In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'01), Santa Barbara, California, USA, May 2001.

# Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes

Heiko Schuldt

Database Research Group
Institute of Information Systems
Swiss Federal Institute of Technology (ETH)
CH–8092 Zürich, Switzerland
schuldt@inf.ethz.ch

## ABSTRACT

In this paper, we propose *process locking*, a dynamic scheduling protocol based on ideas of ordered shared locks, that allows for the correct concurrent and fault-tolerant execution of transactional processes. Transactional processes are well defined, complex structured collections of transactional services. The process structure comprises flow of control between single process steps and also considers alternatives for failure handling purposes. Moreover, the individual steps of a process may have different termination characteristics, i.e., they cannot be compensated once they have committed. All these constraints have to be taken into consideration when deciding how to interleave processes. However, due to the higher level semantics of processes, standard locking techniques based on shared and exclusive locks on data objects cannot be applied. Yet, process locking addresses both atomicity and isolation simultaneously at the appropriate level, the scheduling of processes, and accounts for the various constraints imposed by processes. In addition, process locking aims at providing a high degree of concurrency while, at the same time, minimizing execution costs. This is done by allowing cascading aborts for rather simple processes while this is prevented for complex, long-running processes within the same framework.

## 1. INTRODUCTION

Traditional locking techniques enforce write access to data objects to be exclusive. In combination with strict two phase protocols, this has strong impacts on the degree of concurrency. Locks with constrained sharing [1] relax this exclusiveness and allow locks to be shared between different transactions in dedicated orders, thus increasing the degree of parallelism. But, this gain goes along with the possibility of cascading aborts. Hence, since avoiding cascading aborts (ACA) [6] is considered to be an important feature to shield independent transactions, most database systems still exploit strict two phase locking protocols (S2PL) [11] with traditional lock compatibility modes.

While the S2PL approach is well suited in the flat transaction model based on read/write operations applied to rather short transactions, it generally reaches its limits when transactions become more complex, i.e., when their duration significantly exceed those of booking or funds transfer transactions, or when transactions are settled at a higher level of semantics, comprising semantically rich operations. Transactional processes [24] are an example of such complex transactions: they follow the philosophy of hyperdatabases [23] which are databases sitting on top of databases and which manage objects being composed of database objects or, as in the case of processes, manage transactions being composed of transactions. Processes integrate transactions provided by transactional systems, thereby bringing them into a higher level semantics. In particular, processes establish flow of control between the individual steps as one of their basic semantic elements. This even allows the specification of alternatives that can be executed in the case of failures; hence, it leads to more general properties than those of traditional ACID transactions. Finally, a process may reach a point-of-no-return, i.e., execute a step that cannot be compensated, already in the middle of execution, which requires that the subsequent commit of the process has to be enforced. All these different aspects need to be taken into consideration when deciding how to interleave processes. To this end, by extending and applying the unified theory of concurrency control and recovery [4, 28] to processes, we have identified correctness criteria that simultaneously account for isolation and atomicity [24].

Due to the black box semantics of the single steps of processes, locking techniques at data level cannot be applied but must be used at the level of individual transactions. Hence, also a distinction between read and write access is not possible such that transactions would have to be executed exclusively which, combined with S2PL, would imply considerable restrictions on the degree of concurrency. Yet, locks with constrained sharing allow to increase parallelism but may require the cascading abort of processes which is, in worst case, not possible when they have already passed a point-of-no-return. Hence, processes demand vital extensions of ordered shared locking such that cascading aborts are generally possible for certain processes (otherwise, the degree of concurrency would be impractically limited) while cascading aborts must be avoided for other processes, es-

| | **Execution cost** $c(a_i)$ | **Failure probability** $p(a_i)$ | **Compensation cost** $c(a_i^{-1})$ |
|---|---|---|---|
| $a_i^c$ **compensatable** | $0 < c(a_i^c) < \infty$ | $0 \leq p(a_i^c) < 1$ | $0 \leq c(a_i^{-1}) < \infty$ |
| $a_i^p$ **pivot** | $0 < c(a_i^p) < \infty$ | $0 \leq p(a_i^p) < 1$ | $c(a_i^{-1}) = \infty$ |
| $a_i^r$ **retriable** | $0 < c(a_i^r) < \infty$ | $p(a_i^r) = 0$ | $0 \leq c(a_i^{-1}) \leq \infty$ |
| $a_i^{-1}$ **compensating** | $0 \leq c(a_i^{-1}) < \infty$ | $p(a_i^{-1}) = 0$ | $c((a_i^{-1})^{-1}) = \infty$ |

**Table 1: Execution Costs and Failure Probabilities of Activities**

pecially for those which cannot be aborted but rather have to commit. These extensions can be found in *process locking*, a dynamic scheduling protocol addressing the correct execution of transactional processes.

The paper is organized as follows: In Section 2, we briefly introduce the process model and the correctness criteria for transactional processes. Section 3 develops the process locking protocol and Section 4 presents extensions of this basic protocol. In Section 5, we discuss related work. Section 6 concludes.

# 2. TRANSACTIONAL PROCESSES

We consider an architecture with two layers. The top layer controls the execution of *transactional processes*, as specified in *process programs*. Each one of these process programs is a set of partially ordered activities. Each activity, in turn, corresponds to a conventional transaction executed in a transactional application. Hence, activities are, by definition, atomic and therefore either terminate committing or aborting. The bottom layer of the system model is formed by the universe of all available independent transactional applications (subsystems). Each of these subsystems has to provide serializable (CPSR) executions and avoid cascading aborts (ACA) [6]. The concurrent execution of transactional processes is controlled by a *transactional process manager (PM)*, which is responsible for scheduling the invocation of transactions in the underlying applications.

The activity model, the process model, and the criteria that jointly account for correct concurrency control and recovery in transactional processes follow the ideas presented in [24]. In here, we briefly repeat the most important notions needed to develop a dynamic scheduling protocol for transactional processes.

## 2.1 Activity Model

Activities differ in terms of their termination properties. According to the flexible transaction model, we consider three cases: compensatable, retriable, and pivot [18, 30]. Let $\mathcal{A}^*$ be the set of all activities available in the system. A straightforward way to specify these termination properties is to identify the execution cost $c(a_i)$ with $c : \mathcal{A}^* \mapsto \mathbb{R}_0^+$ and the failure probability $p(a_i)$ with $p : \mathcal{A}^* \mapsto [0, 1)$ of each activity $a_i \in \mathcal{A}^*$, e.g., based on some heuristics on the associated subsystem transactions.

A *compensatable* activity $a_i^c$ has a compensating activity $a_i^{-1}$, namely a compensation transaction provided by the same system, which semantically undoes its effects. This is reflected by the compensation cost of $a_i$, i.e., the execution cost of $a_i^{-1}$, which yields a finite value. In some cases, compensation may be superfluous — similar to the inverse of a read operation in the traditional read/write model. Therefore, the cost of compensating activities may equal zero. Activities $a_i^p$ for which no compensating activity $a_i^{-1}$ exists, that is, activities which are not compensatable, are *pivot*. This is reflected by infinite costs of their compensation $a_i^{-1}$. Activities $a_i^r$ whose executions are guaranteed to successfully terminate after a finite number of invocations are called *retriable*. Hence, their failure probability equals zero, independent of their cost and the cost of their compensation. Compensating activities $a_i^{-1}$ finally must be retriable, thus guaranteed to succeed. Moreover, they are themselves not compensatable. The characteristics of activities are summarized in terms of execution costs and failure probabilities in Table 1 (the termination properties of activities are denoted by their superscript).

In contrast to the original flexible transaction model, retriability on the one hand and the availability of compensation on the other hand are orthogonal properties. An activity can have both, one of them, or neither.

## 2.2 Process Model

A process program specifies the execution of activities according to the results of past activities, thereby also allowing alternative executions for failure handling purposes. More formally,

DEFINITION 1 (PROCESS PROGRAM). *A* process program $PP = (\mathcal{A}, <, \lhd)$ *is a tuple where* $\mathcal{A} \subseteq \mathcal{A}^*$ *is a set of activities,* $<$ *is a partial order (precedence order) between these activities with* $< \subseteq (\mathcal{A} \times \mathcal{A})$*, and* $\lhd$ *is a partial order (preference order) defined over* $<$ *with* $\lhd \subseteq (< \times <)$ *establishing alternative executions.* □

A correct process program can be viewed as a tree whose nodes are activities and whose edges correspond to order constraints between these activities. The first non-compensatable activity on a path from the root in the tree is a *primary pivot* of the process. It is a no-return activity: if it commits, the process cannot rollback any more; it must be able to complete. Generally, a pivot may have a $\lhd$-ordered set of children where the last one consists only of retriable activities (assured termination tree), and all previous ones, called subprocess programs, have (recursively) the properties of a process program. After successfully executing a pivot, the process program may try different alternatives, given by $\lhd$, and only if they fail, execute the one whose termination is assured. Note that a process program may have no pivot, in which case it has the same properties as a regular transaction. If a program allows for concurrent execution of activities, we group a partially ordered set of activities as a single (multi-activity) node of the tree, rather than as singleton nodes. While the activities of such multi-activity nodes can be executed concurrently, they are $<$-ordered with respect the activities of preceding and subsequent nodes. Pivot activities are always represented as singleton nodes.

In analogy to the traditional terminology, the execution of a process program is termed *process*:

DEFINITION 2 (PROCESS).
*A* process $P_i^k = (\mathcal{A}_i, \prec_i)$ *is a tuple reflecting the execution of a process program* $PP_k$ *where* $\mathcal{A}_i$ *is a set of activities that contains (a subset of) the regular activities* $\mathfrak{A}_k$ *of* $PP_k$*, but may also contain compensating activities for some of them. Additionally,* $\mathcal{A}_i$ *may contain one of* $C_i$ *or* $A_i$*. Moreover, the* required order $\prec_i \subseteq (\mathcal{A}_i \times \mathcal{A}_i)$ *is the minimal partial order that contains* $<_k$ *and that relates regular activities and their compensation, that enforces the execution of compensating activities in reverse order of the corresponding regular activities, and that respects the preference order on alternatives imposed by* $\lhd_k$. □

If $\mathcal{A}_i$ contains one of the termination activities $C_i$ or $A_i$, respectively, then $P_i$ is said to be *complete.* A process that is either running or completing is called *active.* A process program execution is not a path in the tree. It may contain aborted activities and compensating activities for the process or for its subprocesses. Once instantiated, a process is in the state *running.* Prior to the commit of a primary pivot $p_i^*$, an abort changes the state to *aborting*, where compensating activities are executed in reverse order. After finally having compensated each activity (called abort process execution), a process is in the state *aborted.* The commit of a primary pivot $p_i^*$ causes a state change from *running* to *completing.* The program may now try several alternatives in the order given by $\lhd_k$. The failure, i.e., abort of an alternative, leads to an abort subprocess execution and causes it to try the next one. Finally, if an alternative completes, the process commits by $C_i$. When no pivot activity exists, a process can directly change its state from *running* to *committed.* When a completing process is in an assured termination tree, activities may abort, but then they are retried until they succeed. In [30], it has been shown that well-formed flex structures guarantee that one execution path exists that can be executed correctly while all other paths leave no effects. Process programs having tree structure where at least one child of each pivot activity is an assured termination tree also capture this property. We refer to such process programs as having *guaranteed termination.* Essentially, this property is a generalization of the "all-or-nothing" semantics of traditional ACID transactions [24].

The correct structure of process programs with respect to guaranteed termination —imposed by the termination characteristics of single process activities and the orders between them— nicely meets the semantics of practical applications. The distinction between compensatable steps followed by a pivot step as point-of-no-return (the commit decision) and subsequent retriable steps, the latter being arranged in two alternatives for successful or unsuccessful outcomes can be found, for instance, in electronic commerce payment processes [8, 25, 19].

We summarize the description of processes by comparing them to transactions: A conventional transaction may contain irreversible actions. However, these have to be deferred until the commit. Therefore, the essential termination property of a transaction is that it can be aborted at any time until a commit decision is reached. In contrast, a process may have to perform a non-compensatable activity in the middle of its execution; this activity cannot be deferred to its end. After this activity, the process can no longer abort. Consequently, part of the responsibility for the proper termination now lies on the process programs. While transaction programs do not deal with the issue of how to continue after an abort, process programs contain explicit alternatives for aborted subprocesses.

## 2.3 Process Schedules and Correctness

A process schedule $\mathcal{S}$ reflects the concurrent execution of process programs with guaranteed termination. Hence, it does not only include regular activities but also recovery related ones as they are considered in a process:

DEFINITION 3 (PROCESS SCHEDULE).
*A* process schedule $\mathcal{S}$ *is a quadruple* $(\mathcal{P}_\mathcal{S},\ \mathcal{A}_\mathcal{S},\ \prec_\mathcal{S},\ <_\mathcal{S})$ *where*

1. $\mathcal{P}_\mathcal{S}$ *is a set of processes* $P_i = (\mathcal{A}_i,\ \prec_i)$.
2. $\mathcal{A}_\mathcal{S}\ \subseteq\ \mathcal{A}^*$ *with* $\mathcal{A}_\mathcal{S} = \{a_{i_j} \mid (a_{i_j} \in \mathcal{A}_i) \wedge (P_i \in \mathcal{P}_\mathcal{S})\}$ *is the set of all activities of all processes of* $\mathcal{P}_\mathcal{S}$.
3. $\prec_\mathcal{S}$ *with* $\prec_\mathcal{S}\ \subseteq\ (\mathcal{A}_\mathcal{S} \times \mathcal{A}_\mathcal{S})$ *is a partial order between activities of* $\mathcal{A}_\mathcal{S}$*, called the* required order*, which is the union of the required orders of all processes of* $\mathcal{P}_\mathcal{S}$*, that is* $\prec_\mathcal{S} = \bigcup_{P_i \in \mathcal{P}_\mathcal{S}} \prec_i$.
4. $<_\mathcal{S}$ *with* $<_\mathcal{S}\ \subseteq\ (\mathcal{A}_\mathcal{S} \times \mathcal{A}_\mathcal{S})$ *is a partial order between activities of* $\mathcal{A}_\mathcal{S}$*, called* execution order*, reflecting the observed order in which activities are executed. The required order* $\prec_\mathcal{S}$ *of* $\mathcal{S}$ *is contained in the observed execution order* $<_\mathcal{S}$*, that is* $\prec_\mathcal{S}\ \subseteq\ <_\mathcal{S}$. □

If $\mathcal{S}$ comprises active processes, then it is said to be *partial*, otherwise $\mathcal{S}$ is called *complete.*

Note that since a process schedule is defined at the level of activities, it also includes committed activities of aborted processes. However, since the underlying subsystems guarantee both serializability (CPSR) and atomicity (ACA), activities returning with abort can be omitted in $\mathcal{S}$. In particular, retriable activities appear at most once in a process schedule, namely when they are committed.

All processes are considered to be independent. Thus, the only possibility for some flow of information between concurrent processes is when conflicting activities share some resources in the underlying subsystem. A common mechanism to verify whether there is flow of information between arbitrary activities is *commutativity.* Following [28], the notion of commutativity is defined using return values: two activities, $a_{i_k}, a_{j_l} \in \mathcal{A}^*$ commute, if for all activity sequences $\alpha$ and $\omega$ from $\mathcal{A}^*$ the return values of all activities in the activity sequence $\langle \alpha\ a_{i_k}\ a_{j_l}\ \omega \rangle$ are identical to the return values of the sequence $\langle \alpha\ a_{j_l}\ a_{i_k}\ \omega \rangle$. Conversely, two activities are in conflict if they do not commute. Due to the kind of system we consider, activities may only conflict if they are executed in the same subsystem. In practical applications, a common assumption is that commutativity is perfect. This is the case when, for all pairs of activities $a_i, a_j \in \mathcal{A}^*$, either all possible combinations $(a_i^\alpha, a_j^\beta)$ for $\alpha, \beta \in \{-1, 1\}$ commute, or all possible combinations conflict [28]. Information about commutativity is crucial and must be available to the process manager. Since each subsystem provides CPSR executions, all conflicting activities are ordered in a process schedule.

The formalism we use to derive correctness criteria for process schedules is based on the unified theory of concurrency control and recovery [4, 28], hence addresses both atomicity and isolation jointly, but extends and generalizes the unified theory so as to account for the special semantics and structure of processes [24].

Definition 4 (Reducible Process Schedule). *A process schedule $\mathcal{S} = (\mathcal{P_S}, \mathcal{A_S}, \prec_\mathcal{S}, <_\mathcal{S})$ is* reducible (RED) *if it can be transformed to a serial process schedule $\underline{\mathcal{S}} = (\mathcal{P}_{\underline{\mathcal{S}}}, \mathcal{A}_{\underline{\mathcal{S}}}, \prec_{\underline{\mathcal{S}}}, <_{\underline{\mathcal{S}}})$ by applying the following two transformation rules finitely many times:*

1. Commutativity Rule: *The order $a_{i_k} <_\mathcal{S} a_{j_l}$ of two activities $a_{i_k}, a_{j_l} \in \mathcal{A_S}$ can be replaced by $a_{j_l} <_{\underline{\mathcal{S}}} a_{i_k}$ if the following conditions hold:*
   (a) *Either $a_{i_k}$ and $a_{j_l}$ commute and belong to different processes $(i \neq j)$, or they are from the same process $(i = j)$ and are not ordered in $\prec_\mathcal{S}$, i.e., their process program allows parallel execution.*
   (b) *There is no $a_{q_t} \in \mathcal{A_S}$ with $a_{i_k} <_\mathcal{S} a_{q_t} <_\mathcal{S} a_{j_l}$.*
2. Compensation Rule: *If two activities $a_{i_k}, a_{i_k}^{-1} \in \mathcal{A_S}$ such that $a_{i_k} <_\mathcal{S} a_{i_k}^{-1}$ and there is no activity $a_{q_t} \in \mathcal{A_S}$ with $a_{i_k} <_\mathcal{S} a_{q_t} <_\mathcal{S} a_{i_k}^{-1}$, then $a_{i_k}$ and $a_{i_k}^{-1}$ can both be removed from $\underline{\mathcal{S}}$.* □

The above definition does not require a process schedule to be complete and thus features a major difference compared with the original unified theory where, prior to the application of reduction techniques, the expansion of a schedule is required. Expansion leads to a a complete schedule where each running transaction is aborted and where all recovery-related operations are made explicit. Yet, reduction for process schedules does not require expansion, since these schedules already consider abort related activities.

Definition 5 (Prefix–Reducibility). *A process schedule $\mathcal{S}$ is* prefix–reducible (P-RED) *if each prefix of $\mathcal{S}$ is RED.* □

When scheduling processes dynamically, each prefix of a process schedule has to be reducible. In addition to the previous ideas considering non-complete process schedules $\mathcal{S}$, all completing processes must be able to commit while all aborting ones must abort correctly such that they do not leave any effects. Thus, all active processes have to terminate so as to ensure that all compensating activities are taken into account in the reduction phase. Therefore, reduction techniques have to be applied to the completed process schedule $\mathcal{C}(\mathcal{S})$ of $\mathcal{S}$, leading to the notion of *correct termination (CT)*:

Definition 6 (Correct Termination). *A complete process schedule $\mathcal{C}(\mathcal{S})$ has* correct termination (CT) *property if it is prefix reducible (P-RED).* □

In the original unified theory, a criterion (SOT, serializable with ordered termination) has been introduced that allows to reason about concurrency control and recovery without expanding a schedule [4]. However, a similar criterion does not exist in the case of processes. The reason being is that in the traditional transaction model, all recovery-related operations are known beforehand, i.e., the inverses of all regular operations. When, in addition, commutativity is perfect, the also the commutativity behavior of all these operations is known. In transactional process management, however, the completion of a process schedule $\mathcal{S}$ introduces new activities which are not related to the ones that have already been committed. By this, additional pairs of conflicting activities may be present in $\mathcal{C}(\mathcal{S})$. Hence, relying only on information of a partial process schedule $\mathcal{S}$ is not sufficient for reasoning about correct concurrency control and recovery.

While CT guarantees that all aborts are performed correctly in a completed process schedule, a process manager deciding dynamically on the execution of activities has additionally to make sure that each arbitrary subset of all active (sub-)processes of a partial process schedule can be aborted correctly. Since this might induce cascading aborts, it must be enforced that no completing process depends on a running (sub-)process in the sense that the abort of the latter implies the abort of a completing process. This leads to *process-recoverability*, a generalization of the traditional notion of recoverability:

Definition 7 (Process-Recoverability). *A process schedule $\mathcal{S} = (\mathcal{P_S}, \mathcal{A_S}, \prec_\mathcal{S}, <_\mathcal{S})$ is* process-recoverable (P-RC)*, if for each pair of conflicting activities $a_{i_k}^c$ and $a_{j_m}$ of $\mathcal{S}$ with $a_{i_k}^c <_\mathcal{S} a_{j_m}$ where $a_{i_k}^c$ is compensatable, where $a_{i_k}^{-1} \not<_\mathcal{S} a_{j_m}$ and $a_i^* \not<_\mathcal{S} a_{j_m}$ (with $a_i^*$, we denote the next point-of-no-return succeeding $a_{i_k}^c$ with respect to $\prec_i$; this may either be the commit $C_i$ or a pivot activity), the following holds:*

1. *If $a_{j_m}$ is compensatable and when $a_j^*$ is in $\mathcal{S}$, then the following order has to exist in $\mathcal{S}$: $a_i^* <_\mathcal{S} a_j^*$ where $a_j^*$ is the next point-of-no-return succeeding $a_{j_m}^c$ with respect to $\prec_j$ (again, this may be $C_j$ or a pivot of $P_j$).*
2. *If $a_{j_m}$ is not compensatable, then the following order has to exist in $\mathcal{S}$: $a_i^* <_\mathcal{S} a_{j_m}$.* □

When no pivot activities exist as in the traditional case, then, according to Definition 7.1, only the order $C_i <_\mathcal{S} C_j$ is imposed. But the special semantics of pivot activities is also reflected in this definition: pivot activities are treated in a similar way than the commit of a process since, by successfully committing a pivot, all preceding compensatable activities can no longer be compensated.

# 3. PROCESS LOCKING

In this section, we present *process locking*, a dynamic scheduling protocol that enforces CT process schedules and which also guarantees that each prefix of a complete process schedule is P-RC. We introduce the basic ideas of process locking and develop the protocol in detail.

## 3.1 Introduction to Process Locking

Process locking aims at providing a dynamic scheduling protocol that supports P-RED and P-RC multi-process executions and which guarantees the correct termination (CT) of each partial process schedule. It allows a process manager to dynamically decide on the execution, deferment, and rejection of activities. Such a process manager relying on the process locking protocol has been implemented as part of the WISE system [2, 16] which, in turn, is based on the process support engine OPERA [3, 14].

Process locking makes use of some basic assumptions on the process programs to be executed and on the commutativity behavior. Each process program has to be inherently correct, i.e., it has to follow the guaranteed termination property. Additionally, commutativity is assumed to be perfect.

A process manager has to enforce CT, even for partial process schedules. Hence, it has to guarantee that all partial processes can be completed correctly. In general, since the (correct) structure of all process programs is known to the process manager, execution orders, especially in terms of the completed process schedule $\mathcal{C}(\mathcal{S})$ of some partial process schedule $\mathcal{S}$, could be determined beforehand by exploiting information on the future activities of processes. However, process programs may contain execution paths that are not followed (e.g., due to some decision choosing one subtree but skipping others, or since alternative executions need not be executed). Therefore, the a priori determination of multi-process executions would have to consider more activities than are actually executed and thus, would be very restrictive in nature. Additionally, it requires the complex analysis of the transitive closure of the commutativity behavior of all future activities. This is even the case when completion is restricted to the safe alternatives (assured termination trees) of completing processes. Moreover, the predetermination of execution paths would also neglect the complexity of activities, i.e., their execution time, and would also prevent the support for dynamic changes of processes and process programs [20].

Another strategy to avoid unresolvable situations, i.e., cyclic conflicts in which two or more completing processes are involved (note that this could not be resolved by the abort of any of these processes), is to allow at most one completing process at any point in time. Yet, although limiting concurrency, a dynamic scheduling protocol following this restriction needs only to consider activities that are actually executed and does not have to take into account all potential future activities of active processes. Hence, process locking applies this strategy to enforce the correct completion of partial process schedules. This distinguished completing process then has a special status and will be preferred against all other processes, much like the "golden transaction" of the System R database [12], the only transaction of the system that is allowed to perform undo operations at a time.

## 3.2 Process Locking: The Core Protocol

In short, the process locking protocol is based on and extends ideas of locks with constrained sharing [1] and timestamp ordering [27, 6]. In what follows, we will motivate the necessity of advanced mechanisms for supporting CT and present the protocol in detail.

### *3.2.1 Locks with Constrained Sharing*

Most concurrency control protocols apply locking techniques to control concurrent access to shared resources. In the traditional read/write model, the semantics of data access is exploited which allows to share locks for read operations which do not change any database state, but requires exclusive access when data is written.

These locking techniques associate locks with single data objects. However, activities of processes correspond to transactions which are executed in some subsystem. In general, these transactions and the objects accessed by them are not known to the process manager and rather appear as black boxes. For these reasons, conventional locking techniques at data level cannot be applied to transactional processes. Hence, despite of the black box characteristics of the actual implementation of activities, the process manager nevertheless exploits information on their commutativity behavior (by means of a commutativity relation) which allows to dynamically identify pairs of conflicting activities. Yet, in contrast to traditional approaches, process locking associates locks with activity types to indicate whether or not an activity can be invoked in the context of a process schedule.

But, activities of transactional processes cannot be qualified as read or write. They are, in general, located at a semantical higher level of abstraction. As a result, the difference between shared access and exclusive access blurs and locks on activities would inevitably have to be exclusive. When combined with two phase locking or even strict two phase locking [11], this would unnecessarily reduce the degree of concurrency, especially since the execution of single activities and thus also those of processes might be very long.

In order to avoid (nearly) serial executions of processes, sharing of locks should nevertheless be allowed. To this end, the ideas of locks with constrained sharing [1] can be applied to process activities. Aside of shared and exclusive locks, a third category, namely *ordered shared locks (OSL)*, is considered. Several relaxations of the standard lock compatibility are proposed, the most permissive of them being the case where only shared locks (for concurrent read access to data) and ordered shared locks (for all other concurrent accesses) are exploited. With each pair of shared locks, an order is associated which has to be respected for the execution of the corresponding operations, when acquiring further locks, and when locks are relinquished. A lock $l_i$ of a transaction $T_i$ is said to be *on hold* if $l_i$ was acquired after another transaction $T_j$ has acquired a lock $l_j$ on the same data object but before $l_j$ has been released. The lock *relinquish rule* guarantees that all locks are shared with the same order in that a transaction may not release a lock as long as any of its locks is on hold. In process locking, the ideas of ordered shared locks are now combined with the special semantics that can be found in processes, namely locking at the level of activities.

The prerequisite for the application of locking techniques at activity level is that a complete commutativity relation is available to the process manager. This relation is implemented by an $(n \times n)$ matrix $CON$, with $n$ being the number of all activities in $\mathcal{A}^*$, where $CON(a_i, a_j) = TRUE$ when $a_i$ and $a_j$ conflict, and $CON(a_i, a_j) = FALSE$ otherwise. This matrix indicates conflicts on a rather coarse granularity, on the level of activity types, i.e., for different transaction programs that can be executed in the underlying subsystems, but does not consider parameters associated with these invocations. However, this is the most general possibility that accounts for the black box semantics of activities which, due the lack of detailed information about their implementation and their structure, does in certain cases not allow to consider conflicts on a more fine-grained level.

In the previous section, we have seen that the correct interleavings of processes are governed by the conflict behavior

| held \ acquired | C Lock | P Lock |
|---|---|---|
| C Lock | $\Rightarrow$ | $\not\Leftrightarrow$ |
| P Lock | $\Rightarrow$ | $\not\Leftrightarrow$ |

**Table 2: Compatibility Matrix of C and P Locks ($\Rightarrow$: Ordered Shared; $\not\Leftrightarrow$: Exclusive)**

of activities (CT) and their termination properties (P-RC), i.e., whether or not they can be compensated. Hence, applied to process schedules, ordered shared locks at activity level provide a straightforward means to map allowed interleavings of processes into a compatibility matrix of different lock types. Similar to the usage of the read/write characteristics of operations in traditional locking protocols, the semantics of activities with respect to their termination characteristics (compensatable or pivot) can be exploited.

Therefore, *C locks* for compensatable activities and *P locks* for pivot activities, respectively, are used. Two C locks of different processes as well as a P lock followed by a C lock can be ordered shared. A C lock ordered shared with a subsequent P lock would correspond to a violation of P-RC (the process having acquired the C lock cannot be aborted correctly since the other process which would have to be aborted cascadingly is already completing). Hence, a C lock followed by a P lock cannot be ordered shared but must be exclusive. Finally, the combination of two P locks has also to be exclusive so as to avoid that two completing processes exist at the same time. The lock compatibility matrix of the process locking protocol is depicted in Table 2 where $\Rightarrow$ denotes ordered shared mode and $\not\Leftrightarrow$ stands for non-shared (exclusive) mode.

### 3.2.2 *Timestamp Ordering*

The original OSL protocol has an optimistic character since the compliance of orders is not checked, due to the lock relinquish rule, until the first lock is to be released which actually coincides with the commit decision [1]. This, in turn, implies that violations of the order constraints associated with shared locks are detected at a very late stage and even worse, may occur in situations where appropriate corrective strategies, i.e., the abort of the processes involved, are not possible since these processes are completing, not running.

To circumvent this drawback, we impose early verification of the correct order of shared locks that immediately takes place whenever locks are acquired. To do this, we adopt and apply ideas borrowed from timestamp ordering (TO) protocols [27, 6]. We use the same mechanisms to control the order in which ordered shared locks are acquired than the original TO protocol does for an a priori determination of the serialization order and thus, the order in which shared data objects are accessed. The only prerequisite is that each process is assigned a unique timestamp taken from a strictly monotonically increasing series.

### 3.2.3 *Process Locking: Combining OSL & TO for Processes*

Following the previous discussion, the application of ordered shared locks and timestamp ordering in the context of transactional processes requires that for each activity, i.e., for each transaction program that can be invoked by the process manager, additional information in the form of an ordered list is maintained which comprises the locks held for all invocations of that activity. Each lock, in turn, refers to the process by which the lock has been acquired (and by which the corresponding activity is invoked), thereby implicitly associating each lock with the timestamp of the corresponding process.

Even when combining the extended OSL protocol based on P and C locks with timestamp ordered lock requests, special treatment is necessary for pivot activities. Due to their commit-like semantics, they make compensation unavailable for all preceding activities and lead to state changes (of the process itself or of subprocesses). Recalling the criterion of P-RC, pivot activities must not not executed when the corresponding process has locks on hold. This is captured in process locking by the requirement that all preceding C locks held for compensatable activities have to be converted to P locks before a pivot activity can be executed, thereby reducing the problem to the allowed/disallowed sharing of locks according to Table 2.

The process locking protocol can be briefly summarized as follows: When instantiated, a process $P_i$ is assigned a unique timestamp $ts(P_i)$. Before an activity $a_{i_k}$ is to be executed by the process manager, a lock must be acquired which has to meet the termination property of $a_{i_k}$ (either a C lock or a P lock). This lock then corresponds to an entry in the lock list of the activity. Hence, the process manager exploits not only information about the commutativity of activities but also about their termination properties. However, prior to the permission of a lock, all conflicting activities, and in particular all locks held for these activities have to be analyzed so as to decide whether or not the lock for $a_{i_k}$ can be granted. The following six rules specify the acquisition and the release of locks, respectively, and define process locking in detail:

**Comp–Rule:** For the execution of a compensatable activity $a^c_{i_k}$, a C lock is required. Depending on the process timestamp of $P_i$ and the timestamps of potential other processes holding locks for conflicting activities, a C lock request can either be granted immediately, requires the abort of concurrent processes, or has to be deferred.

*Granting C Locks:* A C lock for some activity $a^c_{i_k}$ of a running process can be granted when either no other process holds a lock for a conflicting activity, or when all locks held for conflicting activities (either C or P locks) are from older processes with respect to the process timestamp. Once the C lock has been successfully acquired, $a^c_{i_k}$ can be executed.

*Aborting Concurrent Processes:* If a process $P_j$ with a younger timestamp, $ts(P_j) > ts(P_i)$, holds a C lock for a conflicting activity $a_{j_l}$, then $P_j$ will be aborted. If $P_j$ is already aborting, then $P_i$ has to wait until $P_j$ is aborted (aborting processes cannot be aborted). Once $P_j$ is aborted correctly, its locks are released, the C lock required for the execution of $a^c_{i_k}$ can be acquired, and $a^c_{i_k}$ can be executed. After completing the abort of $P_j$, it is resubmitted with the same timestamp in order to avoid its starvation. This is possible since $P_i$ is able to execute $a^c_{i_k}$ in the meanwhile such that, when $P_j$ redoes the execution of $a_{j_l}$, the constraints imposed by the process timestamps on the sharing of locks and thus, on the associated C locks, are met.

Additionally, the request of a C lock by a completing process leads to the abort of older processes already holding a C lock for a conflicting activity since completing processes are treated as “first-class processes” and are favored compared to running processes.

*Deferment of C Lock Requests:* If a younger process $P_k$, $ts(P_k) > ts(P_i)$, exists which already holds a P lock for a conflicting activity, then $a^c_{i_k}$ has to be deferred (since $P_k$ cannot be aborted) until the commit of $P_k$. Special treatment is also applied if a completing process $P_k$ with a younger timestamp holds a C lock (this is possible since we allow a pivot activity of a process program to be recursively followed by process programs). Then, $P_i$ has also to be deferred until the commit of the completing process $P_k$. The latter ones are the only cases where the lock sharing order (and thus, the serialization order) and the timestamp order do not coincide.

**Piv–Rule:** When $a^p_{i_k}$ is pivot, then $P_i$ has to acquire a P lock before it can be executed. But, prior to this P lock request, all previously held C locks of $P_i$ have to be converted to P locks so as to guarantee that P-RC will not be violated. Again, a distinction on whether the P lock can be granted immediately after lock conversion, whether it requires the abort of concurrent processes, or whether it has to be deferred is possible.

*Granting P Locks:* A P lock is granted, after lock conversion, if no other process holds a lock for a conflicting activity.

*Aborting Concurrent Processes:* In case younger processes $P_j$, $ts(P_j) > ts(P_i)$, hold C locks for conflicting activities, all these $P_j$ have to be aborted if they are running, otherwise, if they are already aborting, $P_i$ has to wait until they are aborted. After $A_j$, they are resubmitted with the same timestamp so as to avoid starvation.

*Deferment of P Lock Requests:* If older processes hold C locks or if any other process holds a P lock, then the request has to be deferred until the these processes have terminated. This is the case since, according to the lock compatibility matrix, a newly acquired P lock may not be shared with any other lock already held (and since at most one completing process at a time is allowed).

**Comp→Piv–Rule:** This conversion is required for all C locks of a process $P_i$ as prerequisite for the execution of a pivot activity $a^p_{i_k}$. Since the conversion of a C lock to a P lock is similar to the acquisition of a P lock, the same conditions hold: C→P lock conversion succeeds when either no other process holds a lock for a conflicting activity or if all existing locks are C locks held by younger processes $P_j$, $ts(P_j) > ts(P_i)$, which then have to be aborted (and which are resubmitted with the same process timestamp). In case older processes hold C locks or if any other process holds a P lock, then C→P lock conversion has to be deferred until the end of these processes.

**C$^{-1}$–Rule:** Prior to compensating $a^c_{i_k}$, a C lock has to be acquired for $a^{-1}_{i_k}$. Hence, according to the Comp–Rule, all processes $P_j$ which share locks with $P_i$ but have younger timestamps, $ts(P_i) < ts(P_j)$, are aborted (i.e., when these $P_j$ have executed some $a_{j_l}$ that conflicts with $a^c_{i_k}$ and appears after $a^c_{i_k}$ in $<_S$). All older processes $P_k$ having common locks with $P_i$ are not affected by the C lock request for $a^{-1}_{i_k}$.

**Abort–Rule:** The abort $A_i$ of a process $P_i$ leads to the release of all locks held by $P_i$.

**Commit–Rule:** A process $P_i$ is only allowed to commit if it does not share any locks with older processes $P_j$. At commit time, all locks of $P_i$ are released; hence, process locking follows the strict two phase locking paradigm [11]. Otherwise, if $P_i$ holds locks shared with older processes $P_k$, $ts(P_k) < ts(P_i)$, $C_i$ is deferred until all these $P_k$ have committed.

Note that, although compensating activities are themselves required to be pivot, we do not demand them to acquire P locks since this would require C→P lock conversion. Yet, it is sufficient for guaranteeing CT to abort only processes which have executed activities between a regular and a compensating activity which is already captured by requiring C locks for compensation.

Obviously, by allowing to share locks in timestamp order, a process may induce cascading aborts. Avoiding cascading aborts would be far too restrictive since it would, in the case of semantical rich activities where a distinction between read and write access to data is impossible, degenerate to rigorousness [7]. However, due to the exclusive treatment of certain combinations of locks, it is ensured that cascading aborts are restricted to running processes, not to completing ones. After the cascading abort of some process $P_j$ is completed, it is resubmitted with the same timestamp in order to avoid starvation. Additionally, process locking makes use of timestamp-based deadlock prevention strategies [21, 6] which, together with the restriction to at most one completing process at a time, guarantees the absence of deadlocks imposed by cyclic wait-for dependencies.

The proofs showing the correctness of process locking with respect to CT and P-RC can be found in the appendix.

# 4. COST-BASED SCHEDULING OF PROCESSES

In process locking, the abort of a process $P_i$ is induced in one of the following cases: i.) due to the failure of an activity when $P_i$ is running, ii.) due to the abort of some $P_j$ which shares locks with $P_i$ where locks of $P_j$ precede those of $P_i$, iii.) due to the violation of timestamp orders when an older, conflicting process $P_j$ issues some lock request, or iv.) due to a conflict of $P_i$ with a completing process $P_j$, independent of timestamp orders.

While the first case is inherent to the guaranteed termination property of single processes, the other cases, however, are based on the optimistic character of process locking that allows certain locks to be ordered shared and on the special treatment of completing, “first-class” processes. Hence, ii.) - iv.) stem from the presence of cascading aborts and may be subject to a refinement of process locking. We have previously shown that the total absence of cascading aborts would impose too strict limitations on the degree of concurrency. But the protocol could be tightened for certain, distinguished processes such that cascading aborts could not affect them although they are running while the possibility of cascading aborts should still be possible for other processes for which this is rather tolerable. Consider, for instance, a process $P_i$ that contains a complex and expensive activity $a_{i_k}$ for which a compensating activity $a_{i_k}^{-1}$ exists (or, similarly, a simple activity $a_{i_k}$ where $a_{i_k}^{-1}$ is expensive). Due to the presence of compensation, $a_{i_k}$ would be treated just like any other compensatable activity and may be subject to compensation when some other process aborts. In order to avoid situations where complex activities have to be compensated, we seamlessly extend the basic process locking protocol, leading to the *cost-based process scheduling* protocol, that allows a process manager to exploit information on the execution cost associated with each activity. Most importantly, this extension allows to refine the degree of concurrency on a per-process basis by allowing cascading aborts for some, rather simple processes while enforcing ACA for "valuable" processes, containing complex and expensive activities.

In short, the basic idea of cost-based process scheduling is to assign a cost threshold to each process program $PP_j$, to accumulate the cost actually effected by a process $P_i^j$ in a process schedule $\mathcal{S}$, and to apply special treatment to $P_i^j$ — similar to the privileges deployed for completing processes— once its accumulated cost exceeds the threshold specified for its process program. Hence, the process manager has to keep track of the cost accumulated by each active process $P_i^j$ in a process schedule $\mathcal{S}$. In order to account also for the cost of aborting $P_i^j$, this should not only consider the execution costs of regular activities but also those of the associated compensating activities, even if $P_i^j$ is not aborting in $\mathcal{S}$. The reason being is that these additional costs will incur in worst case, when $P_i^j$ is aborted. This leads to the notion of *worst-case cost*, $Wcc(P_i^j, \mathcal{S})$, of a process $P_i^j$ in a process schedule $\mathcal{S}$ (where $\mathcal{A}_i^{\mathrm{Reg}}$ is the set of all regular activities of $P_i^j$ in $\mathcal{S}$):

$$Wcc(P_i^j, \mathcal{S}) \;=\; \sum_{a_{i_k} \in \mathcal{A}_i^{\mathrm{Reg}}} \big(\; c(a_{i_k}) + c(a_{i_k}^{-1}) \;\big) \qquad (1)$$

Obviously, the restriction to the regular activities of a process schedule $\mathcal{S}$ when accumulating the worst-case cost $Wcc(P_i^j, \mathcal{S})$ of some process $P_i^j$ in $\mathcal{S}$ stems from the fact that the execution cost of the inverse $a_{i_k}^{-1}$ of each activity $a_{i_k} \in \mathcal{A}_{\mathcal{S}}^{\mathrm{Reg}}$ is already considered in (1). Thus, the notion of worst-case cost exceeds the actual cost that is caused by some process $P_i^j$ but encompasses additionally the execution cost given that $P_i^j$ would change its state to aborting and successfully execute all abort-related activities until it is finally aborted. In addition to the worst-case cost of a process gathered dynamically at run-time, a finite *cost threshold*, $Wcc^*(PP_j)$, has to be defined for each process program $PP_j$ which then accounts for all associated processes $P_i^j$.

When a regular activity $a_{i_k}$ is to be executed in a state characterized by a process schedule $\mathcal{S}$, the worst-case cost of $P_i^j$ has to be adapted *prior* to its invocation such that

$$Wcc(P_i^j, \mathcal{S}') \;=\; Wcc(P_i^j, \mathcal{S}) \;+\; c(a_{i_k}) \;+\; c(a_{i_k}^{-1}) \qquad (2)$$

with $\mathcal{A}_{\mathcal{S}'} = (\mathcal{A}_{\mathcal{S}} \cup a_{i_k})$. Based on $Wcc(P_i^j, \mathcal{S}')$, the process manager can now decide on the treatment of $a_{i_k}$: if the worst-case cost is below the cost threshold as defined in the process program, $a_{i_k}$ can be treated as compensatable, thus deploying the Comp–Rule for lock acquisition. However, when $Wcc(P_i^j, \mathcal{S}')$ exceeds $Wcc^*(PP_j)$, the process manager will treat $a_{i_k}$ as pivot, thereby enforcing that it is not compensated due to the failure of some other process. Compensating activities are still treated according to the $\mathrm{C}^{-1}$–Rule. The algorithm allowing cost-based process scheduling, thereby extending the process locking protocol, is presented in detail in Figure 1.

In order to provide correct process schedules, the cost-based extension of process locking has to identify pivot activities by the worst-case cost of their process and thus, to apply the Piv–Rule for lock acquisition. To this end, $Wcc(P_i^j, \mathcal{S})$ must, in any case, exceed the cost threshold defined in $PP_j$ whenever a process $P_i^j$ changes its state from running to completing.

LEMMA 1 (WORST-CASE COST & PIVOT ACTIVITIES). *The worst-case cost $Wcc(P_i^j, \mathcal{S})$ of $P_i^j$, determined for the execution of a pivot activity $a_{i_k}$ in a process schedule $\mathcal{S}$, exceeds the cost threshold $Wcc^*(PP_j)$.* □

PROOF (LEMMA 1).
*For each pivot activity $a_{i_k}^p$, the cost of compensation, i.e., the execution cost of $a_{i_k}^{-1}$, is infinite. Therefore, when the worst-case cost $Wcc(P_i^j, \mathcal{S})$ of a process $P_i^j$ is updated, according to (2), due to the execution of $a_{i_k}^p$, $Wcc(P_i^j, \mathcal{S}')$ with $\mathcal{A}'_{\mathcal{S}} = \mathcal{A}_{\mathcal{S}} \cup a_{i_k}^p$ will be infinite. The reason being is that $Wcc(P_i^j, \mathcal{S}')$ considers the execution cost of both $a_{i_k}$ and of its compensation. Hence, since $c(a_{i_k}^{-1}) = \infty$, $Wcc(P_i^j, \mathcal{S}')$ evaluates to: $Wcc(P_i^j, \mathcal{S}') = Wcc(P_i^j, \mathcal{S}) + c(a_{i_k}) + c(a_{i_k}^{-1})$ and contains at least one infinite addend. The cost threshold defined for a process program is, per definition, a finite value. Therefore, the worst-case cost of a process in which a pivot activity is to be scheduled for execution, and in particular when a process changes its state from running to completing, always exceeds its cost threshold, independently of $PP_j$ and of $Wcc^*(PP_j)$.* □

Aside of pivot activities, the Piv–Rule for lock acquisition is also applied to activities whose worst-case cost exceeds the cost threshold although these activities may be compensatable. In particular, activities that are treated like pivots but are actually compensatable and belong to a running process are called *pseudo pivots*. For pseudo pivot activities $a_{i_k}$, the following holds (with $\mathcal{A}'_{\mathcal{S}} = \mathcal{A}_{\mathcal{S}} \cup a_{i_k}$):

$$Wcc(P_i^j, \mathcal{S}) < Wcc^*(PP_j) \quad \wedge \quad Wcc(P_i^j, \mathcal{S}') \geq Wcc^*(PP_j)$$
$$\wedge \quad Wcc(P_i^j, \mathcal{S}') < \infty \qquad (3)$$

In particular, and conversely to Lemma 1, a running process $P_i^j$ can always be characterized by finite worst-case cost, hence $Wcc(P_i^j, \mathcal{S}') < \infty$ distinguishes pseudo pivots from the primary pivot of $P_i$.

Cost-based process scheduling now allows to circumvent the abort of a running process $P_i^j$ having executed a pseudo pivot activity (i.e., an activity which is either expensive or having an expensive inverse) due to the abort of some

```
initiate_process(Process Proc, ProcessProgram PP)
begin
   Wcc(Proc) := 0;
   ts(Proc) := assign_timestamp();
   execute_process(Proc, PP);
end  /* initiate_process */


execute_activity(Activity act, Process Proc, ProcessProgram PP)
begin
   if ( act is a compensating activity )  then
      request_C_lock(act);                          /* apply C^-1-Rule          */
      invoke(act);
   else
      comp := get_compensating_activity(act);
      Wcc(Proc) := Wcc(Proc) + c(act) + c(comp);     /* update worst-case cost   */
      if ( Wcc(Proc) < Wcc_max(PP) )  then          /* act is compensatable     */
         request_C_lock(act);                       /* apply Comp-Rule          */
         invoke(act);
      else                                          /* treat act like a pivot   */
         foreach a in Proc do                       /* check all activities and */
            convert_C_to_P_lock(a);                 /* apply Comp→Piv-Rule      */
         od
         request_P_lock(act);                       /* apply Piv-Rule           */
         invoke(act);
      fi
   fi
end  /* execute_activity */
```

**Figure 1: Algorithm for Dynamic Pivot Determination in Cost-Based Process Scheduling**

other process, but without generally requiring the avoidance of cascading aborts for all processes. Hence, this protocol allows to provide the full spectrum of process schedules between ACA and P-RC, i.e., cascading aborts for all running processes, by specifying the cost thresholds of process programs appropriately. Depending on that, a higher parallelism than ACA is possible while, at the same time, cascades are avoided on a per-process basis; thus, the cost-based extensions might be more restrictive than pure process locking.

## 5. RELATED WORK

In the context of transactional workflows, a couple of approaches exist that aim at enriching processes by transactional execution guarantees. However, none of these approaches addresses concurrency control and recovery jointly, thereby also taking into account the special semantics of processes as transactions at a higher level of semantics. Spheres of joint compensation, for instance, consider recovery aspects without concurrency control [17] while others only address isolation but neglect atomicity [5]. Other approaches, like open process management [9], are unnecessarily restrictive in that they defer the visibility of the effects of activities until the commit of their process. ConTracts [29], finally, although addressing concurrency control and recovery jointly, impose strong constraints and require the inverses of all process steps to exist.

The need for a dynamic scheduling protocol for transactional processes is stemming from the special semantics of process programs, i.e., the existence of multiple (alternative) subprocesses which are not necessarily considered in a process. Hence, predeclaration strategies [13] which would have to consider all possible activities and thus, in general considerably more than actually effected, are too restrictive. Similarly, protocols like altruistic locking [22], although designed for long lived transactions, do not provide a feasible solution for transactional processes since they require the access pattern of a process to be known beforehand (which would again include all possible execution paths).

## 6. CONCLUSION

This paper provides a dynamic scheduling protocol, termed process locking, for the correct parallel and fault-tolerant execution of transactional processes with respect to correctness criteria based on the unified theory of concurrency control and recovery. These processes can be considered as transactions at a higher level of semantics which are defined over activities corresponding to transactions in component systems.

Process locking allows to implement a process manager that exploits not only information about the commutativity of activities but also about their termination properties (whether there is an inverse or not), and the process structure, i.e., the orders that are imposed between activities, and allows to dynamically decide on the execution or deferment of single activities. By considering locking techniques at the level of activities rather than at data level and by distinguishing lock types according to the termination properties of activities, process locking takes into account the special semantics of processes. Most importantly, however, process locking allows certain locks to be ordered shared so as to increase concurrency while still conforming to the unified correctness criteria. Since process locking allows cascading aborts for processes, the exploitation of execution costs of single activities, leading to cost-based process scheduling, extends the basic protocol and permits cascading aborts for simple processes while cascading aborts are prevented for complex processes within the same framework. The consideration of execution costs then allows, for instance, to distinguish "expensive" activities which require manual interaction and automated ("cheap") activities that can be easily redone after an abort.

In the context of the WISE project of ETH, a process manager following the process locking protocol and the cost-based extensions has been implemented. In addition, the process manager is completed by the commercial process modeling tool IvyFrame [15] which has been extended so as to allow for the specification of cost information and for the validation of the correctness of single processes with respect to guaranteed termination. This framework has been successfully used in various applications such as electronic commerce, especially for the implementation and coordination of payment processes [25, 19], virtual enterprises [16], and subsystem coordination (i.e., in computer integrated manufacturing [24], or hospital information systems [26]).

*Acknowledgments.* The author thanks Hans-Jörg Schek and Gustavo Alonso for their encouragement and for the various fruitful discussions on the work presented in this paper. The process model and the unified correctness criteria for transactional processes have significantly benefited from discussions with Catriel Beeri.

## 7. REFERENCES

[1] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *Proceedings of the $9^{th}$ ACM Symposium on Principles of Database Systems (PODS'90)*, pages 85–93, Nashville, Tennessee, USA, April 1990. ACM Press.

[2] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. WISE: Business to Business E-Commerce. In *Proceedings of the $9^{th}$ International Workshop on Research Issues in Data Engineering. Information Technology for Virtual Enterprises (RIDE-VE'99)*, pages 132–139, Sydney, Australia, March 1999. IEEE Computer Society Press.

[3] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch. Distributed Processing over Stand-alone Systems and Applications. In *Proceedings of the $23^{rd}$ International Conference on Very Large Databases (VLDB'97)*, pages 575–579, Athens, Greece, August 1997. Morgan Kaufmann Publishers.

[4] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, and G. Weikum. Unifying Concurrency Control and Recovery of Transactions. *Information Systems*, 19(1):101–115, March 1994.

[5] B. Arpinar, S. Arpinar, U. Halici, and A. Doğaç. Correctness of Workflows in the Presence of Concurrency. In *Proceedings of the $3^{rd}$ Next Generation Information Technologies and Systems Conference (NGITS'97)*, Neve Ilan, Israel, June 1997.

[6] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[8] J. Camp, M. Harkavy, D. Tygar, and B. Yee. Anonymous Atomic Transactions. In *Proceedings of the $2^{nd}$ USENIX Workshop on Electronic Commerce*, pages 123–133, Oakland, California, USA, November 1996. The USENIX Association.

[9] Q. Chen and U. Dayal. A Transactional Nested Process Management System. In *Proceedings of the $12^{th}$ International Conference on Data Engineering (ICDE'96)*, pages 566–573, New Orleans, Louisiana, USA, 1996. IEEE Computer Society Press.

[10] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[11] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[12] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–243, June 1981.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[14] C. Hagen. *A Generic Kernel for Reliable Process Support*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1999. Diss. ETH Nr. 13114.

[15] IvyTeam, Zug, Switzerland. `http://www.ivyteam.com`.

[16] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE Approach to Electronic Commerce. *International Journal of Computer Systems Science & Engineering*, 15(5):343–355, September 2000. Special Issue on Flexible Workflow Technology Driving the Networked Economy.

[17] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'95)*, Informatik Aktuell, pages 51–70, Dresden, Germany, March 1995. Springer Verlag.

[18] S. Mehrotra, R. Rastogi, A. Silberschatz, and H. Korth. A Transaction Model for Multidatabase Systems. In *Proceedings of the $12^{th}$ International Conference on Distributed Computing Systems (ICDCS'92)*, pages 56–63, Yokohama, Japan, June 1992. IEEE Computer Society Press.

[19] A. Popovici, H. Schuldt, and H.-J. Schek. Generation and Verification of Heterogeneous Purchase Processes. In *Proceedings of the International Workshop on Technologies for E–Services (TES'00)*, Cairo, Egypt, September 2000.

[20] M. Reichert and P. Dadam. ADEPT$_{\text{flex}}$ — Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, March 1998.

[21] D. Rosenkrantz, R. Stearns, and P. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, June 1978.

[22] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic Locking: A Strategy for Coping with Long Lived Transactions. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems (HPTS'87)*, pages 175–198, Asilomar, California, USA, September 1987. Springer LNCS, Vol. 359.

[23] H.-J. Schek, K. Böhm, T. Grabs, U. Röhm, H. Schuldt, and R. Weber. Hyperdatabases. In *Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE'00)*, pages 14–23, Hong Kong, China, June 2000. IEEE Computer Society Press.

[24] H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 316–326, Philadelphia, Pennsylvania, USA, May/June 1999. ACM Press.

[25] H. Schuldt, A. Popovici, and H.-J. Schek. Automatic Generation of Reliable E-Commerce Payment Processes. In *Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE'00)*, pages 434–441, Hong Kong, China, June 2000. IEEE Computer Society Press.

[26] C. Schuler, H. Schuldt, and H.-J. Schek. Transactional Execution Guarantees for Data–Intensive Processes in Medical Information Systems. In *Proceedings of the 1st European Workshop on Computer-based Support for Clinical Guidelines and Protocols (EWGLP'2000)*, Leipzig, Germany, November 2000.

[27] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, June 1979.

[28] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, 190(2):363–396, January 1998.

[29] H. Wächter and A. Reuter. *The ConTract Model*, chapter 7, pages 219–263. In: [10]. Morgan Kaufmann Publishers, 1992.

[30] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 67–78, Minneapolis, Minnesota, USA, May 1994. ACM Press.

# APPENDIX

## A. PROCESS LOCKING: PROOF OF CORRECTNESS

In what follows, we prove the correctness of process locking. In particular, we show that the protocol produces P-RED process schedules which can be completed correctly (CT) and that these process schedules also meet P-RC.

### A.1 Process Locking and CT

Process locking has to ensure that each prefix of a complete process schedule can be reduced to a serial process schedule and, in particular, that this is also possible for each completed process schedule.

LEMMA 2 (COMMIT ORDERS).
*If two conflicting activities, $a_{i_k}$ and $a_{j_l}$, exit in a process schedule $\mathcal{S}$ with observed execution order $a_{i_k} <_{\mathcal{S}} a_{j_l}$, then, if both processes commit, the following order holds on their commits: $C_i <_{\mathcal{S}} C_j$.* □

PROOF (LEMMA 2).
*Assume that $a_{i_k}$ and $a_{j_l}$ conflict and that the order $a_{i_k} <_{\mathcal{S}} a_{j_l}$ is observed in a process schedule $\mathcal{S}$. Assume further that $C_j <_{\mathcal{S}} C_i$ holds.*

1. *If $C_i <_{\mathcal{S}} a_{j_l}$, that is, if $a_{j_l}$ is executed after the termination of $P_i$, then $a_{i_k} <_{\mathcal{S}} a_{j_l}$ is observed in $\mathcal{S}$ but the order on the commits of $P_i$ and $P_j$ trivially evaluates to: $C_i <_{\mathcal{S}} C_j$.*
2. *If $a_{j_l} <_{\mathcal{S}} C_i$, that is, if both $P_i$ and $P_j$ are executed concurrently, the following two cases can be distinguished:*
   (a) *If the locks for $a_{i_k}$ and $a_{j_l}$ cannot be ordered shared but have to be exclusive, then, due to the Commit-Rule of process locking, $C_i <_{\mathcal{S}} a_{j_l}$ is enforced such that $C_i <_{\mathcal{S}} C_j$ holds.*
   (b) *If the locks for $a_{i_k}$ and $a_{j_l}$ can be ordered shared, then the lock of $P_j$ held for $a_{j_l}$ is on hold as long as $P_i$ is active. Hence, the Commit–Rule requires $C_i <_{\mathcal{S}} C_j$.*

*All possible cases contradict with the above assumption such that $C_i <_{\mathcal{S}} C_j$ is enforced by process locking.* □

Correct termination not only requires all pairs of conflicting activities of committed processes to be in the same order but also accounts for aborted processes.

THEOREM 1 (PROCESS LOCKING AND CT).
*Process locking guarantees CT.* □

PROOF (THEOREM 1).
*Assume that a complete process schedule $\mathcal{S}$ generated by process locking is not CT. This is the case when cyclic conflicts exist in $\mathcal{S}$ that cannot be reduced. Let $P_i \rightarrow P_j \rightarrow \ldots \rightarrow P_n \rightarrow P_i$ the processes involved in this cycle.*

1. *Assume that all activities imposing the cycle are regular activities (either compensatable or pivot).*
   (a) *If all processes of this cycle are committed in $\mathcal{S}$, according to Lemma 2, the following order on the commits would have to hold: $C_i <_{\mathcal{S}} C_j <_{\mathcal{S}} \ldots <_{\mathcal{S}} C_n <_{\mathcal{S}} C_i$, yet in particular $C_i <_{\mathcal{S}} C_i$.*
   (b) *If some process $P_j$ of the cycle is aborted, then the corresponding inverse activities are present in $\mathcal{S}$. Due to the edge $P_j \rightarrow P_{j+1}$, process $P_{j+1}$ has locks on hold and thus cannot commit. Rather, by the $C^{-1}$–Rule for compensating activities of $P_j$, the cascading abort of $P_{j+1}$ is required which again leads to cascading aborts of all processes $P_{j+m}$ having locks on hold (i.e., all $P_{j+m}$ appearing after $P_{j+1}$ in the conflict cycle). Due to the isolation*

*of compensation (c.f. the $C^{-1}$–Rule and the special treatment of aborting processes as part of the Comp–Rule and the Piv–Rule), each process is correctly undone and its regular and compensatable activities can be removed from $\mathcal{S}$. Assume that some $P_{j+m}$ is completing, thus cannot be aborted. In this case, due to the Piv–Rule and the Comp→Piv–Rule, preceding processes with locks on hold are aborted (again, this is done in isolation such that regular/compensating activities can be removed during reduction).*

2. *Assume that the cycle is imposed by regular and compensating activities. Hence, due to the assumption of perfect commutativity, such a cycle would be induced by: $a_{i_k} <_{\mathcal{S}} a_{j_l} <_{\mathcal{S}} a_{i_k}^{-1}$ where $a_{i_k}$ and $a_{j_l}$ and, since commutativity is perfect, also $a_{j_l}$ and $a_{i_k}^{-1}$ do not commute. The $C^{-1}$–Rule for $a_{i_k}^{-1}$ requires $P_j$ to abort (since $P_j$ has a lock on hold, it cannot commit prior to $a_{i_k}^{-1}$) and $a_{i_k}^{-1}$ to be deferred until $A_j$ such that the following order can be observed: $a_{i_k} <_{\mathcal{S}} a_{j_l} <_{\mathcal{S}} a_{j_l}^{-1} <_{\mathcal{S}} a_{i_k}^{-1}$. But, this can be reduced correctly. Moreover, due to the Piv–Rule, $P_j$ is not allowed to be in state completing such that the abort required by the $C^{-1}$–Rule for $a_{i_k}^{-1}$ is possible.*
3. *Assume that the cycle is imposed by compensating activities only. But, due to the $C^{-1}$–Rule and the special treatment of aborting processes which is part of the Comp–Rule and of the Piv–Rule, processes are aborted in isolation such that this case cannot happen.*

*All these cases have shown that cyclic conflicts, although they may appear in a completed process schedule $\mathcal{S}$ (when cyclic conflicts exist, this always goes along with the presence of compensating activities), can be treated correctly by applying the reduction rules. Hence, violations of CT cannot occur.* □

When completing partial process schedules, the restriction that only one completing process at a time is allowed is essential for the avoidance of unresolvable deadlocks since, when cyclic wait-for dependencies exist, at most one of the processes involved in this cycle cannot be aborted to break the cycle.

## A.2 Process Locking and P-RC

In addition to CT, we have seen that a process manager has to guarantee that each partial process schedule is P-RC so as to decide to abort any arbitrary subset of running processes.

THEOREM 2 (PROCESS LOCKING AND P-RC). *Process locking guarantees P-RC.* □

PROOF (THEOREM 2). *Assume that a process schedule $\mathcal{S}$ generated by process locking is not P-RC. According to Definition 7, a violation of the constraints imposed by a pair of conflicting activities $a_{i_k}^c$ and $a_{j_l}$, must exist. Hence, the following two cases depending on the termination property of $a_{j_l}$ have to be considered:*

1. *Assume that $a_{j_l}$ is compensatable, thus its execution necessitates a C lock to be granted to $P_j$. The observed order $a_{i_k}^c <_{\mathcal{S}} a_{j_l}^c$ requires shared C locks between $P_i$ and $P_j$ which is only possible if $P_i$ is older than $P_j$, that is if $ts(P_i) < ts(P_j)$, and if additionally $P_j$ is running.*

   *In case process $P_j$ wants to commit, then, due to the Commit-Rule, $C_j$ has to be deferred until the commit $C_i$ of $P_i$ since $P_j$ has a lock on hold. Therefore, process locking enforces the order $C_i <_{\mathcal{S}} C_j$.*

   *In case $P_j$ wants to execute a pivot activity $a_{j_p}^p$ succeeding $a_{j_l}^c$, that is $a_{j_l}^c \prec_j a_{j_p}^p$, then, according to the Comp→Piv–Rule for lock conversion, all C locks of $P_j$ would have to be converted to P locks. Since $P_i$ is older than $P_j$, this lock conversion for $a_{j_l}^c$ would have to be deferred until the C lock held by $P_i$ is released which will take place at commit time. Therefore, $C_i <_{\mathcal{S}} a_{j_p}^p$ will hold such that P-RC is preserved.*
2. *Assume that $a_{j_l}$ is pivot, thus its execution necessitates a P lock to be granted to $P_j$. Since $a_{i_k}^c$ and $a_{j_l}^p$ are in conflict, this P lock request of $P_j$ would not be reconcilable with the C lock already held by $P_i$. Due to the Piv-Rule, the P lock request of $P_j$ would have to be deferred until $P_i$ has released its locks which, according to the strict 2PL property of process locking, coincides with $C_i$. Therefore, the order $C_i <_{\mathcal{S}} a_{j_l}^p$ will be observed which is compliant to P-RC.*

*Both cases are treated correctly by process locking such that P-RC holds for each process schedule $\mathcal{S}$.* □