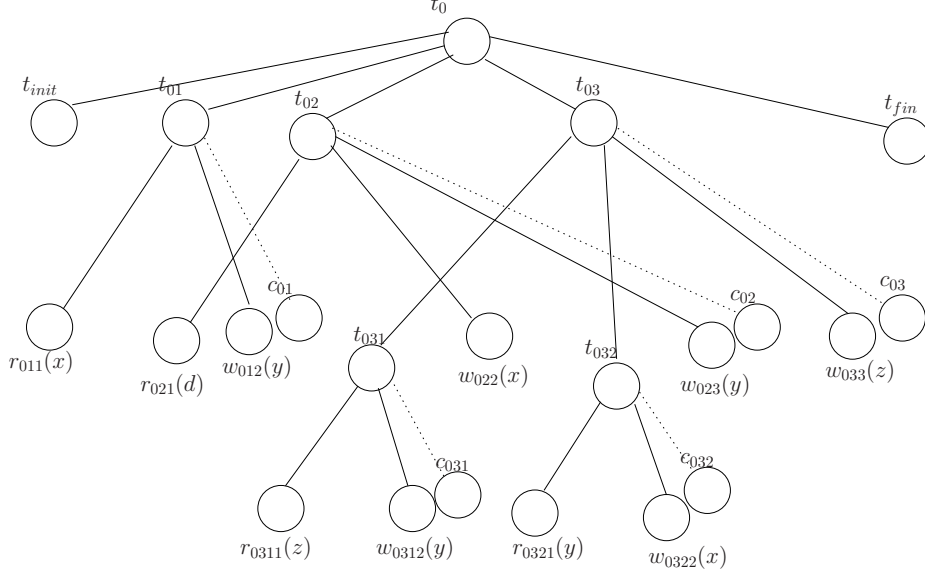
The set of all optConfs in the above example:

$$\{(r_{011}(x), w_{02}^{022}(x)), (r_{011}(x), w_{03}^{032}(x)), (r_{0311}(z), w_{033}(z)), (w_{02}^{022}(x), w_{03}^{032}(x)), (w_{031}^{0312}(y), r_{0321}(y)), \\ (w_{01}^{012}(y), w_{02}^{023}(y)), (w_{01}^{012}(y), w_{03}^{031}(y)), (w_{02}^{023}(y), w_{03}^{031}(y))\}$$

The set of all vConfs in the above example:

$$\{(r_{011}(x), w_{02}^{022}(x)), (r_{011}(x), w_{03}^{032}(x)), (r_{0311}(z), w_{033}(z)), (w_{02}^{022}(x), w_{03}^{032}(x)), (w_{031}^{0312}(y), r_{0321}(y)), \\ (w_{01}^{012}(y), w_{02}^{023}(y)), (w_{01}^{012}(y), w_{03}^{031}(y)), (w_{02}^{023}(y), w_{03}^{031}(y)), (w_{01}^{012}(y), r_{0321}(y)), (r_{0321}(y), w_{02}^{023}(y))\}$$

We have underlined the extra conflicts in this example. We did not mention the conflicts caused by  $t_{init}$  and  $t_{fin}$  in the above conflicts. We discussed the optConfGraphCons algorithm in the previous section to verify if a given schedule  $S$  is in CP-CNO or not. This algorithm can be easily adapted to verify if the schedule  $S$  is in VCP-CNO or not. This algorithm differs only in the way conflict edges are added. We add a conflict edge when two memory operations are in vConf instead of optConf. We call this algorithm vGraphCons algorithm.



**Figure 13: The computation tree for Example 8**

Using `optConfGraphCons` algorithm and `vGraphCons` algorithm we generate the serialization graphs based on both these conflicts. The graphs are shown in Figure 14. The graphs show that the schedule  $S_{10}$  is in CP-CNO but not in VCP-CNO. As one can see from the conflict sets,  $r_{0321}(y)$  and  $w_{02}^{023}(y)$  are in `vConf` but not in `optConf`. They cause the cycle between  $n_{03}$  and  $n_{02}$  in the graph of VCP-CNO. This shows that VCP-CNO is a proper subset of CP-CNO. The `optConf` equivalent serial schedule is:

$$r_{011}(x)w_{012}(y)w_{012}^{01}(y)c_{01}r_{021}(d)w_{022}(x)w_{023}(y)w_{02}^{022}(x)w_{02}^{023}(y)c_{02}r_{0311}(z)w_{0312}(y)w_{031}^{0312}(y)c_{031}r_{0321}(y)w_{0322}(x)w_{032}^{0322}(x)c_{032}w_{033}(z)w_{03}^{031}(y)w_{03}^{032}(x)w_{03}^{033}(z)c_{03}$$

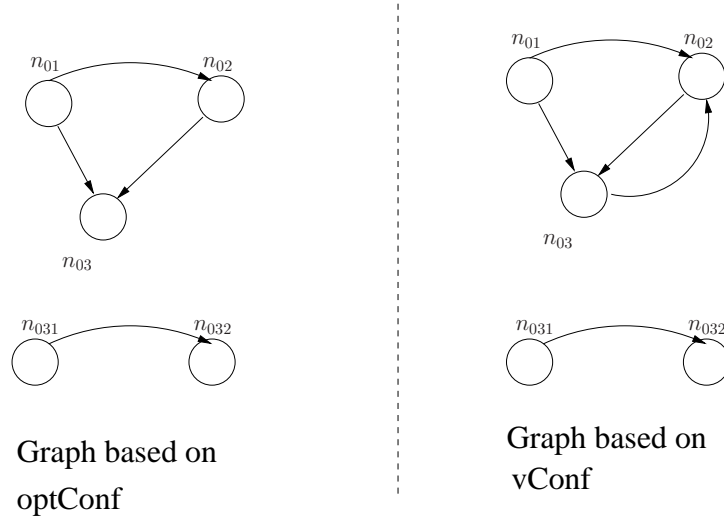
## 5.2 Schedule Partial Order

The second condition in the definitions of the classes CNO and ASC is schedule-partial-order. This condition specifies that for any two peer nodes (transactions or simple-memory operation), say  $n_Y, n_Z$ , in the schedule  $S$  such that  $n_Y$  executes before  $n_Z$  then in the corresponding serial schedule  $SS$ ,  $n_Y$  executes before  $n_Z$  as well. But for some nested STM systems this may not be sufficient. The application that generates the transactions might dictate the STM to be more strict. These systems might want that the condition schedule-partial-order to be modified such that if any node  $n_Y$  occurs before any other transaction  $n_Z$  in  $S$ , then in  $SS$  also  $n_Y$  occurs before  $n_Z$ . That is, the nodes  $n_Y$  and  $n_Z$  need not be peers but any arbitrary nodes. Thus the condition 2 of CNO can restated as follows:

schedule-partial-order Equivalence: For any two nodes  $n_X, n_Y$  in the computation tree represented by  $S$  if  $n_Y$  occurs before  $n_Z$  in  $S$  then  $n_Y$  occurs before  $n_Z$  in  $SS$  as well. Formally,

$$\langle S : \{n_Y, n_Z\} \in S.nodes : (n_Y <_S n_Z) \Rightarrow (n_Y <_{SS} n_Z) \rangle$$

This modification can be made to the definitions of CP-CNO and CP-ASC. To accommodate this change in the graph construction `optConfGraphCons` algorithm, we modify the way completion edges are added. Consider the nodes  $n_Y, n_Z$  in  $S$  for which  $(n_Y <_S n_Z)$ . Let  $t_P$  be a transaction such that it is the least common ancestor of  $n_Y, n_Z$ , i.e.,  $S.lca(n_Y, n_Z) = t_P$ . Since  $n_Y$  occurred before  $n_Z$  in  $S$ ,  $n_Y$  cannot be a



**Figure 14:** These are the serialization graphs based on `optConf` and `vConf` for the schedule in **Example 8**

ancestor of  $n_Z$  nor the vice-versa. Hence  $t_P$  cannot be the same as  $n_Y$  or  $n_Z$  but an ancestor to both. Thus  $t_P$  will have two children  $n_R$  and  $n_T$  such that  $n_Y$  is in  $S.dSet(n_R)$  and  $n_Z$  is in  $T.dSet(n_Q)$ . Next we add a completion edge from  $n_R$  to  $n_T$  in the graph. Then we check for acyclicity of the resulting graph. If the graph is acyclic then in the resultant schedule  $RS$  generated,  $n_Y$  will be before  $n_Z$  i.e.  $n_Y <_{RS} n_Z$ .

## 6 Conclusion

Composing simple transactions to build larger transaction systems is extremely useful property which forms the basis of modular programming. In STMs this can be achieved through nesting of transactions. There have been many implementations of nested transactions in the past few years. But none of them provide a precise and efficient formulation of the guarantees that a nested software transactional memory system should provide.

Concurrent executions of transactions in Transactional Memory are expected to ensure that aborted transactions also, as the committed ones, read consistent values. In addition, the property that aborted transactions should not affect the consistency for the other transactions following it is desirable. Incorporating these simple-sounding criteria has been non-trivial even for non-nested transactions as can be seen in recent publications [5, 9, 3].

In this paper, we have considered these requirements for closed nested transactions. We have also defined new conflict-preserving classes that allow polynomial membership test, by means of constructing conflict-graphs and checking acyclicity. Further, the conflict preserving classes have resulted in the elegant design of a scheduler. The conflict-graph has separate components for each (parent) sub-transaction. Each component can be maintained at a different site (process executing the sub-transaction) autonomously and the checking can be done in a distributed manner.

We have chosen a novel representation of schedules, namely, adding commit-writes, that facilitates easy association of lastWrites for the read operations. We believe that this representation will be useful for dealing with commit-pending transactions also. Our future work includes the study of how the above two properties

manifest in executions with open nested transactions and with non-transactional steps.

## References

- [1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.
- [2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 70–81, New York, NY, USA, 2006. ACM.
- [3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards Formally Specifying and Verifying Transactional Memory. In *REFINE*, 2009.
- [4] Rachid Guerraoui, Thomas Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *DISC '08: Proc. 22nd International Symposium on Distributed Computing*, pages 305–319, sep 2008. Springer-Verlag Lecture Notes in Computer Science volume 5218.
- [5] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.
- [6] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.
- [7] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
- [8] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 280–281, New York, NY, USA, 2009. ACM.
- [9] Damien Imbs and Michel Raynal. A lock-based stm protocol that satisfies opacity and progressiveness. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 226–245, Berlin, Heidelberg, 2008. Springer-Verlag.
- [10] J.E.B. Moss. Open Nested Transactions: Semantics and Support. In *Workshop on Memory Performance Issues*, 2006.
- [11] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM.
- [12] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

- [13] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM.
- [14] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
- [15] Sathya Peri and K.Vidyasankar. An efficient scheduler for closed nested transactions that satisfies all-read-consistency and non-interference. In *13th International Conference on Distributed Computing and Networking*, Hongkong, China, 2012.
- [16] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Inf. Process. Lett.*, 50(4):177–183, 1994.
- [17] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.
- [18] K. Vidyasankar. Generalized theory of serializability. *Acta Inf.*, 24(1):105–109, 1987.
- [19] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.