A Definition of Data Consistency Using Event Lattices

Christian Rehn
Institut für Informatik
Technische Universität München
85748 Garching, Germany

Abstract *Replication of data is a common technique to enhance performance in distributed systems in which multiple activities use shared passive objects. If replicated data is updateable rather than read-only, modifications must be propagated to other copies of the replicated data in order to assure a consistent view. The delay of update propagations affects the observations of activities running on the different nodes of a distributed system. Read operations on the same passive object may return different values depending on the nodes they are executed on and the replica being accessed. A memory consistency model defines the legal ordering of data modifications issued by an activity on a node of the distributed system, as observed by remote activities. This paper presents a formal definition of the term* data consistency *using an event lattice model. With this definition the consistency of an arbitrary execution of a concurrent system can be determined.*

Keywords: consistency model, event lattices, distributed systems, shared memory paradigm

1 Introduction

Replication of data is a common technique to enhance performance in distributed systems in which multiple activities use shared passive objects for indirect communication. Replication is the ability to provide local copies of data, so that there is no need to access that information across the network, thus providing improved local processing performance.

If replicated data is updateable rather than read-only, modifications must be propagated to other copies of the replicated data in order to assure a consistent view. To reduce network traffic and maintain reasonable response times and system throughput rates, changes are often not instantly propagated to the copies but delayed until certain synchronization operations are executed. The extent to which updates may be deferred is variable. In particular they vary as to the degree of data consistency they provide as opposed to the amount of independence that is supported.

The delay of update propagations affects the observations of activities running on the different nodes of a distributed system. Read operations on the same passive object may return different values depending on the nodes they are executed on and the replica being accessed. A *memory consistency model* defines the legal ordering of data modifications issued by an activity on a node of the distributed system, as observed by remote activities. A *strong consistency model* means that updates are propagated more or less immediately between replicas. Therefore read operations on a data item should return a value that shows the result of the 'last'[1] write operation on that data. *Weak consistency models* allow the delay of update propagation and result in different local views on the ordering of data modifications and the actual value. Therefore the consistency model restricts the values that a read operation on a data item can return.

[1] As there normally is no global clock in a distributed system it is difficult to define precisely which write operation is the last one.

This paper presents a formal definition of data consistency in distributed systems using an event lattice model. Section 2 introduces the model of event lattices to describe the execution of activities in a distributed system. In section 3 this model is used to define the terms 'consistent execution' and 'consistency rule'. Section 4 combines the results from section 2 with the consistency models known from literature. Finally section 5 summarizes the results of this paper.

2 Event Lattice Model

The execution of a single activity results in a program-ordered linear event trace [DR95]. Assuming that a *start-synchronization* takes place between a dynamically created activity and its creator and a *stop-synchronization* occurs between a terminating activity and its creator, the event traces can be combined according to their creation and termination dependencies in order to model a global view on a distributed execution of a concurrent program. This combination of traces results in an event lattice structure [Reh03].

Definition 1 (event lattice)
An event lattice $\mathcal{L}$ *is a 4-tuple* $(E, \sqsubseteq, D, W)$ *where*

- E *is a finite, nonempty set of events;*
- $\sqsubseteq$ *is a partial order on* E*;* $e_1 \sqsubseteq e_2, e_1 \neq e_2$ *implies that the computation of event* e_2 *depends on* e_1*, i.e. the execution of* e_1 *must be finished before* e_2 *can be executed;*
- $(E, \sqsubseteq)$ *is a lattice;*
- D *is a finite nonempty set of data objects;*
- W *is the family* $(W_e \mid e \in E)$ *of the spheres of action of the events in* E*. Each sphere of action* W_e *is a pair* (V_e, N_e) *where* $V_e \subseteq D$ *is the set of input data and* $N_e \subseteq D$ *is the set output data of event* e*.*

Figure 1 illustrates the Hasse diagram[2] for the event lattice in example 1 showing the execution of three activities A, B and C. Event lattices can describe executions at different levels of granularity. In figure 1 an event $e \in E$ models a read, write or synchronization event which is appropriate for defining consistency. Examples for read events are a_4 and b_4, for write events a_3 and b_3. Synchronization events are involved in the coordination of executions. It may be the creation of a new activity (start-synchronization, for example a_1 and b_1 in figure 1), a stop-synchronization (for example b_6 and a_5) or any other alternative to synchronize the execution of activities as for example synchronous message passing (a_2, c_2) or barriers (ab_1). An instance of an event lattice for this level of granularity looks as follows:

Example 1

- $E = \{a_1, a_2, \ldots, a_5, ab_1, b_1, b_2, \ldots, b_6, c_1, c_2, c_3, c_4\}$,
- $\sqsubseteq$ *is the reflexive and transitive closure of* $\{(a_1, a_2), (a_2, a_3), (a_3, ab_1), (ab_1, a_4), (a_4, a_5), (a_1, b_1), (b_1, b_2), (b_2, b_3), (b_3, ab_1), (ab_1, b_4), (b_4, b_5), (b_5, b_6), (b_2, c_1), (b_6, a_5), (c_1, c_2), (c_2, c_3), (c_3, c_4), (a_2, c_2), (c_4, b_5)\}$,
- $D = \{x, y\}$
- $\forall e \in E \backslash \{a_4, b_4\} : V_e = \emptyset$, $V_{a_4} = \{y\}$, $V_{b_4} = \{x\}$, $\forall e \in E \backslash \{a_3, b_3, c_3\} : N_e = \emptyset$, $N_{a_3} = N_{c_3} = \{x\}$, $N_{b_3} = \{y\}$

A single event lattice $\mathcal{L}$ models not only one execution but a set of executions $ex(\mathcal{L}) = \{x_1, x_2, \ldots, x_n\}$ as $\sqsubseteq$ only describes the minimal ordering requirements on E. For example all sequential executions on a single node $ex_{seq}(\mathcal{L}) \subseteq ex(\mathcal{L})$ which means all extensions of the $\sqsubseteq$ relation that are a total order on E are executions of $\mathcal{L}$.

The event lattice model introduced above can describe the data objects involved in an

[2] Here the smallest element is on the left in contrast to the usual illustrations of Hasse diagrams where the smallest element is at the bottom.

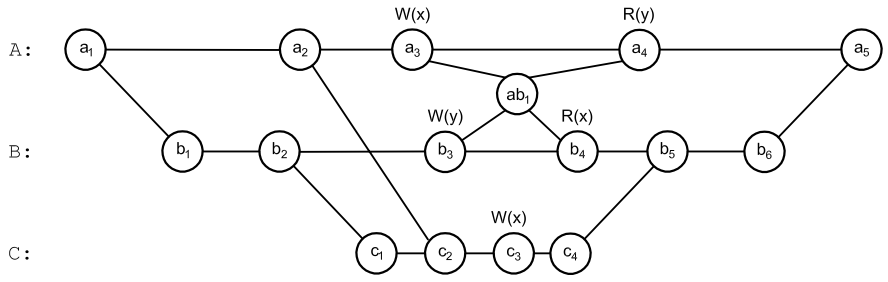

Figure 1: Hasse diagram of an event lattice

event occurrence but not the range of values a data object can hold and the effect of an event execution on a data object. In order to include the semantics of a system execution, an interpretation $\mathcal{I}$ must be added to an event lattice $\mathcal{L}$.

Definition 2 (interpreted event lattice) *An* interpretation $\mathcal{I}$ of an event lattice $\mathcal{L}$ *is a pair* $(\mathcal{Z}, \mathcal{F})$ *where*

- $\mathcal{Z}$ *is the family* $(Z(d) \mid d \in D)$ *of ranges of values of the data objects* $d \in D$*;*
- $\mathcal{F}$ *is the family* $(f_e \mid e \in E)$ *of functions of the events in* E*. For each event* $e \in E$ *the function* f_e *is a total mapping*

$$f_e : \bigtimes_{v \in V_e} Z(v) \rightarrow \bigtimes_{n \in N_e} Z(n)$$

An interpreted event lattice $(\mathcal{L}, \mathcal{I})$ *is composed of an event lattice* $\mathcal{L}$ *and an associated interpretation* $\mathcal{I}$*.*

In the following we write the functions f_e for the read and write events $e \in E$ as a pair $a(b)$ where $a \in \{R, W\}$ indicates if it is a read or a write event and b is the identifier of the data object. We assume that for a read event e the sphere of action is given by $V_e = \{b\}$ and $N_e = \emptyset$ and for a write event e' by $V_{e'} = \emptyset$ and $N_{e'} = \{b\}$. To define consistency it is necessary to choose an appropriate event granularity. Therefore an instruction that reads the values of a number of data objects and writes a result computed from the returned values to a data object is considered as a number of single read events and one write event if the instruction is not executed atomically. Figure 1 already shows the interpreted version of the event lattice from example 1.

For a single event lattice $\mathcal{L}$ and a given interpretation $\mathcal{I}$ there may exist *distinguishable executions* as the read operations may return different values according to the actual event occurrences of unordered, i.e. *concurrent events.*

Definition 3 (concurrent events) *Let* $\mathcal{L} = (E, \sqsubseteq, D, W)$ *be an event lattice. We call two events* e_i *and* e_j concurrent *if they are not ordered by the relation* $\sqsubseteq$*, i.e.* $(e_i, e_j) \notin \sqsubseteq$ *and* $(e_j, e_i) \notin \sqsubseteq$*.*

Events a_3 and c_3 are an example for concurrent events in figure 1.

The set of executions $ex(\mathcal{L})$ of the uninterpreted event lattice is segmented into disjoint non-empty subsets $X_1, X_2, \ldots, X_l$, where X_i is the set of undistinguishable executions of the interpreted event lattice $(\mathcal{L}, \mathcal{I})$. In order to describe a subset $X_i \in ex((\mathcal{L}, \mathcal{I}))$ the pairs $a(b)$ presenting read and write events can be extended to triples $a(b)c$ where c is the value read or written in an execution $x \in X_i$. We write $ex(e) = a(b)c$, $e \in E$ for a read or written event occurrence.

3 Consistent Executions

After introducing the event lattice model the term 'consistency' is defined in this section based on event lattices. We start by asking: what is a minimal consistent execution, what are the least possible demands on the execution of a concurrent system regarding the values returned by read events. To answer this question

we take a look at the $\sqsubseteq$ relation of an event lattice $\mathcal{L} = (E, \sqsubseteq, D, W)$. $\sqsubseteq$ is the result of the the total order of events executed in a single activity (*activity-order*) and the order introduced by synchronization dependencies (*sync-order*). A minimal consistent execution must definitely be consistent with the $\sqsubseteq$ relation. In addition there is a natural order on read and write events as a value cannot be read before it has been written. We call this dependency the *causal-order*.

Definition 4 (causal-order $\sqsubseteq_c$)
Let $\mathcal{L} = (E, \sqsubseteq, D, W)$ be an event lattice and let $(\mathcal{L}, \mathcal{I})$ be an interpreted event lattice. Assuming that all values $c \in Z(b)$ written to data objects $b \in D$ in an execution $x \in ex((\mathcal{L}, \mathcal{I}))$ are distinguishable[3]*:*

$$\begin{aligned} \sqsubseteq_c &:= \{(e_i, e_j) \mid \exists b \in D, \exists c \in Z(b) : \\ &\quad e_i \in E_{W(b)c},\, e_j \in R_{R(b)c}\} \\ \textit{where} & \\ E_{W(b)c} &:= \{e \in E \mid ex(e) = W(b)c\} \\ &\quad \textit{for } b \in D,\, c \in Z(b) \\ E_{R(b)c} &:= \{e \in E \mid ex(e) = R(b)c\} \\ &\quad \textit{for } b \in D,\, c \in Z(b) \end{aligned}$$

A minimal consistent execution of an interpreted event lattice $(\mathcal{L}, \mathcal{I})$ is an execution $x \in ex((\mathcal{L}, \mathcal{I}))$, so that the Hasse diagram augmented by the causal-order is still a Hasse diagram which means it is acyclic. It is minimal in the sense that you can abandon neither the activity- nor the sync-order, as it is defined in the source code by the application developer and you cannot abandon the causal-order as the writing of a value is an indispensable condition for reading it.

Definition 5 (min. consistent execution)
Let $\mathcal{L} = (E, \sqsubseteq, D, W)$ be an event lattice, $\mathcal{I} = (\mathcal{Z}, \mathcal{F})$ an interpretation of $\mathcal{L}$ and $ex((\mathcal{L}, \mathcal{I}))$ the set of executions for $(\mathcal{L}, \mathcal{I})$. An execution $x \in ex((\mathcal{L}, \mathcal{I}))$ is minimal consistent iff $\sqsubseteq \cup \sqsubseteq_c$ is an order relation.

To illustrate minimal consistency figure 2 shows two Hasse diagram clippings. The clipping on the left is not part of a minimal consistent execution whereas the clipping on the right meets the requirements of definition 5.

Minimal consistency is the weakest demand on an execution. By adding additional rules we get stronger consistency models. One possible rule might be 'all read events of a data object $d \in D$ that read the value $v \in Z(d)$ must be executed before the write of a different value w into d'. In general consistency rules add additional order restrictions to the $\sqsubseteq$ relation of an event lattice and have the structure described in definition 6.

Definition 6 (data consistency rule)
Let $\mathcal{L} = (E, \sqsubseteq, D, W)$ be an event lattice. A data consistency rule *or short* consistency rule *on $\mathcal{L}$ is an extension $\sqsubseteq_r$ of the relations $\sqsubseteq \cup \sqsubseteq_c$ which has one of the following forms:*

- $(e_1, e_2) \in \sqsubseteq$, $e_1 \in F \subseteq E$, $e_2 \in G \subseteq E \Rightarrow$
  $\forall e_3 \in H \subseteq E, \forall e_4 \in G \subseteq E :$
  $(e_3, e_4) \in \sqsubseteq_r$ *or*

- $(e_1, e_2) \in \sqsubseteq \cup \sqsubseteq_c$, $e_1 \in F \subseteq E$,
  $e_2 \in G \subseteq E \Rightarrow$
  $\forall e_3 \in H \subseteq E, \forall e_4 \in G \subseteq E :$
  $(e_3, e_4) \in \sqsubseteq_r$

Using consistency rules we can now define the term 'consistent execution'.

Definition 7 (consistent execution)
Let $R = \{r_1, r_2, \ldots, r_m\}$ be a set of consistency rules defining the relations $\sqsubseteq_{r_1}, \sqsubseteq_{r_2}, \ldots, \sqsubseteq_{r_m}$ and let $(\mathcal{L}, \mathcal{I})$ be an interpreted event lattice. Let $\sqsubseteq_R$ be the union of all relations defined by the rules, $\sqsubseteq_R := \sqsubseteq_{r_1} \cup \sqsubseteq_{r_2} \cup \cdots \cup \sqsubseteq_{r_m}$.

An execution $x \in ex((\mathcal{L}, \mathcal{I}))$ of the interpreted event lattice is consistent in regard to the set of rules R *iff $\sqsubseteq \cup \sqsubseteq_c \cup \sqsubseteq_R$ is an order relation.*

[3] In the rest of this paper we assume that all values written to data objects are distinguishable. Therefore we can always unambiguously determine the write event a read event causally dependes on, i.e. which event wrote the value returned by a read event.

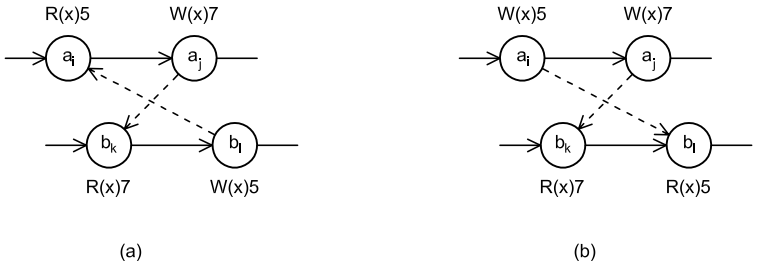

Figure 2: (a) Violation of minimal consistency requirements; (b) clipping of a minimal consistent execution;

The next section will present three consistency models described in literature and the corresponding consistency rules.

4 Consistency Rules

There exist a lot of publications mainly in the field of distributed shared memory (DSM) systems concerning consistency models [AG96, CBZ91, CBZ95, ZIS+97]. In this section we will translate three known consistency models into data consistency rules (see definition 6). If all executions that are possible with a specific memory system implementation meet a set of rules R we say the memory system implements the consistency model described by the rules in R.

We start by identifying the rules for *causal consistency* as the translation of the definition into rules is simple and straight forward. [Tan95] defines causal consistency as follows: 'For a memory to be considered causally consistent, it is necessary that the memory obey the following condition: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.' The rule for causal consistency is given in definition 8.

Definition 8 (causal consistency rule)
$\forall b \in D$, $\forall c, d \in Z(b)$, $c \neq d$:
$(e_1, e_2) \in \sqsubseteq$, $e_1 \in E_{R(b)c}$, $e_2 \in E_{W(b)d} \Rightarrow$
$\forall e_3 \in E_{R(b)c}$, $\forall e_4 \in E_{R(b)d} : (e_3, e_4) \in \sqsubseteq_{causal}$

Figure 3 shows two Hasse diagram clippings to illustrate the causal consistency rule. The precedence constraint resulting from $\sqsubseteq_{causal}$ is drawn with dotted lines. As there is a cycle in the left clipping it is not part of a causal consistent execution but there is no cycle in 3(b).

A slightly weaker consistency model is called *FIFO consistency* which is defined as follows [Tan95]: 'Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.' The FIFO consistency rule looks as follows:

Definition 9 (FIFO consistency rule)
$\forall b \in D$, $\forall c, d \in Z(b)$, $c \neq d$:
$(e_1, e_2) \in \sqsubseteq$, $e_1 \in E_{W(b)c}$, $e_2 \in E_{W(b)d} \Rightarrow$
$\forall e_3 \in E_{R(b)c}$, $\forall e_4 \in E_{R(b)d} : (e_3, e_4) \in \sqsubseteq_{FIFO}$

If the FIFO consistency rule is applied to figure 3(a) the Hasse diagram remains acyclic. This proves that FIFO consistency is weaker than causal consistency. We get the same result by looking at the rules

- causal-order (definition 4):
  $(e_1, e_2) \in \sqsubseteq_c$, $e_1 \in E_{W(b)c}$, $e_2 \in E_{R(b)c}$
- left side of causal consistency rule (definition 8):
  $(e_1, e_2) \in \sqsubseteq$, $e_1 \in E_{R(b)c}$, $e_2 \in E_{W(b)d}$

Both relations together imply the left side of the FIFO consistency rule in definition 9.

Finally we will define rules for the strongest possible consistency model that can be implemented without a global clock. [Tan95] defines *sequential consistency* as follows: 'The result of any execution is the same as if the (read and

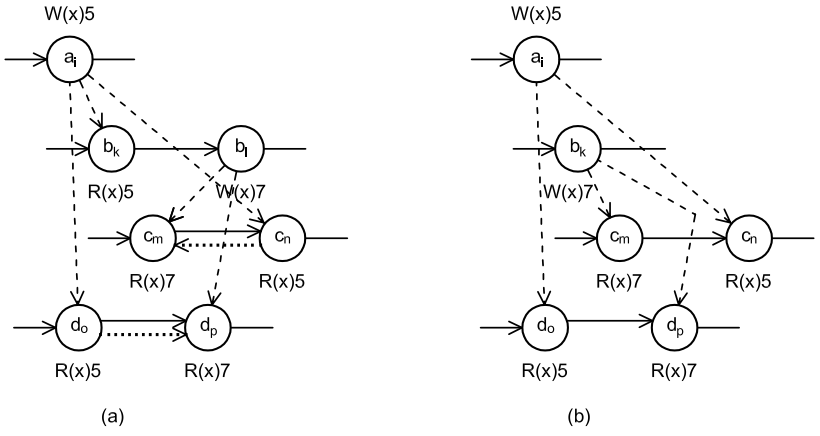

Figure 3: (a) Violation of causal consistency requirements; (b) clipping of a causal consistent execution;

write) operations by all processes on the data store were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.' In other words, an execution is sequentially consistent if there exists a total ordering $\sqsubseteq_t$ of all events which is an extension of the ordering $\sqsubseteq$. In addition $\sqsubseteq_t$ must be consistent with the causal-order $\sqsubseteq_c$ which means a write event $e \in W(x)c$ must appear before all read events $e_i \in R(x)c$ and no other read or write events $e_j \in W(x)d \cup R(x)d$ are allowed to appear between e and e_i.

Sequential consistency can be defined by the following three rules:

Definition 10 (sequential cons. rules)
$\forall b \in D$, $\forall c, d \in Z(b)$, $c \neq d$:

1. $(e_1, e_2) \in \sqsubseteq \cup \sqsubseteq_c$, $e_1 \in E_{W(b)c}$, $e_2 \in E_{R(b)d} \Rightarrow$ $\forall e_3 \in E_{R(b)c}, e_4 \in E_{W(b)d} : (e_3, e_4) \in \sqsubseteq_{seq}$
2. $(e_1, e_2) \in \sqsubseteq \cup \sqsubseteq_c$, $e_1 \in E_{W(b)c}$, $e_2 \in E_{W(b)d} \Rightarrow$ $\forall e_3 \in E_{R(b)c} : (e_3, e_2) \in \sqsubseteq_{seq}$
3. $(e_1, e_2) \notin \sqsubseteq \cup \sqsubseteq_c$, $e_1 \in E_{W(b)c}$, $e_2 \in E_{W(b)d} \Rightarrow$ $(e_1, e_2) \in \sqsubseteq_{seq} \vee (e_2, e_1) \in \sqsubseteq_{seq}$

For an execution to be sequentially consistent, the first two rules from definition 10 are not sufficient if the execution contains concurrent write events on the same data object and at least one read event that reads the value of one of the concurrent write events. This situation is shown in figure 4. Although rules 1 and 2 of Definition 10 have been applied to the event lattice clipping in figure 4 and the clipping does not contain a cycle, it is impossible to extend the partial ordering $\sqsubseteq \cup \sqsubseteq_c$ to a total ordering $\sqsubseteq_t$ of a sequential execution:

- If $(b_m, a_j) \in \sqsubseteq_t$ then $(a_l, a_j) \in \sqsubseteq_{seq}$ according to rule 2 in definition 10. This leads to the cycle $a_j \rightarrow a_k \rightarrow a_l \rightarrow a_j$. Therefore 1 must be written to object x before the 2 is written, i.e. $(a_j, b_m) \in \sqsubseteq_t$.
- The same is true for a_i and b_n. If $(a_i, b_n) \in \sqsubseteq_t$ then $(b_o, b_n) \in \sqsubseteq_{seq}$ according to rule 2 in definition 10. This leads to the cycle $b_n \rightarrow b_o \rightarrow b_n$. Therefore 3 must be written to object y before the 4 is written, i.e. $(b_n, a_i) \in \sqsubseteq_t$.
- $(b_n, a_i) \in \sqsubseteq_t$ and $(a_j, b_m) \in \sqsubseteq_t$ leads to the contradiction $a_i \rightarrow a_j \rightarrow b_m \rightarrow b_n \rightarrow a_i$.

Therefore we need a third rule to handle concurrent write events. Applying this rule to the example in figure 4 results in a cycle as there is no reasonable way to order the concurrent write events. Unfortunately the third rule does not have the form of a consistency rule as described in definition 6.

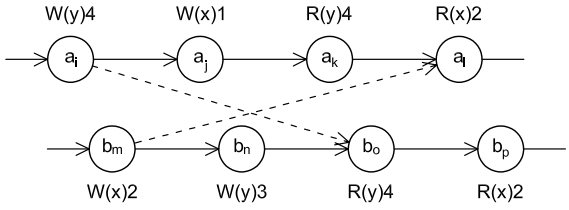

Figure 4: Not sequentially consistent execution

Finally we show that the sequential consistency rules in fact are equivalent to the definition given above. Assuming we have an execution $x \in ex((\mathcal{L}, \mathcal{I}))$ of an interpreted event lattice $(\mathcal{L}, \mathcal{I})$ that is consistent in regard to definition 10. We could extend $\sqsubseteq \cup \sqsubseteq_c \cup \sqsubseteq_{seq}$ to obtain a total ordering by iteratively selecting events according to $\sqsubseteq \cup \sqsubseteq_c \cup \sqsubseteq_{seq}$. If the choice is not unique we can randomly choose among the subset of events that are not preceded by other events that have not been selected yet. Finally we get a total ordering $\sqsubseteq_t$ of all events which is an extension of the ordering $\sqsubseteq \cup \sqsubseteq_c$. This means that all write events $e_i \in W(x)c$ appear before all read events $e_j \in R(x)c$. Due to the definition of the sequential consistency rules, no other read or write events $e_k \in W(x)d \cup R(x)d$ are allowed to appear between e_i and e_j. Therefore x is a sequentially consistent execution. The reverse is also true as if we take the total ordering $\sqsubseteq_t$ of a sequentially consistent execution x', the Hasse diagram augmented by the activity-order, sync-order, causal-order and the sequential consistency rules of definition 10 is always acyclic.

5 Summary

In this paper, we have presented a way to define consistency models using a model of event lattices. It turns out that a distributed execution is consistent if there is no opposition to a set of rules. So it is not adequate to talk about consistency alone, we always have to give a set of rules that must be met. In the case of data consistency the rules are order relations among read and write events. If a consistency rule is violated by an execution the Hasse diagram of the corresponding event lattice gets cyclic. We also have identified a minimal set of rules that must be observed in every execution.

Using three examples, the causal, FIFO and sequential consistency model, we have shown how to transform the informal definitions from literature into consistency rules.

References

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, October 1991.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[DR95] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[Reh03] Christian Rehn. Dynamic Global Scheduling in Cooperative Distributed Systems. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2003*, pages 1427–1433, Las Vegas, NV, June 2003.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice–Hall International, 1995.

[ZIS+97] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Singh, Kal Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 193–205, New York, June 18–21 1997. ACM Press.