Atomicity and Isolation for Transactional Processes

HEIKO SCHULDT
Swiss Federal Institute of Technology (ETH)
GUSTAVO ALONSO
Swiss Federal Institute of Technology (ETH)
CATRIEL BEERI
The Hebrew University
and
HANS-JÖRG SCHEK
Swiss Federal Institute of Technology (ETH)

Processes are increasingly being used to make complex application logic explicit. Programming using processes has significant advantages but it poses a difficult problem from the system point of view in that the interactions between processes cannot be controlled using conventional techniques. In terms of recovery, the steps of a process are different from operations within a transaction. Each one has its own termination semantics and there are dependencies among the different steps. Regarding concurrency control, the flow of control of a process is more complex than in a flat transaction. A process may, for example, partially roll back its execution or may follow one of several alternatives. In this article, we deal with the problem of atomicity and isolation in the context of processes. We propose a unified model for concurrency control and recovery for processes and show how this model can be implemented in practice, thereby providing a complete framework for developing middleware applications using processes.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs; reliability*; H.2.4 [**Database Management**]: Systems—*concurrency; distributed databases; transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*; H.4.1 [**Information Systems Applications**]: Office Automation—*workflow management*

General Terms: Algorithms, Design, Reliability

Additional Key Words and Phrases: Advanced transaction models, business process management, electronic commerce, execution guarantees, locking, rocesses, semantically rich transactions, transactional workflows, unified theory of concurrency control and recovery

Authors' addresses: H. Schuldt, G. Alonso, and H.-J. Schek, Swiss Federal Institute of Technology (ETH) Zürich, Department of Computer Science, ETH Zentrum, CH-8092 Zürich, Switzerland, email: {schuldt,alonso,schek}@inf.ethz.ch; C. Beeri, School of Computer Science and Engineering, The Hebrew University, Jerusalem 91904, Israel, email: beeri@ca.huji.ac.il.

1. INTRODUCTION

Processes are well-defined sequences of computational steps executed in a coordinated manner. *Business processes* are an example. A business process represents the different steps within one organization or among several organizations required to complete a given task, for example, open a bank account or provide an electronic service. One important advantage of using processes is that they make the application logic of an information system explicit. Traditionally, this application logic is coded in C or C++ and is, therefore, difficult to evolve, understand, and maintain. By using processes, information systems have a high-level representation of their internal logic, thereby greatly facilitating the evolution and maintenance of this logic. As processes become more widely used, there is a clear need to address the problem of interprocess interaction. In particular, concurrency control and recovery are not well understood when transactions are grouped into entities with higher level semantics [Weihl 1988; Lynch et al. 1994; Vingralek et al. 1998; Schek et al. 2000] such as processes [Alonso 1997].

Regarding concurrency control, the flow of control of a process is more complex than in a flat transaction. A process may, for example, partially rollback its execution or may follow one of several alternatives. All these different possibilities need to be taken into consideration when deciding how to interleave processes. Also, because these are very long executions compared with traditional transactions, blocking between processes is not acceptable. The question is thus how to allow as much interaction and parallelism as possible while still guaranteeing correctness.

Similarly, in terms of recovery, a process' activities are different from operations within a transaction. Each step has its own termination semantics and there are dependencies among the different steps. Since these steps are executed over heterogeneous, autonomous systems, the transactional properties of these underlying systems must also be considered.

In this article, we present a unified model for concurrency control and recovery in transactional processes that tackles all these problems. We also show how the model has been implemented in a prototype. The challenge is to design correctness criteria that account for both concurrency control and recovery, can cope with the added structure found in processes, and reflect the layered structure of the system. The approach followed in this article extends previous work in the area of concurrency control and recovery in transactional process management [Schuldt et al. 1999a] and it is based on a reformulation of the unified theory of concurrency control and recovery [Schek et al. 1993; Alonso et al. 1994; Vingralek et al. 1998].

With this, the article makes three main contributions. First, we clarify the problem of process structure. Second, starting with the correctness of single processes based on extensions of the flexible transaction model [Elmagarmid et al. 1990; Mehrotra et al. 1992; Zhang et al. 1994, 2001], we provide correctness criteria for the concurrent execution of processes, taking into account the interaction between hierarchical schedulers [Alonso et al. 1997a, 1999a, 1999b]. Third, we describe a working prototype that we have implemented and which is being used in electronic commerce, workflow management, and specialized coordination tools [Alonso et al. 1997b, 1999c, Schuldt et al. 1999b]. This prototype and its application illustrates the usefulness of the process abstraction and the applicability of the theory developed in this article.

The article is structured as follows: Section 2 motivates our approach and discusses sample transactional processes. Section 3 introduces the system model, the model of transaction programs, and the process model. A detailed discussion on how correctness of multiprocess executions can be characterized is provided in Section 4. In Section 5, a dynamic scheduling protocol for the concurrent and fault-tolerant execution of transactional processes is presented. Details of the prototype system we have implemented are given in Section 6. Section 7 discusses related work. Section 8 concludes the article.

2. MOTIVATION

To illustrate and motivate the notion of process and its properties, we present here two examples. First, we briefly discuss some of the characteristics of processes. A process is an execution of a process program. Its steps, called *activities*, typically have transactional properties. A process may include one or more points of no-return. After such an activity is executed, rolling back the process is impossible; it must proceed to the end. If some activity after a no-return point fails, it must be retried. Or, a different path that guarantees completion must be selected. This is in sharp contrast to classical transactions where the only no-return point is the commit operation at the end, and it is assumed that all noncompensatable activities are deferred to commit time. In classical transactions, a transaction scheduler's knowledge about the state of execution of a transaction program is encapsulated in the begin transaction and end transaction messages. For processes, because of the existence of no-return points anywhere in the execution, a process scheduler needs to know more about the state of execution.

The first example shows how process programs define concrete process executions. To this end, we use the payment interactions found in business-to-customer (B2C) electronic commerce applications.

Example 2.1 (*Payment Processes in B2C–Electronic Commerce*). The goal here is to allow on-line purchasing, of possibly several items, in a single atomic transaction (distributed purchase atomicity [Schuldt et al. 2000]). It is assumed that the items are digital goods, delivered on-line. One issue is to tie the delivery of the items to the payment—if the payment fails, the customer should not be able to use the items, and if the items have not been delivered, the payment will not be performed. This can be done through a trusted third party, a payment coordinator, which runs appropriate payment process programs. The structure of such a process program for payments is shown in Figure 1. A crucial aspect is the use of cryptographic techniques to decouple the transfer of encrypted items from the transfer of the appropriate cryptographic keys [Camp et al. 1996; Tygar 1996, 1998]. The delivery of the cryptographic keys is performed by the trusted third party, and is coupled by this party with procedures for ensuring

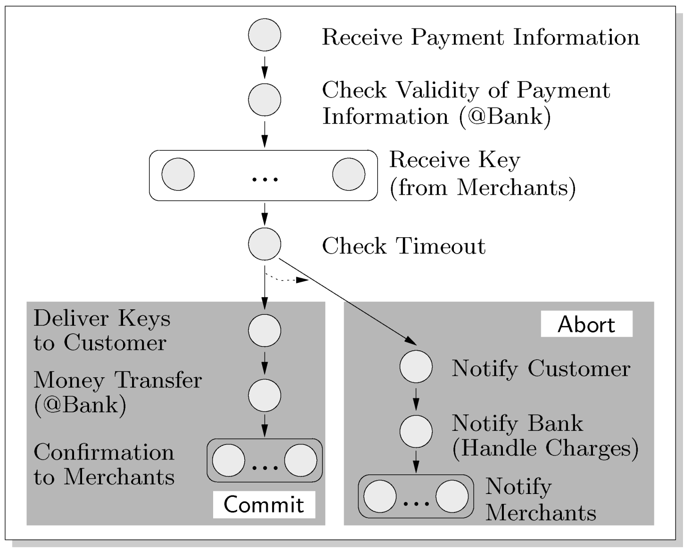

Receive Payment Information
Check Validity of Payment Information (@Bank)
Receive Key (from Merchants)
Check Timeout
Deliver Keys to Customer
Money Transfer (@Bank)
Confirmation to Merchants
Commit
Abort
Notify Customer
Notify Bank (Handle Charges)
Notify Merchants

Fig. 1. Process program for payment interactions in electronic commerce.

the payment. Note that the encrypted goods have been delivered to the customer before the start of the process, but are not usable until the corresponding keys are transferred correctly.

In Figure 1, the individual activities of the process program are depicted as circles, the flow of control between activities as arcs. Note that activities are transactional. For example, after the first step, the customer is committed to the payment. The business transaction may still fail to go through if, for instance, the bank refuses to validate and guarantee the payment. Within the process program, two points of no-return exist, namely "check timeout" and "key delivery." First, after all keys have been received and the timeout check has been successful, execution has to terminate (commit or abort). Check timeout essentially writes a legally binding log entry at the payment coordinator's site recording the receipt of all cryptographic keys. After this log record is generated, the process cannot be simply rolled back by discarding the received information. Second, after the keys have been delivered to the customer in the commit branch, the correct money transfer must take place. This is guaranteed by the bank since in a previous step ("check payment information"), the money (credit card number or e-cash token) has been identified as correct and a lock has been placed on it. If any of the activities after the second point of no-return, for example, "notify merchant" fails, it will be retried until it succeeds. Note, however, that the failure of the second no-return activity, that is, due to the unavailability of the customer preventing correct key delivery, must not lead to a complete rollback of the process (it already follows another no-return activity). Rather, the alternative (abort) branch is taken. The effects of the activities of the abort branch are similar to an abort of the process prior to the first no-return activity. However, their semantics is slightly different since the business

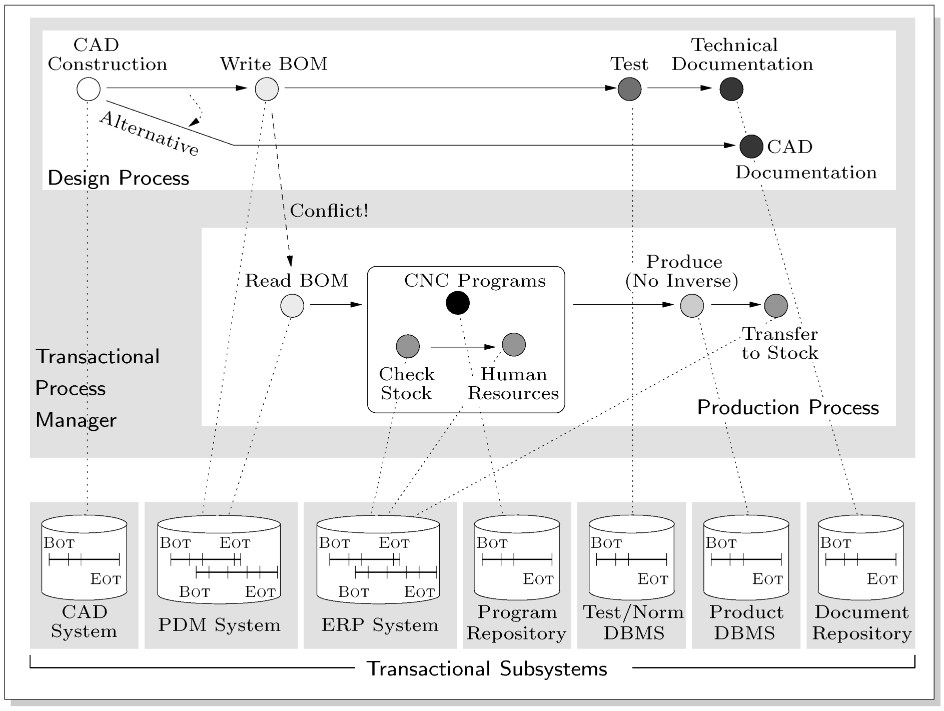


Fig. 2. Concurrent processes in CIM applications.

transaction has already been marked as completed by the previously generated log record. In some cases, the abort branch may even consider a small service fee to be charged from the customer. The shaded boxes represent alternatives where an order indicates a preference over them (here, the commit branch is preferred over the abort branch).

The previous example presented a process in isolation and illustrated the notion of a no-return activity and of a retriable activity. The next example, taken from Computer Integrated Manufacturing, outlines the problems that might occur when process programs are executed in parallel. It shows that the possibility of aborts in the execution of one process may impose restrictions on the execution of another concurrent process. In particular, the second process may be prevented from executing one of its no-return points, until the first process has successfully executed its no-return activity.

Example 2.2 (*Computer Integrated Manufacturing* (*CIM*)). The objective here is to bring goods to the marketplace as soon as possible. To this end, the design and production should run as parallel as possible with the risk that production must be stopped or modified if tests in the design phase fail. The two-process programs in Figure 2 describe the customized design and production of certain goods [Schuldt et al. 1998]. Due to the customization, the design of the product and its manufacture are strongly tied. The link between both processes is the bill of materials (BOM). The BOM is generated within the design process and provides the input required by the production process (via services accessing shared resources in the underlying applications). Thus, the BOM creates a dependency between both processes.

This example shows the tree structure of process programs. This structure guarantees a more general notion of atomicity than traditional transactions. In the case of the design process, for instance, a failure detected during the test activity undoes only the PDM entry and alternatively documents the CAD drawing so as to facilitate later reuse. It does not discard the outcomes of the complex CAD activity as would be the case with the traditional all-or-nothing semantics of atomicity.

An additional prerequisite is to guarantee consistent interaction between processes. Consider the parallel execution of the design and production process. This parallelization is important in practice. As can be seen in Figure 2, only the two activities within the PDM system conflict. For concurrency control purposes, to order these two activities would be sufficient. However, when recovery has to be considered, further dependencies exist. As no inverse for the production activity exists, it must not be executed before the test is terminated successfully. If the test fails, the PDM entry is compensated within the design process and the BOM read by the production process is invalidated. Therefore, all activities of the production process would have to be compensated, too, including the "produce" step which is actually a point of no return of the production process. Yet, if production of parts is already performed, this would lead to severe inconsistencies as no valid design and BOM of these parts exists. Hence, in the context of concurrency, compensation might not be sufficient to guarantee correct executions. Additional dependencies have to be taken into account. These dependencies arise from the special termination characteristics of activities, that is, whether or not an inverse exists. In the traditional transaction model, where each operation has an inverse, these dependencies do not appear.

3. MODEL

3.1 System Model

We consider an architecture with two layers. The top layer involves the execution of *transactional processes*, as specified in *process programs*. A process program is a set of partially ordered activities. Each activity, in turn, corresponds to a conventional transaction, or *transaction program*, executed in a transactional application. The bottom layer of the system model is formed by the available transactional applications (see Figure 3).

The concurrent execution of transactional processes is controlled by a *transactional process manager* (*PM*), which is responsible for scheduling the execution of the transaction programs. In here, we are interested in correctness at the level of the invocation of transaction programs (marked with $(*)$ in Figure 3). For the scheduling, the PM exploits information about the commutativity of transaction programs, termination properties of these programs, and

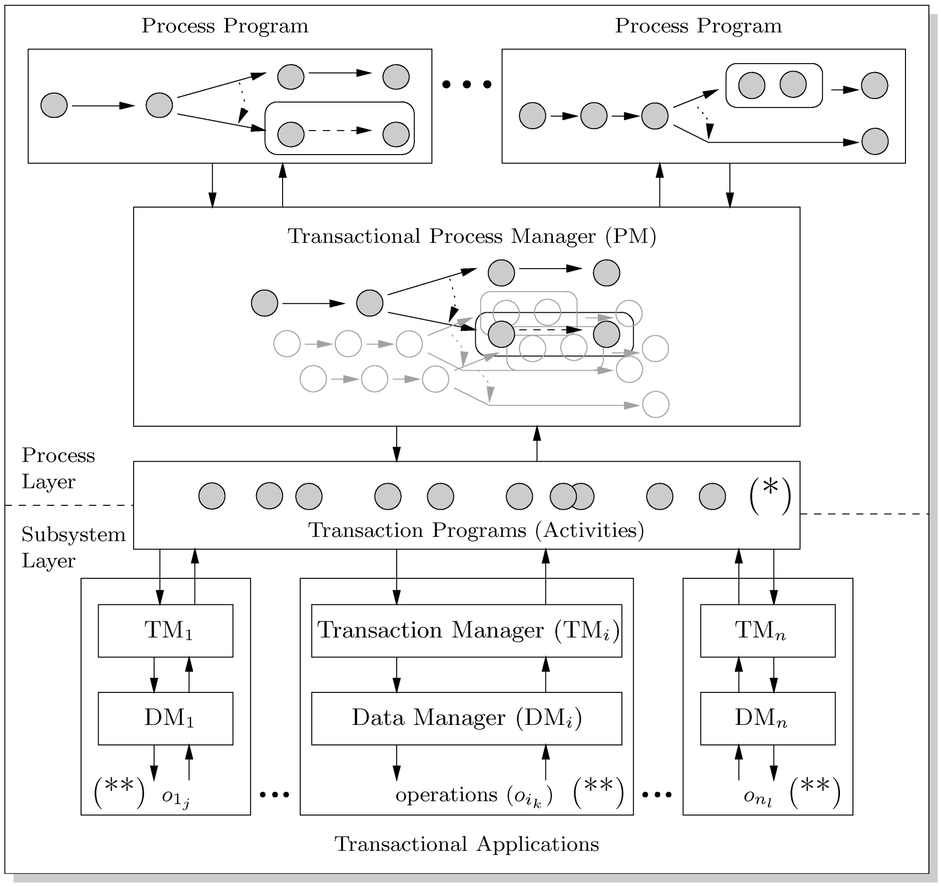

Process Program
Process Program
Transactional Process Manager (PM)
Process Layer
Subsystem Layer
(*)
Transaction Programs (Activities)
TM$_1$
DM$_1$
Transaction Manager (TM$_i$)
Data Manager (DM$_i$)
TM$_n$
DM$_n$
(**) o_{1_j}
operations (o_{i_k}) (**)
o_{n_l} (**)
Transactional Applications

Fig. 3. System model.

the process structure. For the underlying applications, we assume a conventional architecture [Bernstein et al. 1987] where a *transaction manager* (TM_i) executes transactional programs by submitting operations to a *data manager* (DM_i).

3.2 Transaction Programs Model

Processes are conventionally seen as a collection of activities. In here, a process program is a structured collection of *transaction programs*, or *activities*. Activities are, by definition, atomic. Let $\mathcal{A}^*$ be the set of all activities available in the system. To account for aborts and commits of processes, we augment $\mathcal{A}^*$ to $\hat{\mathcal{A}} := \mathcal{A}^* \cup \{C, A\}$ where C denotes the commit of a process and A its abort. Each activity is assumed to provide a return value indicating whether it succeeded or failed. We treat the transactional service run by an activity as a black box. Hence, the process manager can only reason about the activity's outcome on the basis of its return value.

Activities differ in terms of their termination guarantees. We consider three cases: compensatable, retriable, and pivot [Mehrotra et al. 1992; Zhang et al. 1994]. A *compensatable* activity has a compensating activity that semantically undoes the effects of the original activity. More formally,

Definition 3.1 (*Effect-Free Activities*). Let $\sigma = \langle a_i \ a_j \dots a_n \rangle$ be a sequence of activities from $\mathcal{A}^*$. The sequence σ is *effect-free* if, for all possible activity sequences α and ω from $\mathcal{A}^*$, the return values of all activities of α and ω in the concatenated activity sequence $\langle \alpha \ \sigma \ \omega \rangle$ are the same as in the activity sequence $\langle \alpha \ \omega \rangle$.

A special case of effect-free activities is the sequence $\sigma = \langle a_i \ a_i^{-1} \rangle$ consisting of a compensatable activity a_i and its compensating activity a_i^{-1}. More formally,

Definition 3.2 (*Compensatability and Compensating Activity*). An activity $a_i \in \mathcal{A}^*$ is *compensatable* if there is an activity $a_i^{-1} \in \mathcal{A}^*$ where the activity sequence $\sigma = \langle a_i \ a_i^{-1} \rangle$ is effect-free. Activity a_i^{-1} is called the *compensating activity* of a_i.

If an activity is not compensatable, it is called *pivot*. Obviously, once a pivot has executed successfully, the process must proceed. Rolling back by compensation is not an option any more. Activities whose executions are guaranteed to successfully terminate after a finite number of invocations are called *retriable*.

Definition 3.3 (*Retriable Activity*). An activity $a_i \in \mathcal{A}^*$ is *retriable* if each sequence α of activities from $\mathcal{A}^*$ can be expanded to $\langle \alpha \ a_i \rangle$ by invoking a_i a finite number of times such that the last invocation terminates committing while all the previous ones return with abort.

Retriability must be guaranteed by the semantics of the transaction associated with the activity. Retriable activities can be implemented, for instance, by using escrow-like mechanisms [O'Neil 1986]. Then, pairs of activities have to exist where successful execution of the first guarantees that the second is retriable, that is, that its subsequent execution (or repeated execution in case of failures) is guaranteed to eventually succeed. Such escrow-like semantics can be found, for example, in the money transfer activity of the payment process (Example 2.1). In case the initial check of the payment information is positive, the bank guarantees that the money will be eventually credited, if requested. This is usually done by marking an electronic cash token as "being in use" or by reserving a certain amount on the credit card account. Hence, the actual guaranteed (retriable) money transfer is shifted to the end of the process.

Note that in contrast to the flex transaction model [Mehrotra et al. 1992; Zhang et al. 1994], retriability and availability of compensation are orthogonal properties. In our model, an activity can have both, one of them, or neither.

Note further that the semantics of compensation does not require each compensating activity to be itself compensatable. However, we assume each compensating activity to be retriable and, therefore, guaranteed to succeed. In what follows, we indicate the termination property of an activity a_i by a superscript, that is, a_i^c for a compensatable, a_i^p for a pivot, and a_i^r for a retriable activity, respectively.

3.3 Ordering Constraints

A *process program* executes activities according to the results of past activities. Thus, a process program can be viewed as a tree whose nodes are activities and whose edges correspond to order constraints between these activities. A path in the tree reflects the effects of a possible execution.

We generalize this view as follows: If a program allows for concurrent execution of activities, we group a partially ordered set of activities as a single (multiactivity) node of the tree, rather than as distinct nodes. Nodes, whether singleton or multiactivity, are totally ordered with respect to preceding and subsequent nodes of the tree—if node n_1 precedes n_2, then all activities of n_1 must terminate before any activity of n_2 starts. The temporal order on activities, defined as the union of the partial orders within the nodes, and the order induced on them by the ordering of the nodes, is called the *strong order,* $\ll$. It has the semantics of a handshake.

A further generalization is possible if we assume that a process program may allow some activities, say a and b, to be executed concurrently, but *request* that the execution be equivalent to one in which a precedes b (*weak order*, $<$) [Alonso et al. 1997]. Ensuring such constraints is a service provided by the underlying systems, and can be accomplished by, for example, using commit-order serializability [Beeri et al. 1989]. Recalling that a multiactivity node is a partially ordered set of activities, we represent such requests by associating a weak order request and a partial strong order with each multiactivity node. Thus, if activities are strongly ordered in a multiactivity node, they will be executed in this order, since the process program will invoke the second only when the first returns. If they are weakly ordered, the necessary ordering will be enforced by the underlying system.

3.4 Process Structure

The first noncompensatable activity on a path from the root in the tree is a (*primary*) *pivot* of the process. It marks a no-return point in the process: if it commits, the process cannot roll back any more; it must be able to complete. Pivots are always represented as singleton nodes, rather than as members of a multiactivity node. This captures the fact that no other activity of a process may be executed in parallel to a pivot activity. To be able to complete after a pivot commits, there must be at least one tree, the *assured termination tree*, starting from the pivot, consisting only of retriable activities.

After successfully executing a pivot, the process program may try different alternatives, and only if they fail, execute the one whose termination is assured. Generally, a pivot may have an *ordered* set of children, each a subprocess program, such that the last one has an assured termination path, and all previous ones have (recursively) the properties of a process program. In particular, each of these may have its own pivots, assured termination path, and so on.

Note that the children of a regular node are not ordered. The program selects *one* of them, according to results of previous activities.[1]

These ideas can be formalized in the following definition.

Definition 3.4 (*Process Program*). A *process program*, PP, is a tuple $(N, E, \ll, <, Piv, \lhd)$, where

(1) N is a set of nodes. Each node $n \in N$ consists of a set $\mathfrak{a}_n \subseteq \mathcal{A}^*$ of activities. If $card(\mathfrak{a}_n) = 1$, a node is called singleton node; otherwise, n is called multiactivity node. Associated with each multiactivity node $n \in N$ are two different orders on the corresponding activities: a partial strong order, $\ll_n$, and a partial weak order, $<_n$. Each noncompensatable activity is included in a singleton node.

(2) E is a set of directed edges which, together with the set of nodes N, forms a directed tree. Each edge $e = (n_1, n_2) \in E$ corresponds to a strong order constraint.

(3) The (partial) strong order, $\ll$, is the union of the order induced on activities by the edges and of the partial orders, $\ll_n$, $n \in N$, within multiactivity nodes.

(4) The (partial) weak order, $<$, is the union of all weak orders, $<_n$, $n \in N$ of all multiactivity nodes.

(5) Piv is a set of distinguished singleton nodes in the tree: each singleton node in the tree containing an activity that is not compensatable is in Piv. The first nodes of Piv on a path from the root of the tree are called *primary pivots*.[2]

(6) The *preference order*, $\lhd$, defines a total order on the children of each member of Piv. The last child must be the root of an assured termination tree. Each other child must be a process program.

The universe $\mathfrak{a}_{PP} \subseteq \mathcal{A}^*$ of all activities explicitly encompassed in a process program PP is the union of all activities of all nodes, that is $\mathfrak{a}_{PP} := \bigcup_{n \in N} \mathfrak{a}_n$.

Example 2.1 (*Revisited*). The payment process program depicted in Figure 1 encompasses three multiactivity nodes and two pivot activities, "check timeout" and "transfer keys." The abort subtree is the assured termination tree of "check timeout" and the remaining activities of the commit branch are the guaranteed completion after the second pivot activity "transfer keys." The preference order, $\lhd$, guarantees that the commit subtree is preferred to the abort subtree.

Note that a process program may have no pivot, in which case it has the same properties as a regular transaction, that is, it can be aborted any time prior to its commit.

[1]A generalization of this model may also allow a partial order on the children of nonpivot nodes. Then, for failure-handling purposes, this partial order may specify which alternative has to be taken in case of failure.

[2]In case of branching prior to the first pivot activity in a process program tree, actually more than one primary pivots may exist.

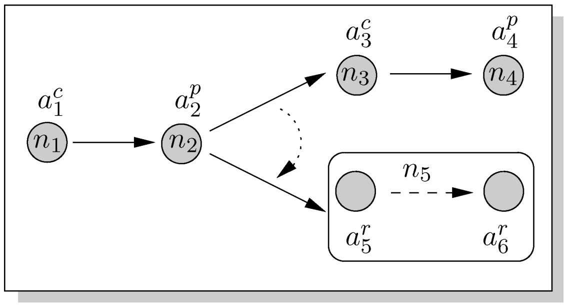


Fig. 4. Process Program PP_1 with strong ($\ll$) and weak ($<$) order constraints and preference order ($\lhd$).

In Zhang et al. [1994], it has been shown that well-formed flex structures always guarantee the existence of one execution path that can be executed correctly while all other paths will leave no effects. The definition above captures this property. If a program has not reached a primary pivot, it can abort by executing compensating activities. After executing the pivot, it must proceed but there is always one alternative guaranteed to succeed. Hence, a process program may encompass several pivots on a path from the root to a leaf as long as each of these pivots is followed by an assured termination tree. We refer to process programs conforming to Definition 3.4 as having *guaranteed termination*. The guaranteed termination property of transactional processes is a generalization of the atomicity in traditional ACID transactions.

In the following, when depicting process programs, we use solid arcs for strong order constraints and dashed arcs for weak order constraints. The preference order, $\lhd$, will be depicted by dotted arcs.

Example 3.2 (*Process Program Structure*). Consider process program PP_1 depicted in Figure 4. PP_1 consists of five nodes: four of them are singleton nodes (n_1 encompassing activity a_1^c, n_2 with activity a_2^p, n_3 with activity a_3^c, and n_4 with activity a_4^p), and the multiactivity node n_5 encompassing activities $\mathfrak{a}_5 = \{a_5^r, a_6^r\}$. The strong order, $\ll$, is given by $\ll = \{(a_1^c \ll a_2^p), (a_2^p \ll a_3^c), (a_3^c \ll a_4^p), (a_2^p \ll a_5^r), (a_2^p \ll a_6^r)\}$ and the weak order of PP_1 by $< = \{(a_5^r < a_6^r)\}$. Finally, the alternative executions following the commit of the primary pivot, a_2^p, are given by an order on its children $\lhd = \{PP_{1_1} \lhd PP_{1_2}\}$ where PP_{1_1} is the subprocess program consisting of nodes n_3 and n_4 while PP_{1_2} is the subprocess program that contains the activities a_5^r and a_6^r of node n_5. Given these orders, a_5^r and therefore also a_6^r can only be executed after a_3^c has failed or after a_4^p has failed and a_3^c has been compensated by a_3^{-1}.

While we included the node identifiers in the process program tree depicted in Figure 4, in what follows we will only attach to each node its associated activities and omit the node identifiers.

3.5 Process States

We denote the execution of a single process program a *process*. We consider partial executions, in which a process may not have terminated. Although

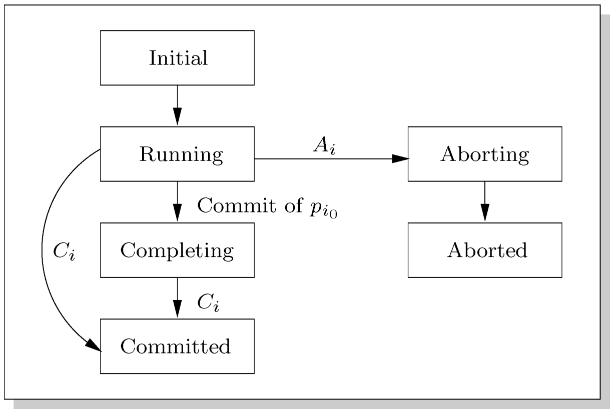


Fig. 5. Possible states of a process P_i.

executions are in general concurrent, the underlying system guarantees serializability at the activity level. Hence, for simplicity of exposition, let us consider totally ordered processes. In such serialized executions, we assume that all activities of a node are executed in an order compatible with the strong and weak orders defined within the node.

A process is assumed to have a unique identifier as subscript and, as a superscript, the id of the process program whose execution it reflects, for example, process P_i^k corresponds to PP_k. The latter index may be omitted when the associated process program is not relevant. Activities within P_i^k are denoted as $a_{i_1}^c,\ a_{i_2}^p, \ldots, a_{i_n}^r$. The superscript denotes the property of an activity, the subscripts are the process ID and a unique ID for the activity. Superscripts are omitted when not relevant or interesting. The commitment of process P_i is denoted by C_i, its abort by A_i.

A process execution is not a path in the tree. It may contain aborted activities, compensating activities for the process or for its subprocesses, and it may contain aborted executions of subprocesses. Yet, the actual effects of a process—all activities that are executed correctly and that are not compensated—are represented by a path in the tree. It is convenient to represent the point on a path reached by a process by the notion of a process state.

The possible states of processes and the associated state changes are as follows (Figure 5):

(1) Once instantiated, a process is in the state *running*.
(2) Prior to the commit of a primary pivot p_{i_0}, an abort, of a compensatable activity or of the primary pivot, changes the state to *aborting*, where compensating activities are executed.
(3) After finally having compensated each activity, a process is in the state *aborted*.
(4) The commit of a primary pivot p_{i_0} causes a state change from *running* to *completing*. The program may now try several alternatives. The failure (i.e., abort of an alternative) causes it to try the next one. Thus, its state specifies

the alternative, say j, in which it is now, and its state within the subprocess of this alternative.

(5) Finally, if an alternative commits, then the subprocess completes and so does the process, which then contains the commit activity C_i. The process changes to the final state *committed*.

A process that is either running or completing is called *active*.

Example 2.1 (*Revisited*). A payment process defined by the process program depicted in Figure 1 is running as long as the timeout check has not committed. In case of failure prior to the timeout check (e.g., failed check of an e-cash token), the process changes to aborting and, after a notification to the customer has been sent, changes to aborted. Once the timeout check has succeeded (where essentially a log entry is generated), the process is completing and has to terminate either along the commit or the abort branch. Due to the assured termination trees following either pivot activity, the process is guaranteed to finally change its state to committed.

3.6 Process Executions

A *process* is a collection of activities that may be performed in an execution of a process program, with the partial order that must hold between them in any execution (in an actual execution, the order may include this partial order). This partial order, the *required order*, is mainly induced by the ordering constraints in the process program, but it also reflects some natural constraints, such as the precedence of a regular activity to its compensation, if the latter occurs in the execution. Note that a process contains a subset of the activities explicitly specified in the associated process program. Since each activity is required to be atomic, the repeated invocation of retriable activities does not explicitly appear in the notion of a process. While aborted instances do not leave any effects, only the last invocation of a retriable activity, the one that is terminated correctly, is present. In addition to the regular activities, a process may contain compensating activities for some of the regular ones. More formally:

Definition 3.5 (*Process*). A *process*, P_i^k, is a tuple $(\mathcal{A}_i, \prec_i)$ reflecting the execution of a process program PP_k where

(1) $\mathcal{A}_i \subseteq (\mathfrak{a}_{PP_k} \cup \{a_{i_j}^{-1} \mid a_{i_j}^c \in \mathfrak{a}_{PP_k}\} \cup \{C_i, A_i\})$ is a set of activities that contains a subset of the regular activities of PP_k, but may also contain compensating activities for some of them. In addition, $\mathcal{A}_i$ may contain one of C_i or A_i.

(2) The *required order* $\prec_i \subseteq (\mathcal{A}_i \times \mathcal{A}_i)$ is the minimal partial order such that
   (a) If $a_{i_l},\ a_{i_m} \in \mathcal{A}_i$ and $a_{i_l} \ll_k a_{i_m}$ in PP_k, then $a_{i_l} \prec_i a_{i_m}$ (preservation of strong order)
   (b) If $a_{i_l}, a_{i_m} \in \mathcal{A}_i$ and $a_{i_l} <_k a_{i_m}$ in PP_k, then $a_{i_l} \prec_i a_{i_m}$ (preservation of weak order)
   (c) If a compensating activity $a_{i_l}^{-1}$ occurs in $\mathcal{A}_i$, then $a_{i_l} \prec_i a_{i_l}^{-1}$ and $a_{i_l} \in \mathcal{A}_i$
   (d) If, in addition to $a_{i_l}^{-1}$, also a compensating activity $a_{i_m}^{-1}$ occurs in $\mathcal{A}_i$ and $a_{i_l} \prec_i a_{i_m}$, then $a_{i_m}^{-1} \prec_i a_{i_l}^{-1}$

(e) Finally, if one of either C_i or A_i is in $\mathcal{A}_i$, then it has to be after all other activities of $\mathcal{A}_i$ with respect to $\prec_i$.

The above definitions show that not all activities of a process are explicitly given by its process program since $\mathcal{A}_i$ also includes compensating activities which are only implicitly present in PP_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*.

Similarly, the order constraints explicitly specified in a process program PP_k only consider regular activities. However, in accordance with Definition 3.5, an order between compensating activities and their regular activities as well as among compensating activities (given that the corresponding regular activities are also ordered) has to be imposed in a process. While a regular activity and its compensation have inevitably to be strongly ordered, we allow two compensating activities to be executed weakly ordered given that the two corresponding regular activities are also allowed to be weakly ordered by the process program. Otherwise, when two regular activities are strongly ordered, also their compensating activities have to be strongly ordered (in reverse order). All these constraints are present in the required order, $\prec_i$, of a process P_i.

A process, that is, the execution of a process program, can be best discussed based on its possible states.

(1) If a process is running, its execution is a sequence of compensatable activities, corresponding to a path from the root.
(2) When a process is aborting, its execution consists of a sequence of committed activities, followed by one aborted activity (that may be the primary pivot, or a preceding activity), followed by a sequence of compensating activities, in reverse order.
(3) The execution of an aborted process consists of a sequence as in the previous case, where each committed activity has a corresponding compensating activity, and the final activity is A_i. We refer to such an execution as an *abort process execution* of the process.
(4) After a process has changed its state from running to completing, it contains the activities on the path from the root to the primary pivot, followed by abort process executions for alternatives $1, \dots, j-1$, and an execution for alternative j compatible with its state.
(5) When a process is committed and when it has changed its state from completing to committed, before committing, retriable activities of the assured termination tree (of the process or of a subprocess) may abort, but then they are retried, so a process execution may contain a subsequence of m aborted instances of a retriable activity, followed by a committed one. When the commit is executed in state running, C_i is preceded only by compensatable activities.

3.7 Process Termination

We note that as long as the pivot on the selected path has not committed, the process can always abort (if executed in isolation). When P_i is either running

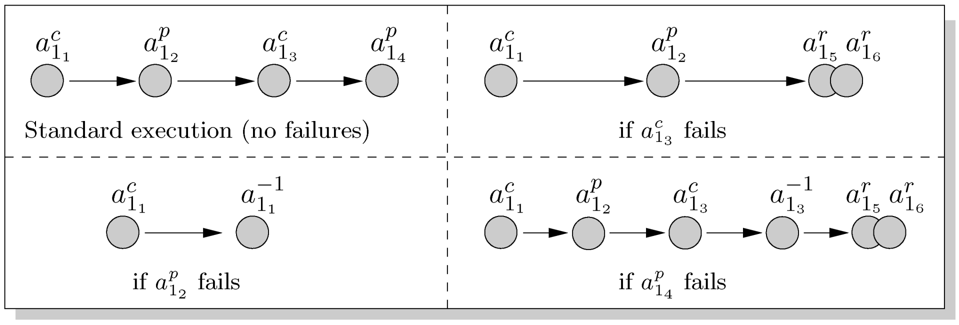

$a^c_{1_1}$ $a^p_{1_2}$ $a^c_{1_3}$ $a^p_{1_4}$
Standard execution (no failures)
$a^c_{1_1}$ $a^p_{1_2}$ $a^r_{1_5}$ $a^r_{1_6}$
if $a^c_{1_3}$ fails
$a^c_{1_1}$ $a^{-1}_{1_1}$
if $a^p_{1_2}$ fails
$a^c_{1_1}$ $a^p_{1_2}$ $a^c_{1_3}$ $a^{-1}_{1_3}$ $a^r_{1_5}$ $a^r_{1_6}$
if $a^p_{1_4}$ fails

Fig. 6. Possible executions of process program PP_1.

or aborting, it is said to be *backward-recoverable*. The sequence of activities to be executed in one of these states is called its *backward recovery path*. All activities of P_i preceding each primary pivot are compensatable. Therefore, if a primary pivot or one of the preceding activities fails, or if an abort A_i of P_i is requested for some other reason, backward recovery can be performed by successively applying compensation activities.

Once a primary pivot has terminated successfully, the process is in the state completing. Since it has a final alternative consisting of only retriable activities, and since all previous alternatives are smaller subprocesses with the same properties, a process in this state is guaranteed to complete. A process, P_i, is said to be *forward-recoverable* when it is completing. The sequence of activities leading from any activity succeeding a primary pivot to the well-defined termination of a process is the *forward recovery path*.

The partially ordered set of activities that need to be executed to terminate a partial process is called the *completion* of P_i, and denoted by $\mathcal{C}(P_i)$. Note that in the case of P_i being in state running, $\mathcal{C}(P_i)$ consists only of compensating activities. If P_i is in state completing, the structure of $\mathcal{C}(P_i)$ is more complex. If the process is in the assured termination subprocess, then it consists of a path of retriable activities; this is also the case if another subprocess of P_i is in an assured path. If the process is now in another subprocess, and the deepest subprocess is completing, the completion of P_i is determined by the completion of that subprocess. If this deepest subprocess is aborting, then it consists of its completion (bringing it to the aborted state), followed by the assured termination path of its parent. Hence, $\mathcal{C}(P_i)$ may consist of both compensating activities (for backward recovery of an aborting subprocess) followed by a path of retriable activities of a parent process.

Example 3.4 (*Execution of Process Programs*). Consider again process program PP_1 depicted in Figure 4. It is a process program with guaranteed termination. The four possible complete executions of PP_1 are depicted in Figure 6. In all these executions, activity $a^p_{1_2}$ is its primary pivot. Denote by P^1_1 a partial execution of PP_1. Before the successful termination of $a^p_{1_2}$, P^1_1 is running and in this state, the completion $\mathcal{C}(P^1_1)$ consists of $\{a^{-1}_{1_1}\}$, if $a^c_{1_1}$ has been executed successfully. After successful termination of $a^p_{1_2}$, P^1_1 is completing. After $a^c_{1_3}$, for instance, has committed and $a^p_{1_4}$ failed, the completion of P^1_1 evaluates to $\mathcal{C}(P^1_1) = \{a^{-1}_{1_3} \prec_1 a^r_{1_5} \prec_1 a^r_{1_6}\}$.

3.8 Correct Subsystem Execution

Before discussing scheduling at the process manager (PM) level, we first have to clarify the requirements imposed on the subsystems. In here, according to the traditional transaction model, we consider all operations of a transaction program—except for those that can be executed after the commit decision—to be compensatable.

Each subsystem has to produce SR-ACA schedules [Bernstein et al. 1987]—that is schedules that are both serializable and avoid cascading aborts—when executing operations belonging to transactions at the data manager (DM) level of each subsystem (marked with (**) in Figure 3). In addition, each subsystem must

—provide order preserving serializability (OPSR) [Bernstein et al. 1979; Papadimitriou 1979] so that the serialization order in each subsystem matches the strong order imposed by the process manager.

—allow the process manager to determine the serialization order for any pair of activities of the same subsystem and to impose this order to the subsystem. This can be achieved using commit-order serializability [Breitbart et al. 1991; Raz 1992] in all subsystems. This allows the process manager to map the desired serialization order of activities to the commit order of the associated transactions. Furthermore, commit ordering implies order preserving serializability, thus fulfilling the compliance of strong (temporal) orders in all subsystems. When commit-order serializability is provided, the requirement of order preservation as a handshaking mechanism between schedulers [Beeri et al. 1989] holds.

A general discussion of these requisites and on how they can be implemented in practice can be found in Schuldt et al. [1998, 1999b] and Schuler et al. [1999].

Note that the consideration of a two-level system is not an intrinsic requirement of transactional process management. Rather, it is motivated by the goal of this article to clarify the contracts between subsystems and a process manager in order to provide correct process executions. Yet, the model can be extended to address multiple levels as long as the above mentioned requirements are met and the resources accessed by different subsystems are pairwise disjoint.

4. CORRECT PROCESS EXECUTION

4.1 Process Schedule

The main prerequisite for multiprocess executions is that each process program itself is inherently correct:

Axiom 4.1 (*Guaranteed Termination*). All process programs fulfill the guaranteed termination property.

Given the correct structure of process programs with guaranteed termination, a *process schedule* reflects concurrent processes, that is, the concurrent execution of process programs. The notion of process schedule $\mathcal{S}$ defined over a set of processes $\mathcal{P}_\mathcal{S}$ includes both regular activities and recovery related, that is, compensating, activities as they are considered in processes. Furthermore, a process schedule not only includes the observed execution order $<_\mathcal{S}$ between activities, but also the required order $\prec_\mathcal{S}$ which is specified for each process by its corresponding process program. The need for including the required order explicitly in a process schedule stems from the fact that a process program may not require any order on some activities, that is, they are not related by $\prec_i$, yet they are ordered by $<_\mathcal{S}$ in some execution. Therefore, without $\prec_\mathcal{S}$, the requested orders of processes could not be reconstructed from a process schedule. However, in order to apply reduction techniques (Section 4.6) to process schedules, this information is required.

Definition 4.2 (*Process Schedule* $\mathcal{S}$). 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 (partial) processes $P_i = (\mathcal{A}_i, \prec_i)$.
(2) $\mathcal{A}_\mathcal{S} \subseteq \hat{\mathcal{A}}$ 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) $\prec_\mathcal{S}$ is a partial order between activities of $\mathcal{A}_\mathcal{S}$, called the *required order*, with $\prec_\mathcal{S} \subseteq (\mathcal{A}_\mathcal{S} \times \mathcal{A}_\mathcal{S})$ which includes the required order for each process of $\mathcal{P}_\mathcal{S}$, that is $\prec_\mathcal{S} = \bigcup_{P_i \in \mathcal{P}_\mathcal{S}} \prec_i$.
(4) $<_\mathcal{S}$ is a partial order between activities of $\mathcal{A}_\mathcal{S}$, called *execution order*, with $<_\mathcal{S} \subseteq (\mathcal{A}_\mathcal{S} \times \mathcal{A}_\mathcal{S})$ reflecting the observed order in which activities are executed. The required order $\prec_\mathcal{S}$ of $\mathcal{S}$ is contained in its observed execution order $<_\mathcal{S}$, that is $\prec_\mathcal{S} \subseteq <_\mathcal{S}$.

This definition of a process schedule reflects the invocation of activities at the process manager level as marked with (*) in Figure 3. In general, processes do not need to have terminated in $\mathcal{S}$. Therefore, the above definition includes both partial and complete processes. Accordingly, if all processes P_i of a process schedule $\mathcal{S}$ have terminated (either by C_i or A_i), then $\mathcal{S}$ is said to be *complete*; otherwise, $\mathcal{S}$ is called *partial*.

Note that since a process schedule is defined at the level of activities, it also includes activities of aborted processes. However, since the underlying subsystems guarantee both serializability and atomicity, activities returning with abort can be omitted in $\mathcal{S}$. The serializability property of subsystems allows to consider all activities of $\mathcal{S}$ that belong to the same subsystem (that correspond to transactions in the same subsystem) as totally ordered.

4.2 Commutativity and Equivalence

We consider all processes to be independent. Thus, the only possibility for information to flow between concurrent processes is when conflicting activities share some resources, that is when there is flow of information between the associated transactions of both activities. Hence, a common mechanism to verify the equivalence between schedules is commutativity. Following Vingralek et al. [1998], the notion of commutativity is defined using the return values of activities: we assume each activity a to provide a return value, which includes a description of a's outcome (success or failure, respectively). The return value of a process is a function of the return values of all its activities.

Definition 4.3 (*Commutativity*). Two activities $a_{i_k}, a_{j_l} \in \mathcal{A}^*$ *commute* if for all activity sequences α and ω 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 activity sequence $\langle \alpha\ a_{j_l}\ a_{i_k}\ \omega \rangle$.

Two activities are in conflict if they do not commute. Information about the commutativity behavior of activities is crucial for the process manager. Hence, a commutativity relation has to be available to it for scheduling purposes. This commutativity relation specifies, for each pair of activities from $\mathcal{A}^*$, whether or not they commute, thereby also considering predicates on the parameters associated which the invocation of both activities. Yet, a process manager is able to determine pairs of conflicting activities in a concrete context which is indicated by their actual parameters.

Activities may only conflict if they are executed in the same subsystem. This observation is based on the kind of layered architecture transactional process management addresses. As depicted in Figure 3, all subsystems are independent. In particular, the resources on which subsystems operate are pairwise disjoint. Hence, since the conflict behavior of each pair of activities coincides with the conflict behavior of the corresponding transactions, once the necessary information about commutativity of transactions in each subsystem is given, this can be used to derive the global commutativity relation at the process manager level, encompassing all activities of $\mathcal{A}^*$. Furthermore, this observation guarantees that return value commutativity can actually be applied as each subsystem is supposed to report the return values of its activities in a possibly individual but per-subsystem unique format.

In practical applications, a common assumption is that commutativity is perfect:

Definition 4.4 (*Perfect Commutativity* [*Vingralek et al. 1998*]). A commutativity relation is *perfect*, if for every two activities $a_i \in \mathcal{A}^*$ and $a_j \in \mathcal{A}^*$ the following holds: if a_i commutes with a_j, then a_i^{α} has to commute with a_j^{β} for all possible combinations of $\alpha, \beta \in \{-1, 1\}$ or, if a_i and a_j do not commute, then a_i^{α} does not commute with a_j^{β} for all possible combinations of $\alpha, \beta \in \{-1, 1\}$ with the exception of the null activity λ as an inverse activity commuting with every other activity.

Perfect commutativity requires inverses to be carefully defined. Consider, for example, the activities on electronic cash token that can be found in Example 2.1. Checking a token $t \in T$, $c : T \mapsto \{-1, 0\}$ verifies the validity and, in case of approval, sets a lock on this particular token and $c(t) = 0$, or, if the token is not approved, terminates without lock and $c(t) = -1$. Obviously, two checks do not commute. In case the inverse of a check activity would imprecisely be defined by a release activity $r : T \mapsto \{0\}$ which releases a lock on a token, if any exists, and otherwise does nothing, then perfect commutativity would not be given. However, with more carefully defined inverses that reflect the outcome of the regular activity, this is not the case. More realistically, the inverse $c^{-1}(t, x)$ of a check activity would be $c^{-1} : T \times \{-1, 0\} \mapsto \{-1, 0\}$ where x is the return value of the corresponding regular activity, with the following semantics: $c^{-1}(t, -1) = \lambda$ (actually, there is nothing to undo) while $c^{-1}(t, 0) = r(t)$.

With the notion of commutativity, *conflict-equivalence* is defined as follows:

Definition 4.5 (*Conflict Equivalence*). Two process schedules $\mathcal{S}_i = (\mathcal{P}_{\mathcal{S}_i}, \mathcal{A}_{\mathcal{S}_i}, \prec_{\mathcal{S}_i}, <_{\mathcal{S}_i})$ and $\mathcal{S}_k = (\mathcal{P}_{\mathcal{S}_k}, \mathcal{A}_{\mathcal{S}_k}, \prec_{\mathcal{S}_k}, <_{\mathcal{S}_k})$ are *conflict equivalent*, if they are defined over the same set of processes ($\mathcal{P}_{\mathcal{S}_i} = \mathcal{P}_{\mathcal{S}_k}$) and if all pairs of conflicting activities appear in the same order in the observed execution orders $<_{\mathcal{S}_i}$ and $<_{\mathcal{S}_k}$ of both process schedules.

Since two conflict equivalent process schedules, $\mathcal{S}_i$ and $\mathcal{S}_k$, are defined over the same set of processes ($\mathcal{P}_{\mathcal{S}_i} = \mathcal{P}_{\mathcal{S}_k}$), they also contain the same set of activities ($\mathcal{A}_{\mathcal{S}_i} = \mathcal{A}_{\mathcal{S}_k}$) and the same required orders ($\prec_{\mathcal{S}_i} = \prec_{\mathcal{S}_k}$), although their observed orders may differ.

4.3 Process-Serializability

Following the traditional approach [Bernstein et al. 1987], when discussing concurrency control, we disregard recovery operations. Thus, using the notion of conflict equivalence, a process schedule is process-serializable if its projection on all committed and active (running and completing) processes in which also all activities of abort subprocess executions are omitted is conflict equivalent to some serial process schedule.

Definition 4.6 (*Committed and Active Projection* (*CA*)). Let $\mathcal{S} = (\mathcal{P}_{\mathcal{S}}, \mathcal{A}_{\mathcal{S}}, \prec_{\mathcal{S}}, <_{\mathcal{S}})$ be a process schedule. The *committed and active projection* $CA(\mathcal{S})$ of $\mathcal{S}$ is a process schedule which is defined on all processes that are running, completing, and committed in $\mathcal{S}$. $CA(\mathcal{S})$ contains all activities from $\mathcal{A}_{\mathcal{S}}$ belonging to these processes, except for the activities of aborted or aborting subprocesses of completing or committed processes. Furthermore, $CA(\mathcal{S})$ contains the required orders for all these processes, projected on the activities of $CA(\mathcal{S})$. Analogously, the observed order of $CA(\mathcal{S})$, $<_{CA(\mathcal{S})}$, is the projection of $<_{\mathcal{S}}$ on its activities.

Definition 4.7 (*Process-Serializability* (*P-SR*)). A process schedule $\mathcal{S} = (\mathcal{P}_{\mathcal{S}}, \mathcal{A}_{\mathcal{S}}, \prec_{\mathcal{S}}, <_{\mathcal{S}})$ is *process-serializable* (*P-SR*) if its committed and active projection $CA(\mathcal{S})$ is conflict-equivalent to a serial process schedule $\mathcal{S}_{ser} = (\mathcal{P}_{\mathcal{S}_{ser}}, \mathcal{A}_{\mathcal{S}_{ser}}, \prec_{\mathcal{S}_{ser}}, <_{\mathcal{S}_{ser}})$.

Note that $CA(\mathcal{S})$ may contain both a smaller set of processes compared to $\mathcal{S}$ (analogously to the traditional consideration of serializability considering the committed projection of a schedule in which also a smaller set of transactions is present) and, for these processes, also a possibly smaller set of activities (namely, only the ones of (sub-)processes that are neither aborted nor aborting).

Example 4.1 (*Non-P-SR Execution*). Consider the two processes, P_1^1 and P_2^2, being executed in parallel. In Figure 7, the associated process programs

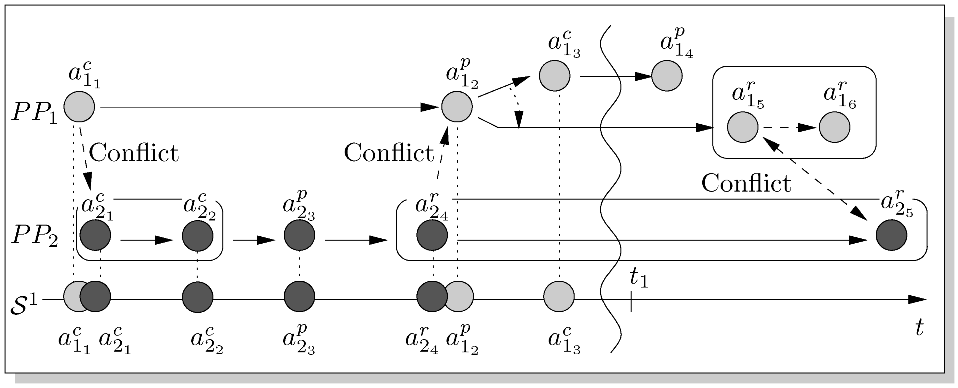


Fig. 7. Non-P-SR concurrent execution of process programs PP_1 and PP_2.

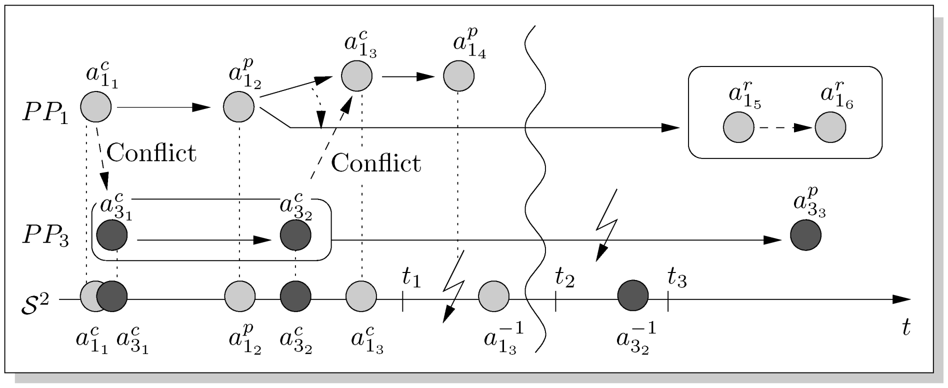


Fig. 8. Non-prefix-closed P-SR process schedule $\mathcal{S}^2_{t_2}$.

PP_1 and PP_2 and the process schedule $\mathcal{S}^1$ reflecting the concurrent execution of P_1 and P_2 are depicted. In total, three pairs of activities exist that do not commute: $(a^c_{1_1}, a^c_{2_1})$, $(a^p_{1_2}, a^r_{2_4})$, and $(a^r_{1_5}, a^r_{2_5})$. At time t_1, $\mathcal{S}^1_{t_1} = (\mathcal{P}_{\mathcal{S}^1_{t_1}}, \mathcal{A}_{\mathcal{S}^1_{t_1}}, \prec_{\mathcal{S}^1_{t_1}}, <_{\mathcal{S}^1_{t_1}})$ with the set of processes $\mathcal{P}_{\mathcal{S}^1_{t_1}} = \{P_1, P_2\}$, the set of activities $\mathcal{A}_{\mathcal{S}^1_{t_1}} = \{a^c_{1_1}, a^p_{1_2}, a^c_{1_3}, a^c_{2_1}, a^c_{2_2}, a^p_{2_3}, a^r_{2_4}\}$, the required order $\prec_{\mathcal{S}^1_{t_1}} = \{(a^c_{1_1} \prec_{\mathcal{S}^1_{t_1}} a^p_{1_2} \prec_{\mathcal{S}^1_{t_1}} a^c_{1_3}), (a^c_{2_1} \prec_{\mathcal{S}^1_{t_1}} a^c_{2_2} \prec_{\mathcal{S}^1_{t_1}} a^p_{2_3} \prec_{\mathcal{S}^1_{t_1}} a^r_{2_4})\}$ and the observed execution order $<_{\mathcal{S}^1_{t_1}} = \{(a^c_{1_1} <_{\mathcal{S}^1_{t_1}} a^c_{2_1} <_{\mathcal{S}^1_{t_1}} a^c_{2_2} <_{\mathcal{S}^1_{t_1}} a^p_{2_3} <_{\mathcal{S}^1_{t_1}} a^r_{2_4} <_{\mathcal{S}^1_{t_1}} a^p_{1_2} <_{\mathcal{S}^1_{t_1}} a^c_{1_3})\}$. However, $\mathcal{S}^1_{t_1}$ is not P-SR since no equivalent serial execution exists according to Definition 4.7 due to a conflict cycle between P_1 and P_2 caused by the orders $(a^c_{1_1} <_{\mathcal{S}^1_{t_1}} a^c_{2_1})$ and $(a^r_{2_4} <_{\mathcal{S}^1_{t_1}} a^p_{1_2})$. Since none of these activities corresponds to an aborted (sub-)process, they must be present in an equivalent serial process schedule which, however, would, in any case have a different ordering of one pair of conflicting activities.

Process-serializability is not prefix-closed as the following example shows.

Example 4.2 (*P-SR is not Prefix-Closed*). Consider the concurrent execution of P^1_1 and P^3_3 reflected in process schedule $\mathcal{S}^2$ as depicted in Figure 8.

Activities $a^c_{1_1}$ and $a^c_{3_1}$ as well as $a^c_{3_2}$ and $a^c_{1_3}$ do not commute (denoted by dashed arcs). At time t_1, $\mathcal{S}^2_{t_1}$ is not process-serializable: both P_1 and P_3 are active (P_3 running and P_1 completing and the subprocess including $a^c_{1_3}$ is neither aborting nor aborted) but the execution of both processes is not equivalent to any serial execution (due to cyclic conflicts). Assume now that either activity $a^p_{1_4}$ fails or that the process manager decides to abort the subprocess of P_1 containing $a^c_{1_3}$ and $a^p_{1_4}$ in order to break the conflict cycle. Both cases lead to the compensation of $a^c_{1_3}$ and the execution of the assured termination path of P_1. By changing the state of P_1's subprocess from running to aborting, the conflict in which $a^c_{1_3}$ is involved will be neglected. Yet, the compensating activity $a^{-1}_{1_3}$ undoes the effects of its regular activity. At time t_2 after $a^{-1}_{1_3}$ has committed, process schedule $\mathcal{S}^2_{t_2}$ is P-SR ($CA(\mathcal{S}^2_{t_2})$ is conflict equivalent to a serial execution of P_1—without its aborted subprocess—followed by P_3).

In traditional transactions where only total backward recovery is allowed, this phenomenon cannot occur (at least when commutativity between activities is perfect) since whenever a cycle in a conflict graph exists, there is no way for all involved transactions to finally commit successfully. In process executions where partial backward recovery combined with alternative executions is possible, this may however be the case.

Let PSG($\mathcal{S}$) be a *process serialization graph* of a process schedule $\mathcal{S}$ that contains a node for each running, completing, and committed process and a directed edge from P_i to P_j whenever a pair of conflicting activities $a_i \in P_i$ and $a_j \in P_j$ exists in $\mathcal{S}$ with the observed order $a_i < a_j$. Whenever a subprocess is aborted, all edges introduced by activities of this subprocess (i.e., all conflicts in which an activity of an aborted subprocess is involved) are removed from PSG($\mathcal{S}$).

THEOREM 4.8 [PROCESS SERIALIZATION GRAPH AND P-SR]. *A process schedule $\mathcal{S}$ is P-SR if and only if its process serialization graph PSG($\mathcal{S}$) is acyclic.*

PROOF OF THEOREM 4.8

IF: Let PSG($\mathcal{S}$) be an acyclic process serialization graph of process schedule $\mathcal{S}$ and let $\mathcal{S}'$ be a process schedule defined over the same set of processes ($\mathcal{P}_{\mathcal{S}} = \mathcal{P}_{\mathcal{S}'}$), containing the same set of activities as $\mathcal{S}$'s committed and active projection $CA(\mathcal{S})$, and encompassing a total order on all processes that is compatible with the orders encompassed in PSG($\mathcal{S}$). Process schedule $\mathcal{S}'$ can be derived from $\mathcal{S}$ by applying a topological sort. Since all pairs of conflicting activities appear in the same order in $CA(\mathcal{S})$ and $\mathcal{S}'$ (thus, they are conflict equivalent) and since $\mathcal{S}'$ is a serial process schedule, $\mathcal{S}$ is P-SR.

ONLY IF: Let $\mathcal{S}$ be a P-SR process schedule, that is, a serial process schedule $\mathcal{S}'$ exists which is conflict equivalent to $CA(\mathcal{S})$. Assume that the process serialization graph PSG($\mathcal{S}$) of $\mathcal{S}$ contains a cycle $P_i \rightarrow P_{i+1} \rightarrow \cdots \rightarrow P_{i+n} \rightarrow P_i$. Since PSG($\mathcal{S}$) does only contain edges where activities are involved that also appear in the committed and active projection of $\mathcal{S}$, all these conflicts have to be present in $\mathcal{S}'$. In the serial process schedule $\mathcal{S}$, therefore P_i has to precede P_{i+1} and P_{i+1}, in turn, has also to precede P_i which obviously leads to a contradiction of the initial assumption that $\mathcal{S}$ is a P-SR process. $\square$

For convenience, process-serializability can be tested using the standard serialization graph $SG(\mathcal{S})$ that is defined as the process serialization graph (each active and committed process corresponds to a node, each conflict leads to an edge in $SG(\mathcal{S})$). In contrast to the process serialization graph, however, $SG(\mathcal{S})$ additionally contains all edges induced by activities of aborted or aborting subprocesses of active or committed processes. Therefore, the process serialization graph PSG($\mathcal{S}$) of a process schedule $\mathcal{S}$ can be considered as the serialization graph, restricted to the committed and active projection $CA(\mathcal{S})$ of $\mathcal{S}$, that is, PSG($\mathcal{S}$) = $SG(CA(\mathcal{S}))$.

THEOREM 4.9 [SERIALIZATION GRAPH AND P-SR]. *A process schedule $\mathcal{S}$ is P-SR if its serialization graph $SG(\mathcal{S})$ is acyclic.*

PROOF OF THEOREM 4.9. Let $SG(\mathcal{S})$ be an acyclic serialization graph of process schedule $\mathcal{S}$. The committed and active projection, $CA(\mathcal{S})$, of $\mathcal{S}$ is obtained from $\mathcal{S}$ by dropping all activities of aborted and aborting subprocesses. Therefore, the serialization graph of $CA(\mathcal{S})$ is still acyclic. Since $SG(CA(\mathcal{S}))$ = PSG($\mathcal{S}$), the process serialization graph of $\mathcal{S}$ is acyclic, which, in accordance with Theorem 4.8, implies that $\mathcal{S}$ is P-SR. □

Note that—in contrast to the traditional transaction model—the converse of Theorem 4.9 does not hold. A process schedule may be P-SR albeit its serialization graph contains a cycle. This is the case since the serialization graph considers all active and committed processes and does not omit conflicts in which activities of aborted or aborting subprocesses are involved.

Although the acyclicity of $SG(\mathcal{S})$ is a stronger criterion than P-SR and allows a subset of all P-SR process schedules only (called SG-P-SR, encompassing all process schedules whose serialization graphs are acyclic), it can be used for dynamic scheduling since the abort of subprocesses does not require the examination of all edges of the graph and the deletion of those in which activities of these subprocesses are involved (which would, however, be the case for PSG). In contrast to PSG, it is not possible to successfully terminate a set of processes (i.e., to commit each of these processes) which induce a cycle in SG such that this cycle disappears after completion of these processes. Therefore, a process manager has to guarantee that the serialization graph is at any point in time free of cycles. This leads to a further restriction of SG-P-SR, namely a prefix-closed variant, called P-SG-P-SR. Note that SG-P-SR itself is not prefix-closed[3]: assume that a cycle in the serialization graph $SG(\mathcal{S}')$ of some process schedule $\mathcal{S}'$ exists. Then, the abort of at least one process involved in this cycle makes the associated node disappear and may lead to a process schedule $\mathcal{S}$ with prefix $\mathcal{S}'$ that meets SG-P-SR.

[3]This is not the case in the traditional transaction model where serializability is based on the notion of committed projection of a schedule. Whenever a schedule is serializable, so is also each prefix. The restriction to the committed projection of a schedule is possible in the traditional model since each active transaction can be aborted, which is, however, not the case for active processes once they have committed a pivot.

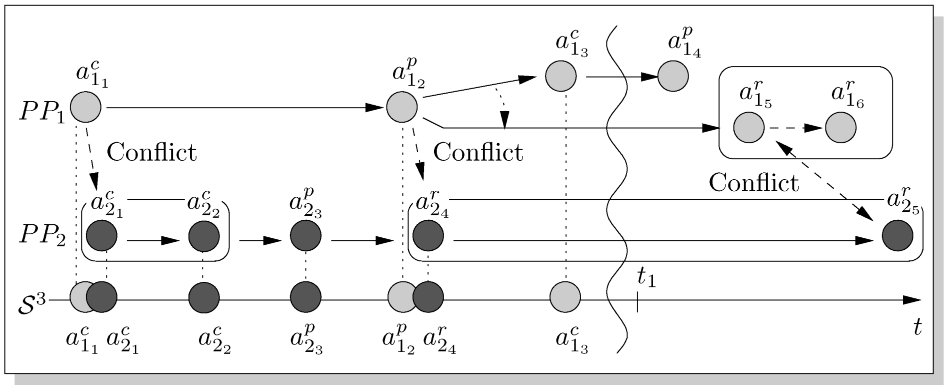

PP_1
PP_2
$\mathcal{S}^3$
Conflict
Conflict
Conflict
t_1
t
$a^c_{1_1}$ $a^p_{1_2}$ $a^c_{1_3}$ $a^p_{1_4}$ $a^r_{1_5}$ $a^r_{1_6}$
$a^c_{2_1}$ $a^c_{2_2}$ $a^p_{2_3}$ $a^r_{2_4}$ $a^r_{2_5}$
$a^c_{1_1}$ $a^c_{2_1}$ $a^c_{2_2}$ $a^p_{2_3}$ $a^p_{1_2}$ $a^r_{2_4}$ $a^c_{1_3}$

Fig. 9. Process-serializable concurrent execution of process programs PP_1 and PP_2.

Example 4.2 (*Revisited*). As we have already seen, the process serialization graph of process schedule $\mathcal{S}^2$ contains, at time t_1, the cycle $P_1 \leftrightarrows P_3$; hence, $\mathcal{S}^2_{t_1}$ is not P-SR. Therefore, this cycle also exists in SG($\mathcal{S}^2_{t_1}$) such that $\mathcal{S}^2_{t_1}$ is also not SG-P-SR. However, the failure of $a^p_{1_4}$ leads to the abort of the subprocess including $a^c_{1_3}$ and $a^p_{1_4}$, which, in turn, leads to the deletion of the edge induced by $a^c_{3_2} <_{\mathcal{S}^2} a^c_{1_3}$ and removes the cycle in PSG($\mathcal{S}^2_{t_2}$). Note that the abort of P_1's subprocess does not make P_1 to completely disappear in PSG($\mathcal{S}^2_{t_2}$) since $a^p_{1_2}$, the primary pivot, has already successfully committed such that P_1 is completing. While the abort of $a^p_{1_4}$ has made the edge corresponding to the conflict pair $a^c_{3_2} <_{\mathcal{S}^2} a^c_{1_3}$ disappear in PSG($\mathcal{S}^2_{t_2}$), it is still present in SG($\mathcal{S}^2_{t_2}$) such that $\mathcal{S}^2_{t_2}$, even though it is P-SR, does not meet the SG-P-SR criterion. Assume further that process P_3 is aborted for some reason after the execution of $a^{-1}_{1_3}$. Therefore, at time t_3, P_3 is aborting and activity $a^{-1}_{3_2}$ has been executed for recovery purposes. Thus, P_3 has disappeared from the committed and active projection of $\mathcal{S}^2_{t_3}$ and the corresponding node is removed from both PSG and SG. Hence, SG($\mathcal{S}^2_{t_3}$) contains only one process, namely P_1, such that SG-P-SR trivially holds for $\mathcal{S}^2_{t_3}$. Since this is not the case for its prefix $\mathcal{S}^2_{t_2}$, $\mathcal{S}^2_{t_3}$ is not P-SG-P-SR.

Example 4.3 (*Correct P-SR Execution*). Consider again the two process programs PP_1 and PP_2, now executed concurrently by processes P_1 and P_2 in process schedule $\mathcal{S}^3$ as depicted in Figure 9. At time t_1, the process schedule $\mathcal{S}^3_{t_1}$ is P-SR. Again, all conflicting activities do not belong to any aborted (sub-)process and must be present in an equivalent serial schedule. The serial execution of all P_1 activities of $\mathcal{S}^3_{t_1}$ followed by all P_2 activities would be conflict equivalent to the execution of process schedule $\mathcal{S}^3_{t_1}$.

While the required order of each process is given and has to be respected in a process schedule, the process manager is responsible for correctly ordering conflicting activities. By this, we mean that the process manager (PM) has to guarantee process-serializability of process schedules (while taking into account all the information available to it). As already mentioned, the process manager has the possibility to impose a temporal order between conflicting activities, or it can submit activities concurrently while at the same time specifying the order in which the associated transactions have to be serialized in the underlying subsystem. In both cases, commit-order serializability protocols implemented in the subsystems guarantee compliance with this order. Note, that when activities do not correspond to transactions in the traditional sense (as we do assume here) but are implemented again by process programs, the notion of commit-order serializability needs to be extended. In this case, for each process in a subsystem, its commit and its pivot activities have to be treated the same way. Each pair of conflicting activities of any two subsystem processes then requires the next points of no return of both processes (be it either a pivot or the process' commit) to follow the imposed serialization order.

The failure of retriable activities now may lead to a special treatment of other activities. Suppose two activities $a^r_{i_k}$ and a_{j_l} are executed concurrently within the same subsystem with a serialization order given by the process manager requiring $a^r_{i_k}$ to be serialized before a_{j_l}. If the local transaction T_{i_k} corresponding to $a^r_{i_k}$ fails after some operations of T_{i_k} have already been executed, then, in general, the local transaction T_{j_l} (which corresponds to activity a_{j_l}) running in parallel to T_{i_k} (with respect to the given weak order) has to be aborted. However, as this is not due to a failure of T_{j_l} (note that ACA is guaranteed by each subsystem such that the abort of T_{i_k} does not influence T_{j_l}), it must not lead to an exception of P_j leading to an other alternative. Therefore, after T_{i_k} is restarted, T_{j_l} has also to be restarted within the subsystem, hence guaranteeing compliance with the serialization order imposed by the process manager.

Therefore, the PM extends a classical transaction scheduler in three ways:

(1) it exploits information about the properties of all activities (compensatable, pivot, or retriable) and thus, also about the different states of active processes (running or completing),
(2) it considers for each process P^k_i the alternative execution paths defined within the process program PP_k,
(3) it respects the required order for each process as defined within the corresponding process program and explicitly imposes appropriate weak orders between conflicting activities.

4.4 Process-Recoverability

Process-Recoverability addresses the possibility to abort a subset of running processes correctly, even in the presence of concurrency. However, avoiding cascading aborts is too strong at the process level since—in the case of semantical rich activities where a distinction between read and write access to data is hardly possible—this would degenerate to strictness or even rigorousness [Breitbart et al. 1991]. Yet, for each arbitrary set $R_{\mathcal{S}}$ of running processes of a process schedule $\mathcal{S}$, a superset $R^*_{\mathcal{S}}$, also of running processes, has to exist such that all processes of $R^*_{\mathcal{S}}$ together with all aborting processes can be aborted correctly without affecting other processes. Note that one cannot require that all partial processes can be aborted, since some of them may have already performed a pivot—an activity representing a point of no return of the process.

That is, for $\mathcal{S}$ there must be a schedule $\mathcal{S}'$ which is defined over the same set of processes and in which all processes of $R^*_{\mathcal{S}}$ that are running in $\mathcal{S}$ and all aborting processes of $\mathcal{S}$ do not leave any effects. Furthermore, the return values of all other activities (not belonging to aborted processes in $\mathcal{S}$ or processes of $R^*_{\mathcal{S}}$) are the same in both process schedules. Since the return values of processes is a function on the return values of all its activities, the latter criterion guarantees the correspondence of the return values of all committed processes in both process schedules. The fact that the return values of all other activities, especially those of completing processes are left unchanged is important because these processes must be able to commit successfully. Since the effects of all aborting processes and of all processes of $R^*_{\mathcal{S}}$ have to be eliminated, correspondence between the final states of $\mathcal{S}$ and $\mathcal{S}'$ is not required.

In the presence of concurrency, care is needed with situations where the execution of a sequence of activities of a process P_j is affected by the compensation of an activity a_i of another process P_i when the subprocess, in which a_i has been executed, is running and is now a candidate for being aborted. For, if P_j is running, then we can decide to abort it as well. But, if P_j is completing (or has already committed), this is impossible. Note that once P_j has committed a pivot successfully, then it cannot be aborted. Therefore, process-recoverability must encompass the restriction that no completing process must be dependent on a running process in the sense that the abort of a running process does imply also the abort of a completing process which, according to the state diagram depicted in Figure 5, is not possible.

To this end, we have to use the notion of abort dependency to formally specify the situations possibly leading to a violation of process-recoverability. In short, there is an abort dependency between two processes P_i and P_j imposed by activities $a^c_{i_k}$ and a_{j_m} with $a^c_{i_k} <_{\mathcal{S}} a_{j_m}$ when the execution of a_{j_m} hinders the compensation of $a^c_{i_k}$.

Definition 4.10 (*Abort Dependency*). An *abort dependency* between two processes P_i and P_j, imposed by activities $a_{i_k} \in P_i$ and $a_{j_m} \in P_j$, exists in a process schedule $\mathcal{S} = (\mathcal{P}_{\mathcal{S}}, \mathcal{A}_{\mathcal{S}}, \prec_{\mathcal{S}}, <_{\mathcal{S}})$ if the following properties hold:

(1) a_{i_k} precedes a_{j_m} in $\mathcal{S}$, that is $a_{i_k} <_{\mathcal{S}} a_{j_m}$
(2) a_{i_k} is compensatable
(3) a_{j_m} is neither preceded in $\mathcal{S}$ by $a^{-1}_{i_k}$ nor by $a^*_{i_k}$, that is $a^{-1}_{i_k} \not<_{\mathcal{S}} a_{j_m}$ and $a^*_{i_k} \not<_{\mathcal{S}} a_{j_m}$ where $a^*_{i_k}$ is the next point of no return of P_i succeeding a_{i_k} (this can either be some pivot activity $a^p_{i_q}$ with $a_{i_k} \prec_{\mathcal{S}} a^p_{i_q}$ or the commit C_i of P_i).
(4) In $\mathcal{S}$, an activity a_{j_l} of P_j exists which precedes a_{j_m} ($a_{j_l} \prec_{\mathcal{S}} a_{j_m}$), which conflicts with a_{i_k} and which succeeds a_{i_k} in $\mathcal{S}$, that is $a_{i_k} <_{\mathcal{S}} a_{j_l}$
(5) a_{j_m} conflicts with the inverse activity $a^{-1}_{i_k}$ of a_{i_k}.

Given the conditions of Definition 4.10, in a situation where the sequence $a_{i_k} <_{\mathcal{S}} a_{j_l} <_{\mathcal{S}} a_{j_m} <_{\mathcal{S}} a^{-1}_{i_k}$ appears in a process schedule $\mathcal{S}$, we cannot bring a_{i_k} and $a^{-1}_{i_k}$ together such that the latter one correctly undoes the effects of a_{i_k} (that is, that both activities can be cancelled). Note that, if perfect commutativity holds, then we can take a_{j_l} and a_{j_m} to be the same. This is the case, for instance, in a read-from dependency: If a_{i_k} writes some data that a_{j_l}

reads, then, $a_{i_k}^{-1}$ is also a write and a_{j_l} conflicts with both a_{i_k} and $a_{i_k}^{-1}$. However, due to the semantically rich nature of activities, a dependency may exist in other situations as well. Essentially, abort dependencies are imposed between individual processes—like reads-from dependencies between transactions—while transaction termination dependencies (e.g., Türker et al. [2000]) consider dependencies between subtransactions of a single global transaction.

In the traditional model where each activity is compensatable, the requirement that all running transactions must be able to abort is captured by the notion of recoverability. As we have to deal with two different states of processes determining the way recovery has to be performed, we have to adapt the notion of recoverability to the structure of transactional processes. This leads to the notion of *process-recoverability*. More formally,

Definition 4.11 (*Process-Recoverability* (*P-RC*)). A process schedule $\mathcal{S} = (\mathcal{P}_\mathcal{S}, \mathcal{A}_\mathcal{S}, \prec_\mathcal{S}, <_\mathcal{S})$ is *process-recoverable* (*P-RC*), if for each pair of activities, $a_{i_k}^c$ and a_{j_m} of $\mathcal{S}$ with $a_{i_k}^c <_\mathcal{S} a_{j_m}$ imposing an abort dependency between P_i and P_j, the following holds:

(1) If activity a_{j_m} is compensatable ($a_{j_m}^c$) and when a_j^* is in $\mathcal{S}$, then the following ordering has to exist in $\mathcal{S}$: $a_i^* <_\mathcal{S} a_j^*$ where a_i^* is the next point of no return succeeding $a_{i_k}^c$ with respect to P_i's required order $\prec_i$ (this may either be the commit C_i of P_i or a pivot; when P_i is running, then it will be the primary pivot of P_i; otherwise, the pivot of one of P_i's subprocesses) and 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 process schedule $\mathcal{S}$: $a_i^* <_\mathcal{S} a_{j_m}$ where a_i^* is the next point of no return succeeding $a_{i_k}^c$ with respect to $\prec_i$ (this may either be the commit C_i of P_i or a pivot; when P_i is running, then it will be the primary pivot of P_i, otherwise the pivot of one of P_i's subprocesses).

Note that, analogously to the notion of abort dependency being a generalization of the read-from dependency, the traditional notion of recoverability is a special case of Definition 4.11. When no pivot activities exist as in the traditional case, then, in accordance with 4.11.1, only an order between C_i and C_j with $C_i <_\mathcal{S} C_j$ has to be imposed. The semantics of "quasicommit" of pivot activities is also included in this definition since pivots are treated in the same way as the commit of a process. Obviously, the notion of process-recoverability is weaker than the ACA requirement which would require the total absence of abort dependencies in a process schedule.

For process-recoverability, we do not explicitly require the total absence of abort dependencies. This absence would, in the case of perfect commutativity, for instance prevent to consecutively execute two conflicting compensatable activities a_i^c and a_j^c of two running processes P_i and P_j in a process schedule $\mathcal{S}$ since either a_i^c would have to be compensated prior to a_j^c or a_i^* would have to be executed prior to a_j^c (where a_i^* is either C_i or a primary pivot of P_i). But, this is too strict (it would, in fact, correspond to the notion of rigorousness [Breitbart et al. 1991], applied to transactional processes) since in the case of $a_i^c <_\mathcal{S} a_j^c$, an

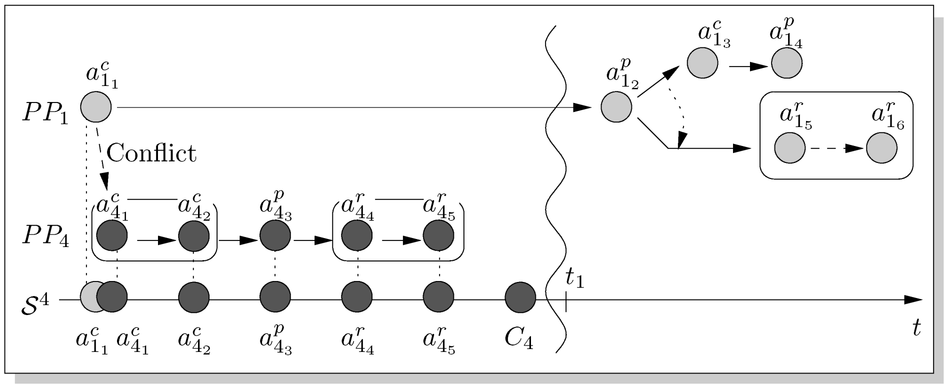


Fig. 10. Process Schedule $\mathcal{S}^4$ violating the P-RC property.

abort of P_i would just require to abort P_j as well (which is possible when the requirements of P-RC are met in $\mathcal{S}$).

In most practical applications, commutativity is perfect. This can be used to enforce P-RC since abort dependencies correspond to pairs of conflicting activities a_{i_k} and a_{j_l} where the first activity to execute is compensatable. In order to enforce P-RC, the same serialization graph used for SG-P-SR and P-SG-P-SR, respectively, can be exploited. In terms of this graph, violations of P-RC may occur because of two reasons: firstly, by introducing an edge corresponding to an abort dependency and secondly, by a state change of a process. The process manager allows edges to be introduced in the serialization graph (due to a conflict $a_{i_k} <_{\mathcal{S}} a_{j_l}$) between two nodes that both correspond to running processes as well as edges from a node corresponding to a completing process to a running process (the first case is not critical since it can be resolved by a joint abort, the second case also when the completing process is in a running subprocess). Edges from running processes to completing processes are prohibited. Edges between two completing processes may in some cases be allowed when a_{i_k} is itself a pivot or when it is succeeded by a pivot in $\mathcal{S}$, but in general, this kind of edges has to be dealt with care. Furthermore, the processes associated with nodes may only change their state from running to completing or from running to committed when they have no incoming edges originating from nodes corresponding to running processes (i.e., they are not dependent on running processes). Incoming edges originating from nodes corresponding to completing processes may be allowed but must also be dealt with care, especially in the case where the source node is in a running subprocess. Note that these mechanisms guarantee P-RC although they may rule out certain schedules that are correct with respect to P-RC. More details of a protocol dynamically providing P-SG-P-SR and P-RC are discussed in Section 5.

Example 4.4 (*Non-P-RC Execution*). Consider process schedule $\mathcal{S}^4$ depicted in Figure 10 reflecting the execution of two processes P^1_1 and P^4_4 and assume further that commutativity is perfect in this example. At t_1, process P_4 is committed while P_1 is running. An abort dependency exists between P_1

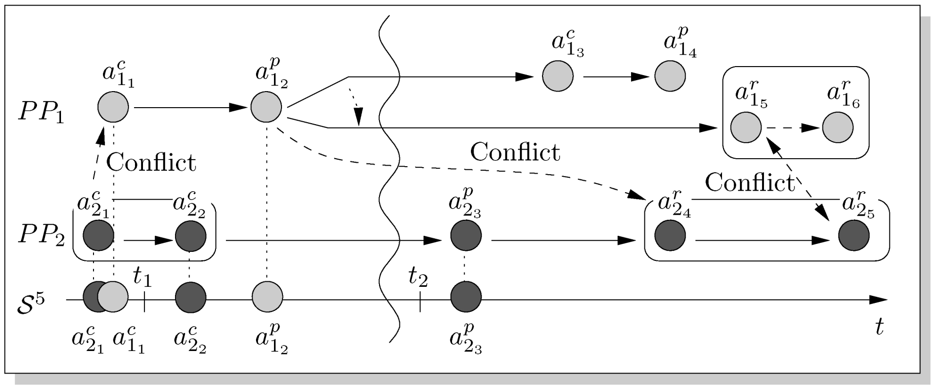


Fig. 11. Violation of P-RC by a state change of P_1 from running to completing in $\mathcal{S}^5_{t_2}$.

and P_4, imposed by $a^c_{1_1}$ and $a^c_{4_1}$. According to Definition 4.11, P-RC requires the following order in $\mathcal{S}^4$: $a^p_{1_2} <_{\mathcal{S}^4} a^p_{4_3}$ and, due to $a^p_{4_3} \prec_{P_4} C_4$ also $a^p_{1_2} <_{\mathcal{S}^4} C_4$ which, however, is violated in $\mathcal{S}^4_{t_1}$. Therefore, in an execution where P_1 does not leave any effects (where activity $a^c_{1_1}$ does not appear), the return value of P_4 would not be the same as it is in $\mathcal{S}^4_{t_1}$. Thus, $\mathcal{S}^4_{t_1}$ is not process-recoverable.

Example 4.5 (*P-RC and Abort Dependencies*). Consider the execution of P_1 and P_2 at time t_2 in process schedule $\mathcal{S}^5$ as depicted in Figure 11. Process P_1 is completing and P_2 is running. An abort dependency between P_2 and P_1 exists by $a^c_{2_1} <_{\mathcal{S}^5} a^c_{1_1}$. At time t_1 when this dependency has been introduced, it was allowed since both processes have been running. However, by executing $a^p_{1_2}$, P_1 changed to completing making the abort dependency a disallowed one because the order $a^p_{2_3} <_{\mathcal{S}^5} a^p_{1_2}$ required by P-RC was violated. An abort of P_2 at time t_2 could no longer be treated correctly by a cascading abort including also P_1. Therefore, process schedule $\mathcal{S}^5_{t_2}$ is not process-recoverable.

The analysis of process-serializability has shown that although a given process schedule $\mathcal{S}$ may fulfill the P-SR property, there might exist a prefix $\mathcal{S}'$ of $\mathcal{S}$ that is not process-serializable. This phenomenon cannot be found in process-recoverability which is subject of the following lemma:

LEMMA 4.12 (P-RC & PREFIXES). *Let $\mathcal{S}$ be a P-RC process schedule. Then, each prefix $\mathcal{S}'$ of $\mathcal{S}$ is also process-recoverable.*

PROOF OF LEMMA 4.12. Let $\mathcal{S}$ be a P-RC process schedule and let $\mathcal{S}'$ be a prefix of $\mathcal{S}$ that is not P-RC. Since $\mathcal{S}'$ is not P-RC, there must exist at least one abort dependency between a pair of processes (P_i, P_j) whose constraints on the subsequent points of no return are not met. Let (a_{i_k}, a_{j_m}) be the pair of activities imposing such an abort dependency in $\mathcal{S}'$ and let a_{j_p} be the next point of no return succeeding a_{j_m} (in case a_{j_m} is not compensatable, then $a_{j_m} = a_{j_p}$). Therefore, in $\mathcal{S}'$, the following orders have to exist: $a_{i_k} <_{\mathcal{S}'} a_{j_m} <_{\mathcal{S}'} a_{j_p}$ and $a^*_{i_k} \not<_{\mathcal{S}'} a_{j_p}$ where $a^*_{i_k}$ is the next point of no return succeeding a_{i_k}. But this order must also be present in $\mathcal{S}$, which will then, in turn, also not be P-RC. □

4.5 Process-Reducibility

So far, we have addressed isolation without considering atomicity (P-SR), and the possibility to abort running processes (P-RC). What is missing is a unified criterion that jointly addresses both problems and that allows to check whether the abort of (sub-)processes in the presence of concurrency is possible or not. This includes the property that no conflicting activity a_{j_m} must be executed between a regular activity a_{i_k} and its compensation $a_{i_k}^{-1}$ except for the case where the compensation of a_{j_m} also appears between a_{i_k} and $a_{i_k}^{-1}$.

To this end, reduction techniques based on the permutation and cancellation of activities can be applied. Recalling the notion of commutativity, it is obvious that two consecutive activities of different processes can be permuted in a process schedule $\mathcal{S}$ if they do commute; hence, this permutation does neither affect the final state achieved after $\mathcal{S}$ nor the return values of all transactional processes. Additionally, also the elimination of two consecutive activities does neither influence the return values nor the final state when they together form an effect-free sequence. More formally,

Definition 4.13 (*Reducible Process Schedule* (*P-RED*)). A process schedule $\mathcal{S} = (\mathcal{P_S},\ \mathcal{A_S},\ \prec_\mathcal{S},\ <_\mathcal{S})$ is reducible (P-RED) if it can be transformed to a serial process schedule $\mathcal{S} = (\mathcal{P}_{\underline{\mathcal{S}}}, \mathcal{A}_{\underline{\mathcal{S}}}, \prec_{\underline{\mathcal{S}}}, <_{\underline{\mathcal{S}}})$ 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} belong to different processes ($i \neq j$) and they do commute, or they belong to the same process ($i = j$) and are not ordered in $\prec_\mathcal{S}$, that is, the corresponding process program allows an unrestricted parallel execution of both activities
   - (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 $a_{i_k}, a_{i_k}^{-1}$ can be removed from $\underline{\mathcal{S}}$.

Reducibility for process schedules is similar to the reduction applied in the traditional unified theory of concurrency control and recovery except for one major difference. Prior to the application of reduction techniques, the unified theory requires the expansion of a schedule where each running transaction is treated as aborted. This expansion leads to a complete schedule where each transaction either has committed or aborted. Reduction for process schedules does not require expansion (since the notion of process schedule already considers abort related activities and process-recoverability does not require all active processes to abort).

P-RED is not prefix-closed (see, e.g., process schedule $\mathcal{S}^2$ of Example 4.2). A restriction of P-RED which requires that each prefix of a process schedule is P-RED leads to the notion of prefix-process-reducibility, P-P-RED.

In order to include aborting (sub-)processes, and in analogy to reduction in the traditional unified theory of concurrency control and recovery which is only applied after a given schedule has been expanded, processes have to terminate in order to ensure that all compensating activities are considered in the reduction phase.

4.6 Correct Termination

In addition to the previous ideas addressing noncomplete process schedules, all completing processes must be able to commit correctly. Therefore, reduction techniques (and thus, the P-RED criterion) have to be applied to the completed process schedule, $\mathcal{C}(\mathcal{S})$, of a process schedule $\mathcal{S}$. This leads to the notion of *correct termination* (*CT*). In the completed process schedule, all aborting processes of $\mathcal{S}$ and all running processes of $R^*_{\mathcal{S}}$ for which an abort is requested are aborted and all completing processes of $\mathcal{S}$ have committed. Correct termination has to guarantee that it is possible to perform all aborts and all completions correctly, even in the presence of concurrency.

While guaranteed termination addresses the well-formed structure and inherent correctness of single processes, correct termination addresses the correctness of complete multiprocess executions. The first step in formulating correct termination is that a given process schedule is completed:

Definition 4.14 (*Completed Process Schedule*). Let $\mathcal{S} = (\mathcal{P}_{\mathcal{S}},\ \mathcal{A}_{\mathcal{S}},\ \prec_{\mathcal{S}},\ <_{\mathcal{S}})$ be a process schedule. The *completed process schedule* $\mathcal{C}(\mathcal{S})$ is a process schedule defined over the same set of processes ($\mathcal{P}_{\mathcal{S}} = \mathcal{P}_{\mathcal{C}(\mathcal{S})}$) with

(1) Each activity $a \in \mathcal{A}_{\mathcal{S}}$ is also in $\mathcal{A}_{\mathcal{C}(\mathcal{S})}$, that is $A_{\mathcal{S}} \subseteq \mathcal{A}_{\mathcal{C}(\mathcal{S})}$

(2) $\mathcal{S}$ is a prefix of its completed process schedule $\mathcal{C}(\mathcal{S})$. That is, for each pair of activities a, a^* with $a \in \mathcal{A}_{\mathcal{S}}$ and $a^* \in \mathcal{A}_{\mathcal{C}(\mathcal{S})} \backslash \mathcal{A}_{\mathcal{S}}$, the following has to hold: $a <_{\mathcal{C}(\mathcal{S})} a^*$

(3) $\mathcal{C}(\mathcal{S})$ is complete. That is, in $\mathcal{C}(\mathcal{S})$, all processes that are aborting in $\mathcal{S}$ are aborted and all completing processes of $\mathcal{S}$ are committed in $\mathcal{C}(\mathcal{S})$. Furthermore, for each arbitrary set of running processes $R_{\mathcal{S}}$ of $\mathcal{S}$, there must be a set $R^*_{\mathcal{S}}$ of running processes with $R_{\mathcal{S}} \subseteq R^*_{\mathcal{S}} \subseteq \mathcal{R}_{\mathcal{S}}$, $\mathcal{R}_{\mathcal{S}}$ being the set of all of $\mathcal{S}$'s running processes, where all processes of $R^*_{\mathcal{S}}$ are aborted in $\mathcal{C}(\mathcal{S})$ and all processes of $\mathcal{R}_{\mathcal{S}} \backslash R^*_{\mathcal{S}}$ are committed in $\mathcal{C}(\mathcal{S})$.

Once a process schedule $\mathcal{S}$ is completed to $\mathcal{C}(\mathcal{S})$, correct termination requires the existence of a schedule $\underline{\mathcal{C}(\mathcal{S})}$ that is serial on all processes and that is equivalent to $\mathcal{C}(\mathcal{S})$ with respect to the return values of all processes and the initial and final state. Note that this is more than process-serializability of $\underline{\mathcal{C}(\mathcal{S})}$. In P-SR, only running, completing and committed processes are considered (in this case, since $\underline{\mathcal{C}(\mathcal{S})}$ is complete, this would be only committed processes), but aborted processes are ignored (including aborted subprocesses of completing processes). However, an important aspect of correct termination is to address also the correct abort of processes, that is, the correct execution of compensating activities in the presence of concurrency, which requires $\underline{\mathcal{C}(\mathcal{S})}$ to be serial on all processes. Correct termination therefore requires a completed process schedule to be reducible. If this is the case, it is guaranteed that all aborted

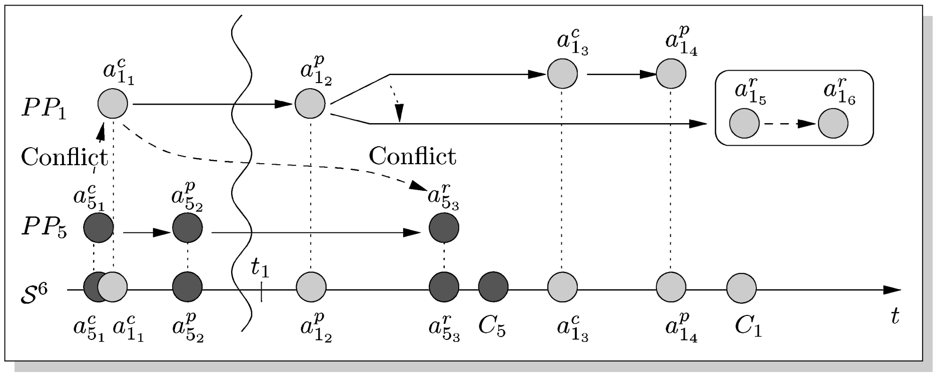


Fig. 12. Incorrect completion $\mathcal{C}(\mathcal{S}^6_{t_1})$ of P-SR & P-RC process schedule $\mathcal{S}^6_{t_1}$.

(sub-)processes do not leave any effect since all their regular activities together with the corresponding compensation activities will have disappeared.

Definition 4.15 (*Correct Termination* (*CT*)). A complete process schedule $\mathcal{C}(\mathcal{S})$ has *correct termination* (*CT*) property if it is reducible (P-RED).

Example 4.6 (*Non-CT Execution*). Consider the concurrent execution of the process programs PP_1 and PP_5 by processes P^1_1 and P^5_5, reflected in process schedule $\mathcal{S}^6$ (see Figure 12). At time t_1, $\mathcal{S}^6_{t_1}$ is correct with respect to P-RC (P_5 is already completing, that is, $a^c_{5_1}$ will never be compensated) and P-RED (thus, also P-SR) holds. However, the completion $\mathcal{C}(\mathcal{S}^6_{t_1})$ of $\mathcal{S}^6_{t_1}$ does not have the CT property. The execution of $a^r_{5_3}$, which is inevitably required to complete P_5, introduces cyclic dependencies. Since in the meanwhile also P_1 has changed to completing, this conflict cycle cannot be resolved. Therefore, the only possibility to successfully complete $\mathcal{S}^6_{t_1}$ correctly would be to abort P_1 which would then impose activity $a^{-1}_{1_1}$ to be executed. After the commit of $a^{-1}_{1_1}$, P_5 would be able to proceed forward by executing $a^r_{5_3}$ and to finally terminate correctly.

Example 4.7 (*Correct CT Execution*). A CT execution of process programs PP_1 and PP_2 is given by process schedule $\mathcal{S}^7$ depicted in Figure 13. At time t_1, $\mathcal{S}^7_{t_1}$ is both P-RED and P-RC (the constraints imposed by the only abort dependency $a^c_{1_1} <_{\mathcal{S}^7} a^c_{2_1}$ are met). Furthermore, although conflicts exist between the activities of the completion of both processes, the completed schedule $\mathcal{C}(\mathcal{S}^7)$ is correct since it is equivalent to a serial schedule where P^1_1 precedes P^2_2. Note that completion of $\mathcal{S}^9_{t_1}$ does not require P_2 to abort although it is running in $\mathcal{S}^9_{t_1}$. However, once P_2 changes its state from running to completing, it must be ensured that it will commit correctly. In process schedule $\mathcal{C}(\mathcal{S}^7)$, this is trivially the case since this state change by the execution of $a^p_{2_3}$ is performed after the commit of P_1, $C_1 <_{\mathcal{C}(\mathcal{S}^7)} a^p_{2_3}$ and since the serialization order in the reduced process schedule $\underline{\mathcal{C}(\mathcal{S}^7)}$ is $P_1 \rightarrow P_2$.

Following the ideas of the unified theory of concurrency control and recovery [Schek et al. 1993; Alonso et al. 1994; Vingralek et al. 1998], the notion of CT for

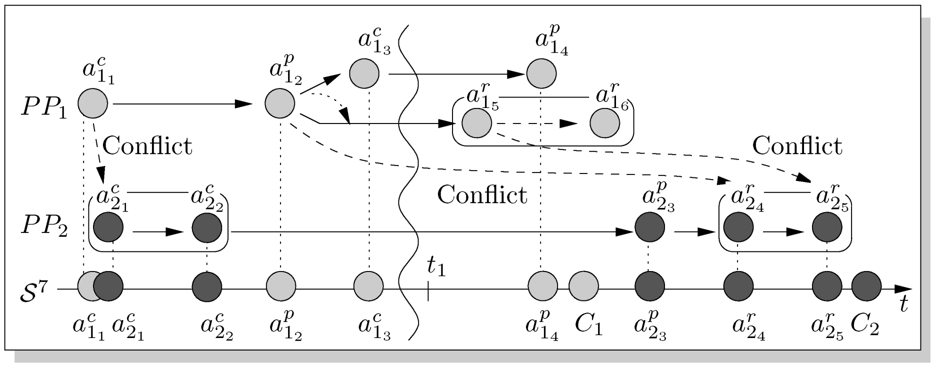


Fig. 13. CT execution of PP_1 and PP_2 in the completed process schedule $\mathcal{S}^7$.

process schedules addresses atomicity and isolation jointly. However, it extends this theory for two reasons.

First, we do not require all running processes to abort when a process schedule is completed. Moreover, CT does not necessarily require completing processes to terminate via the retriable activities of their assured termination trees but rather allows to continue the execution of subprocesses according to the processes' preference order.

Second, the traditional unified theory does not consider recovery-related operations in a schedule until expansion. During this expansion phase, each abort operation in a schedule is replaced by all appropriate undo operations which are, by a set of rules, related both to all conflicting regular operations and to undo operations of concurrent transactions, thereby assuming that these rules are actually respected by the system. In the case of transactional process management where the scheduling of compensating activities is explicitly performed by the process manager, expansion is made obsolete. However, despite the differences in the model, the correctness criteria induced by CT as well as the reduction rules for permutation and elimination of activities still follow the original unified theory of concurrency control and recovery.

In the original unified theory, the criterion SOT (serializable with ordered termination) has been introduced in order to reason about correct concurrency control and recovery of a schedule S without considering its expanded schedule $\tilde{S}$ [Alonso et al. 1994]. However, a similar, SOT-like criterion does not exist in the case of transactional process management. The reason being is that in the traditional transaction model, all operations required for recovery purposes are known beforehand. When, in addition, commutativity is perfect, then also the commutativity behavior of all recovery-related operations is known. In transactional process management and especially in the presence of completing processes, that is, when pivot activities have to be considered, such a criterion does not exist.

According to Definition 4.15, CT for a process schedule $\mathcal{S}$ can be verified only over the completed process schedule $\mathcal{C}(\mathcal{S})$. However, completing a process schedule is not practical. Therefore, special considerations or even restrictions are required to guarantee that no violation of CT occurs during completion:

—A first approach is to require that all activities of all completing processes of $\mathcal{S}$ have to be known beforehand to analyze whether conflicting activities exist (transitively) or not and whether the joint completion of these processes may violate CT (even when conflicting activities exist, they may correspond to paths that are not effected).

—Alternatively, the analysis can be restricted to the assured termination trees of all completing processes only. Then, it has to be ensured that at least one concurrent execution of all assured termination trees exists in which all completing processes successfully commit, that is, where P-SR is not violated. However, this would again be based on future activities and would, at the same time, restrict each completing process to a path within its assured termination tree.

—The previous variant could even be further restricted in that no pair of conflicting activities must exist in the assured termination trees of all completing processes that would trivially fulfill the requirement of CT (together with P-RED and P-RC of the given prefix $\mathcal{S}$ of $\mathcal{C}(\mathcal{S})$) since no new conflicts will be introduced during completion. Again, the drawback of this approach is that it forces each completing process to execute the assured termination trees while neglecting all other subprocesses with higher priority.

—Yet another approach could also be to allow only one completing process at a time. Although limiting concurrency, this variant does not restrict completion to the assured termination trees only and does also not require information about the future behavior of process programs.

These possibilities for guaranteeing CT stem from the fact that the way a given process schedule has to be completed is left intentionally vague (cf. Definition 4.14). Hence, the problem is shifted to the design and implementation of concrete protocols.

4.7 Relationship Between Classes of Process Schedules

In the previous sections, we have reformulated the traditional notions of serializability and recoverability in the context of transactional processes and, in particular, we have introduced criteria that jointly consider isolation and atomicity in transactional process management. In this section, we recall the different criteria that have been introduced and we show how they are related.

To this end, we first consider the different levels of serializability for transactional processes:

Corollary 4.16 (P-SR $\supset$ SG-P-SR $\supset$ P-SG-P-SR). *The classes P-SR, SG-P-SR, and P-SG-P-SR are related in the following way*:

(1) *SG-P-SR $\supset$ P-SG-P-SR. P-SG-P-SR is a proper subclass of SG-P-SR.*

(2) *P-SR $\supset$ SG-P-SR. SG-P-SR is a proper subclass of P-SR.*

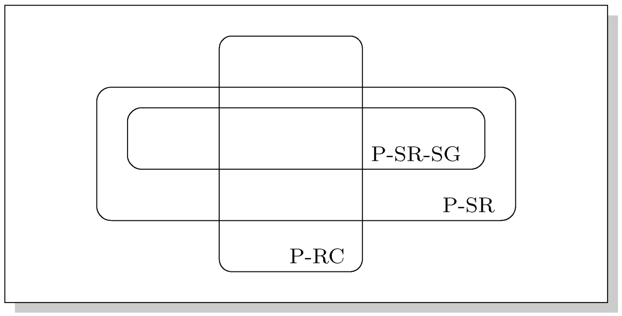


Fig. 14. Relation between P-SR, SG-P-SR, and P-RC.

PROOF OF COROLLARY 4.16

(1) *SG-P-SR* $\supset$ *P-SG-P-SR*. When P-SG-P-SR holds for some process schedule $\mathcal{S}$, not only each prefix of $\mathcal{S}$ is SG-P-SR but also $\mathcal{S}$ itself. Process schedule $\mathcal{S}^2_{t_3}$, which is SG-P-SR (after P^3_3 changes its state to aborting, it does not appear in $SG(\mathcal{S}^2)$ at time t_3), with prefix $\mathcal{S}^2_{t_2}$ that is not SG-P-SR finally shows that P-SG-P-SR is a proper subclass of SG-P-SR.
(2) *P-SR* $\supset$ *SG-P-SR*. A process schedule $\mathcal{S}$ is SG-P-SR when $SG(\mathcal{S})$ is acyclic. Since P-SR holds when PSG($\mathcal{S}$) is acyclic and since PSG($\mathcal{S}$) is obtained from $SG(\mathcal{S})$ by deleting edges, each SG-P-SR process schedule is also P-SR. Process schedule $\mathcal{S}^2_{t_2}$ of Example 4.2 is P-SR but not SG-P-SR. Therefore, SG-P-SR is a proper subclass of P-SR. □

The following discussion analyzes the relation between P-RC and P-SR as well as the relation between P-RC and the two variants of serialization graph-based process-serializability. This is also illustrated in Figure 14.

THEOREM 4.17 (P-RC VS. P-SR; P-RC VS. SG-P-SR; P-RC VS. P-SG-P-SR). *P-RC, the class of process-recoverable process schedules and P-SR, the class of process-serializable process schedules are not comparable. Analogously, P-RC and SG-P-SR as well as P-RC and P-SG-P-SR are not comparable.*

PROOF OF THEOREM 4.17. The relation between P-SR and P-RC is shown using the following examples:

(1) *¬P-SR & P-RC.* Consider, for instance, process schedule $\mathcal{S}^2_{t_1}$ of Example 4.2. It has been shown that $\mathcal{S}^2_{t_1}$ is not P-SR. However, as no order imposed by abort dependencies is violated, $\mathcal{S}^2_{t_1}$ is P-RC.
(2) *P-SR & ¬P-RC.* An example for a process schedule fulfilling P-SR but violating P-RC can be found in $\mathcal{S}^4_{t_1}$ of Example 4.4.
(3) *P-SR & P-RC.* The classes P-SR and P-RC are not disjoint. This is shown by process schedule $\mathcal{S}^8_{t_1}$ of Example 4.8 which accounts for both criteria.

Cases (1)–(3) also show the relation between P-RC and SG-P-SR. The same is true for the relation between P-RC and P-SG-P-SR. □

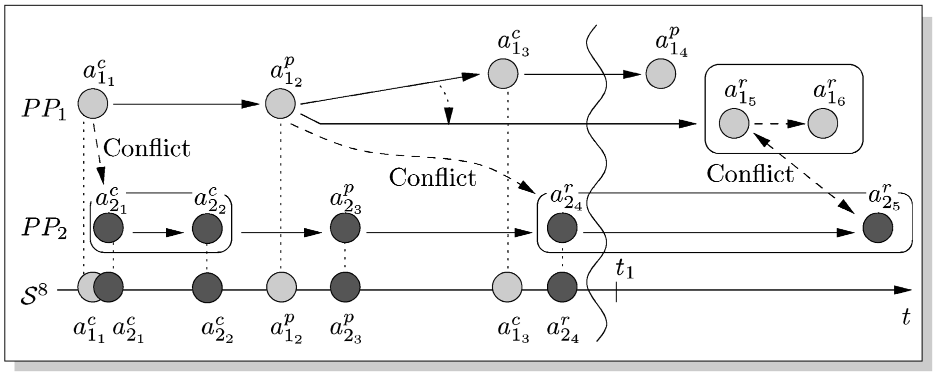


Fig. 15. P-SR & P-RC execution of process programs PP_1 and PP_2.

Example 4.8 (*P-SR and P-RC Execution*). Consider again the concurrent execution of process programs PP_1 and PP_2, reflected in process schedule $\mathcal{S}^8$ depicted in Figure 15. At time t_1, $CA(\mathcal{S}^8_{t_1})$ is conflict equivalent to the serial execution of P_1 before P_2. As additionally also P-RC is met, $\mathcal{S}^8_{t_1}$ accounts for both criteria, namely P-SR and P-RC, simultaneously. Note that the pair of conflicting activities $(a^p_{1_2} <_{\mathcal{S}^8} a^r_{2_4})$ does not impose an abort dependency between P_1 and P_2 since $a^p_{1_2}$ is not compensatable.

Process-reducibility has been introduced as a criterion to account for both atomicity and isolation in transactional processes. In what follows, we compare P-RED and its prefix-closed variant, P-P-RED, to process-recoverability and to the different levels of serializability that have been identified for transactional processes.

THEOREM 4.18 (P-SR $\supset$ P-RED). *P-RED is a proper subclass of P-SR.*

PROOF OF THEOREM 4.18. Let $\mathcal{S}$ be a P-RED process schedule and assume that $\mathcal{S}$ is not P-SR. Then, a cycle $P_i \rightarrow P_{i+1} \rightarrow \cdots \rightarrow P_{i+m} \rightarrow P_i$ has to exist in the committed and active projection $CA(\mathcal{S})$ of $\mathcal{S}$. Since no aborting and aborted (sub-)processes appear in $CA(\mathcal{S})$, none of the activities involved in the conflict cycle is compensated in $\mathcal{S}$. Therefore, this cycle cannot be eliminated by any reduction rule which contradicts with the initial assumption of $\mathcal{S}$ being P-RED. Furthermore, P-RED is a proper subclass of P-SR. The latter considers only committed and active (sub-)processes. A conflict cycle imposed by compensating activities only will not affect P-SR but leads to a violation of P-RED which is shown in Example 4.9. □

Example 4.9 (*Non-P-RED Execution*). Consider process schedule $\mathcal{S}^9$ reflecting the concurrent execution of process programs PP_3 and PP_4 illustrated in Figure 16. At time t_1, no active or committed process exists. Therefore, P-SR trivially holds. However, $\mathcal{S}^9_{t_1}$ is not P-RED. The pairs of conflicting activities $a^c_{3_1} <_{\mathcal{S}^9_{t_1}} a^c_{4_1}$ and $a^{-1}_{3_1} <_{\mathcal{S}^9_{t_1}} a^{-1}_{4_1}$ lead to a case that cannot be resolved by reduction techniques.

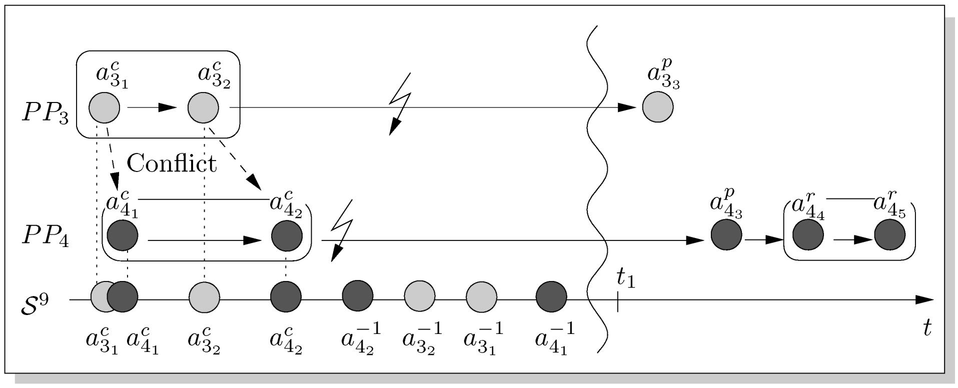


Fig. 16. Non-P-RED but P-SR process schedule $\mathcal{S}^9$.

LEMMA 4.19 (P-RED VS. SG-P-SR; P-RED VS. P-SG-P-SR). *P-RED and SG-P-SR are not comparable. Analogously, P-RED and P-SG-P-SR are not comparable.*

PROOF OF LEMMA 4.19. The following examples show the relation between P-RED and SG-P-SR:

(1) *¬P-RED & SG-P-SR*. Consider again process schedule $\mathcal{S}^9_{t_1}$ of Example 4.9. It has already been shown that $\mathcal{S}^9_{t_1}$ is P-SR. Since no active processes exist in $\mathcal{S}^9_{t_1}$, it is also SG-P-SR. However, due to the cyclic conflicts imposed by the compensating activities in the case of perfect commutativity, P-RED does not hold.
(2) *P-RED & ¬SG-P-SR*. Process schedule $\mathcal{S}^2_{t_2}$ of Example 4.2 is P-RED but it is not SG-P-SR.
(3) *P-RED & SG-P-SR*. Process schedule $\mathcal{S}^3_{t_1}$ of Example 4.3 shows that P-RED and SG-P-SR are not disjoint. Since no (sub-)process of $\mathcal{S}^3_{t_1}$ is aborted, thus it contains no compensating activity, and since $SG(\mathcal{S}^3_{t_1})$ is acyclic, both criteria hold simultaneously.

Cases (1)–(3) also apply for the relation between P-RED and P-SG-P-SR. □

We have previously shown that P-RED and P-SR are not comparable. The same is true when comparing the classes P-RED and P-RC:

THEOREM 4.20 (P-RC VS. P-RED). *P-RC and P-RED are not comparable.*

PROOF OF THEOREM 4.20. The relation between P-RC and P-RED is shown using the following examples:

¬P-RC & P-RED. It is possible that violations of constraints imposed by abort dependencies exist in a process schedule $\mathcal{S}$ albeit $\mathcal{S}$ is P-RED. This is the case when, for instance, the (sub-)processes involved in the abort dependency is not aborted (note that P-RC does not require all active processes to be aborted), that is, the compensating activity that would violate P-RED is not present in $\mathcal{S}$. Process schedule $\mathcal{S}^4_{t_1}$ of Example 4.4 represents such a case: $\mathcal{S}^4_{t_1}$ is not P-RC but

it is P-RED; yet the compensation of a_{1_1}—which is not executed at t_1—would also violate P-RED.

P-RC & ¬P-RED. Consider process schedule $\mathcal{S}_{t_1}^2$ of Example 4.2. Since no compensating activity exists, the compensation rule cannot be applied and the commutativity rule does not allow to transfer $\mathcal{S}_{t_1}^2$ to a serial schedule. Thus, it is not P-RED. However, it can be shown that $\mathcal{S}_{t_1}^2$ is P-RC since no constraints imposed by abort dependencies are violated.

P-RC & P-RED. Process schedule $\mathcal{S}_{t_1}^8$ of Example 4.8 holds for both P-RC and P-RED. The commutativity rule allows to rearrange all activities in order to transform $\mathcal{S}_{t_1}^8$ to a serial execution of P_1 succeeded by P_2. The compliance of $\mathcal{S}_{t_1}^8$ with P-RC has already been shown. □

Process-reducibility not only guarantees process-serializability but it also addresses the correct execution of compensating activities, both from aborting processes and aborting subprocesses. However, P-RED does not provide both P-SR and P-RC jointly:

LEMMA 4.21 (P-RED $\not\subseteq$ P-SR $\cap$ P-RC). *P-RED does not hold for P-RC and P-SR simultaneously.*

PROOF OF LEMMA 4.21. Consider again process schedule $\mathcal{S}_{t_1}^4$ of Example 4.4. We have already shown that $\mathcal{S}_{t_1}^4$ is P-RED. However, it has also been shown that it is not P-RC. Therefore, it does also not meet P-RC $\cap$ P-SR, that is, both process-recoverability and process-serializability simultaneously. □

P-RED does not jointly hold for both P-RC and P-SR for two reasons. First, we do not require all active processes of a process schedule $\mathcal{S}$ to abort but only a subset $R_{\mathcal{S}}^*$ of all running processes. Therefore, violations of constraints imposed by abort dependencies do not affect P-RED when the corresponding processes will not be aborted. Second, P-RED does not require process schedules to be complete. Even if violations of P-RC exist and the associated (sub-)processes do not commit, they might be aborting and the compensating activity finally leading to a violation of P-RED might not yet be present in the process schedule. However, P-RED ensures that no aborted process P_i is involved in an abort dependency with another process P_j that is not aborted. More formally,

Definition 4.22 (*Abort-Process-Recoverability* (*P-RC-A*)). A process schedule $\mathcal{S}$ is P-RC-A if the constraints imposed by P-RC—restricted to all abort dependencies between activities $a_{i_k}^c$ and a_{j_l} where the (sub-)process of $a_{i_k}^c$ is aborted—are met in $\mathcal{S}$.

Obviously, since the absence of abort dependencies imposed by pairs of activities (a_{i_k}, a_{j_m}) where P_i is aborted, but not P_j, is essential in a P-RED process schedule, it meets both P-SR and P-RC-A simultaneously, that is P-RED $\subset$ P-SR $\cap$ P-RC-A.

COROLLARY 4.23 (P-P-RED). *The following relationships can be identified between the classes P-P-RED and P-RED, P-P-RED and SG-P-SR, P-P-RED* *and P-SG-P-SR, as well as between P-P-RED and P-RC*:

(1) *P-P-RED* $\subset$ *P-RED. P-P-RED is a proper subclass of P-RED.*

(2) *P-P-RED vs. SG-P-SR. P-P-RED and SG-P-SR are not comparable.*

(3) *P-P-RED vs. P-SG-P-SR. P-P-RED and P-SG-P-SR are not comparable.*

(4) *P-P-RED vs. P-RC. P-P-RED and P-RC are not comparable.*

PROOF OF COROLLARY 4.23

(1) *P-P-RED* $\subset$ *P-RED.* When each prefix of a process schedule $\mathcal{S}$ is P-RED, so is also $\mathcal{S}$. Moreover, P-P-RED is a proper subset of P-RED: process schedule $\mathcal{S}^2_{t_2}$ of Example 4.2, for instance, is P-RED although its prefix $\mathcal{S}^2_{t_1}$ is not P-RED.

(2) *P-P-RED vs. SG-P-SR.* Process schedule $\mathcal{S}^7_{t_1}$ is not P-P-RED but it is SG-P-SR. Conversely, a process schedule $\mathcal{S}$ might be P-P-RED but not SG-P-SR when a conflict cycle is induced by two completing processes $a^c_{i_l} <_{\mathcal{S}} a^c_{j_m} <_{\mathcal{S}} a^{-1}_{j_m} <_{\mathcal{S}} a^{-1}_{i_l}$ with $a^p_{i_g} <_{\mathcal{S}} a^c_{i_l}$ and $a^p_{j_h} <_{\mathcal{S}} a^c_{j_m}$. In this case, since both P_i and P_j are completing, the conflicts of their aborted subprocesses are included in $SG(\mathcal{S})$ but by applying the commutativity and compensation rules, all activities of the conflict cycle can be cancelled. Finally, P-P-RED and SG-P-SR are not disjoint since, for instance, process schedule $\mathcal{S}^8_{t_1}$ meets both criteria.

(3) *P-P-RED vs. P-SG-P-SR.* A process schedule $\mathcal{S}$ is P-SG-P-SR but not P-P-RED if it is, for instance, defined over two processes, say P_i and P_j, and if it contains a conflict cycle $a^c_{i_l} <_{\mathcal{S}} a^c_{j_m} <_{\mathcal{S}} a^{-1}_{i_l} <_{\mathcal{S}} a^{-1}_{j_m}$ where both processes are aborted or aborting and where no other pairs of conflicting activities exist. Although the cycle cannot be eliminated by the reduction rules, SG is, for $\mathcal{S}$ as well as for each prefix $\mathcal{S}'$ of $\mathcal{S}$, acyclic (since at least one process has been aborting when the conflict cycle was introduced). A process schedule $\mathcal{S}$ over two completing processes, P_i and P_j, is P-P-RED but not P-SG-P-SR when, for instance, a conflict cycle $a^c_{i_l} <_{\mathcal{S}} a^c_{j_m} <_{\mathcal{S}} a^{-1}_{j_m} <_{\mathcal{S}} a^{-1}_{i_l}$ with $a^p_{i_g} <_{\mathcal{S}} a^c_{i_l}$ and $a^p_{j_h} <_{\mathcal{S}} a^c_{j_m}$ exists. Although all activities of the conflict cycle can be eliminated by applying reduction rules, a cycle is present in $SG(\mathcal{S})$. Process schedule $\mathcal{S}^8_{t_1}$ of Example 4.8 accounts for both criteria such that P-P-SG-SR and P-P-RED are not comparable.

(4) *P-P-RED vs. P-RC.* Process schedule $\mathcal{S}^4_{t_1}$ of Example 4.4 is P-P-RED but not P-RC. When a conflict cycle $a^p_{i_k} <_{\mathcal{S}} a^p_{j_m} <_{\mathcal{S}} a^p_{i_l}$ exists in a process schedule $\mathcal{S}$ formed by pivot activities only and when, in addition, $\mathcal{S}$ is free of abort dependencies, it is P-RC but not P-P-RED. Therefore, the classes P-P-RED and P-RC are not comparable. $\square$

In Lemma 4.21, we have proven that process-reducibility does not meet P-SR and P-RC jointly. In what follows, we show that the same is true for P-P-RED, the prefix-closed subclass of P-RED:

LEMMA 4.24 (P-P-RED $\not\subseteq$ P-RC $\cap$ P-SR). *P-P-RED does not hold for P-RC and P-SR simultaneously.*

PROOF OF LEMMA 4.24. In order to show the relation between P-P-RED and P-SR $\cap$ P-RC, again process schedule $\mathcal{S}^4_{t_1}$ can be analyzed. Since each prefix of $\mathcal{S}^4_{t_1}$ can be correctly reduced, it meets P-P-RED. However, since $\mathcal{S}^4_{t_1}$ is not P-RC,

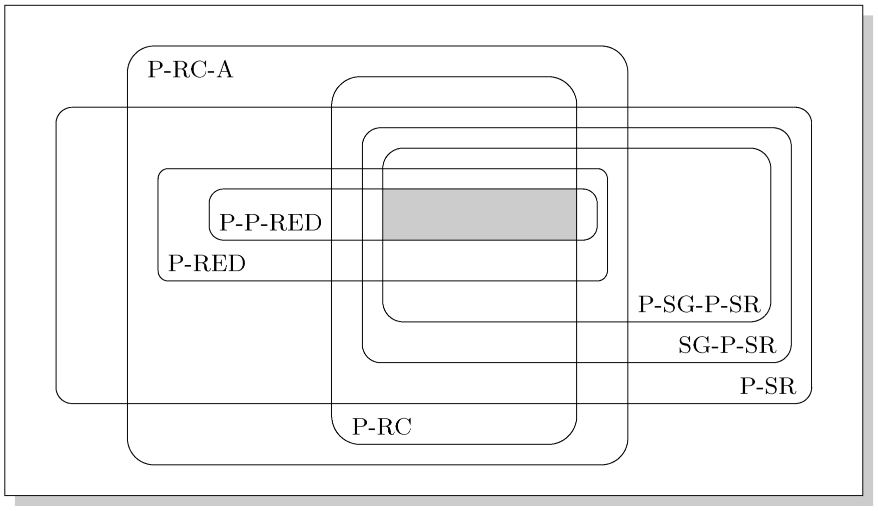


Fig. 17. Relation between P-SR, SG-P-SR, P-SG-P-SR, P-RC, P-RC-A, P-RED, and P-P-RED.

P-P-RED does not provide process-serializability and process-recoverability jointly. □

The above discussion is summarized in Figure 17 where the relationships between the classes P-SR, SG-P-SR, P-SG-P-SR, P-RC, P-RC-A, P-RED, and P-P-RED (thus, also CT) are illustrated.

Since CT corresponds to P-P-RED for completed process schedules, it holds for both P-SR and P-RC-A. But since the process-recoverability requirement explicitly includes the possibility to chose, for each partial process schedule $\mathcal{S}$, an arbitrary set $R_{\mathcal{S}}$ of running processes and to abort all processes of its superset $R^*_{\mathcal{S}}$ correctly, P-RC-A is not sufficient. The reason being is that, in contrast to the traditional unified theory, we do not require all active processes to abort and that, when scheduling is performed dynamically, the subset of active processes that will finally be aborted in the completed process schedule is not known in advance. Therefore, a dynamic scheduling protocol for transactional processes has to provide P-RC and P-P-RED simultaneously. In addition, such a dynamic protocol has to guarantee that CT, the correct termination, is possible for each partial process schedule by implementing one of the four strategies for completion we have discussed in Section 4.6.

5. PROCESS LOCKING: DYNAMIC PROTOCOL FOR PROCESS SCHEDULES

Based on the WISE system [Alonso et al. 1999c; Lazcano et al. 2000], a transactional process manager has been implemented that supports P-P-RED & P-RC & P-SG-P-SR process executions and which guarantees the correct termination of each partial process schedule. Before introducing *process locking* [Schuldt 2001a], the protocol that has been implemented, the basic assumptions are clarified.

First of all, each process program to be executed has to be inherently correct: it has to follow the guaranteed termination property (Axiom 4.1). Additional assumptions address commutativity and compensation: the commutativity relation must be perfect (see Definition 4.4) and compensation must be state-independent.

Assumption 5.1 (*Commutativity*). Commutativity is perfect.

Assumption 5.2 (*State-Independent Compensation*). Compensation is state-independent, that is a compensating activity can be executed (exactly once) but at any point in time (thus in any state) after the commit of the according forward activity.

Summarizing the discussion of Section 4, a dynamic scheduling protocol has to enforce P-P-RED, P-RC, and, for completed process schedules, CT (i.e., it must avoid unresolvable situations in which two or more completing processes are involved).

The process locking protocol within the WISE system is based on and extends ideas of locks with constrained sharing [Agrawal and Abbadi 1990] and timestamp ordering [Thomas 1979]. In what follows, we will motivate the necessity of advanced mechanisms for supporting CT and present the protocol in detail. In particular, process locking exploits the restriction to only one completing process at a time to correctly terminate a partial process schedule (as discussed in Section 4.6) in order to avoid the consideration of future activities of process programs. Hence, due to these restrictions, process locking does not only provide P-P-RED and P-RC but also P-SG-P-SR. The superclass P-P-RED & P-RC & P-SG-P-SR of all process schedules produced by process locking is highlighted in gray color in Figure 17.

5.1 Locks with Constrained Sharing

When considering the type of locks to be used, the nature of processes needs to be taken into account. Activities in a transactional process are high level semantic abstractions. They typically correspond to complex application invocations. To use conventional notions like exclusive or shared access would be too restrictive. A more appropriate concept is to exploit locks with constrained sharing [Agrawal and Abbadi 1990]. The idea is to use, in addition to shared and exclusive locks, a third category, termed *ordered shared locks* (*OSL*). OSL can be shared between different transactions under certain constraints: with each sharing, an order is associated which has to be respected for the execution of the respective 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 and 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 in that locks on activities can be ordered shared. Yet, the prerequisite for the application of locking techniques at activity

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

held \ acquired	C	P
C	$\Rightarrow$	$\not\Leftrightarrow$
P	$\Rightarrow$	$\not\Leftrightarrow$

level is that a complete commutativity relation is available to the process manager.

In Section 4, we have seen that the allowed (as well as the disallowed) interleavings of processes are governed by the conflict behavior of activities and their termination properties, that is, whether or not they can be compensated. Hence, applied to process schedules and to the requirements imposed by transactional process management, ordered shared locks at activity level provide a straightforward means to map allowed interleavings of processes into a compatibility matrix of different lock types. For this purpose, and 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 [Schuldt 2001a]; since retriable activities are also either compensatable or pivot, the same types of locks are applied to them. While two C locks of different processes as well as a P lock followed by a C lock may be ordered shared, this is not the case for a C lock followed by a P lock. In the latter case, when the process having requested the C lock is running, this would correspond to an abort dependency which has to be prevented in order to guarantee P-RC. Therefore, C and P locks cannot be ordered shared but must be exclusive. Finally, the combination of two P locks has also to be dealt with care since this combination of locks implies that both associated processes are completing and may impose deadlocks that cannot be resolved. The compatibility matrix is depicted in Table I where $\Rightarrow$ denotes ordered shared mode and $\not\Leftrightarrow$ stands for nonshared (exclusive) mode. Note that this compatibility matrix corresponds to the algorithm deciding whether edges in the serialization graph are allowed or disallowed, as discussed in Section 4.4.

5.2 Timestamp Ordering

The original protocol proposed by Agrawal and El Abbadi [1990] generalizes standard two phase locking. This protocol has an optimistic character since the compliance of orders is not checked until the first lock is to be released (due to the lock relinquish rule). And since preclaiming does not appear to be the ultimate solution in the case of transactional processes (all possible paths would have to be considered although eventually only few of them are effected), the validation would more or less coincide with the commit of a process. This, in turn, means that violations of the order constraint of locks are detected at a very late stage and even worse, may occur in situations where appropriate corrective strategies, that is, 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. To do this, we adopt and apply ideas borrowed from timestamp ordering (TO) protocols [Thomas 1979]. 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.

5.3 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, that is, for each transaction program that can be invoked by the process manager, additional information in the form of an ordered list is maintained that 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 a process timestamp. Finally, for each activity a_i, the set of activities a_j with $CON(a_i, a_j) = TRUE$, taken from a matrix reflecting the complete commutativity relation, has to be available.

Even when combining the original OSL protocol based on P and C locks with timestamp ordered lock requests, special treatment is necessary for pivot activities. The previous sections made the dual character of pivot activities obvious: on the one hand, they are "normal" activities; on the other hand, they have a commit-like semantics since they make compensation unavailable for all preceding activities. Due to this dualism, a pivot activity can neither be treated like the commit of a process nor like a normal activity. Although, for instance violations of constraints imposed by abort dependencies between processes P_i and P_j are no longer possible once a subsequent pivot of P_i is executed, locks on these compensatable activities must not be released (as would be the case for a commit). Otherwise, P-SR could no longer be guaranteed. When pivot activities are considered as regular activities, only abort dependencies could be detected in which the pivot itself is involved, but no others. Reconsider the P-RC algorithm based on the serialization graph, sketched in Section 4.4, where also certain state changes of a process (e.g., by executing a pivot) provoked the verification of all existing dependencies of this process. This verification is captured by the conversion of all C locks already held for compensatable activities to P locks once a pivot activity is to be executed.

Aborting processes executing compensating activities require special treatment because it has to be guaranteed that aborting processes themselves are not aborted. Additionally, once a process is completing, it will be favored in that it may override timestamp orders for lock requests.

Process locking 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, a lock must be acquired which has to meet a_{i_k}'s termination property (either a C lock or a P lock). This lock then corresponds to an entry in the lock list of the activity. 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:

(1) COMP-RULE: EXECUTION OF A COMPENSATABLE ACTIVITY a_i^c. For the execution of a compensatable activity, 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 $a_{i_k}^c$ 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, $a_{i_k}^c$ 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 $a_{i_k}^c$ can be acquired, and $a_{i_k}^c$ 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 $a_{i_k}^c$ 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 $a_{i_k}^c$ 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.

(2) PIV-RULE: EXECUTION OF A PIVOT ACTIVITY a_i^p. When a_{i_k} is a pivot activity, then P_i has to acquire a P lock before it can be executed. However, prior to the P lock request for $a_{i_k}^p$, all C locks of P_i held for activities a_{i_h} preceding $a_{i_k}^p$ have first to be converted to P locks. The reason being is the dual character of pivots. There can be C locks ordered shared with older processes that are still running or are in a running subprocess. The execution of the pivot $a_{i_k}^p$, which additionally corresponds to a state change from running to completing in the case it is a

primary pivot, could violate constraints imposed by potentially existing abort dependencies which, in turn, would correspond to a violation of P-RC. Again, a distinction is required on whether the P lock can be granted immediately after lock conversion, whether it requires the abort of some concurrent processes, or whether it has to be deferred.

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 end of these processes. 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).

(3) COMP $\longrightarrow$ PIV-RULE: CONVERSION OF C LOCKS TO P LOCKS. 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 $\rightarrow$ 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 $\rightarrow$ P lock conversion has to be deferred until the end of these processes.

(4) $\mathbf{C^{-1}}$-RULE: EXECUTION OF A COMPENSATING ACTIVITY a_i^{-1}. When a process P_i is aborting, it must be able to correctly undo all its activities. Eventually, there are processes P_j with younger timestamps than P_i, $ts(P_i) < ts(P_j)$, that have executed an activity a_{j_l} that conflicts with $a^c_{i_k}$ and that appears after $a^c_{i_k}$ with respect to the observed order $<_{\mathcal{S}}$ in a process schedule $\mathcal{S}$. According to the Comp-Rule, this case is allowed, at least when all these processes P_j are running. To correctly undo $a^c_{i_k}$, again a C lock has to be acquired by $a^{-1}_{i_k}$. This leads to the abort of all such processes P_j which have common locks with P_i but younger timestamps. Yet, the abort of a process P_i may induce cascading aborts, conforming with the notion of P-RC. In the case of cascading aborts, all aborted processes are resubmitted in timestamp order, thereby keeping their original timestamp. All older processes P_k having locks in common with P_i are not affected by the C lock request induced by $a^{-1}_{i_k}$ and thus, by the abort of P_i.

(5) ABORT-RULE: ABORT A_i OF A PROCESS. The abort A_i of a process P_i leads to the release of all locks held by P_i.

(6) COMMIT-RULE: COMMIT C_i OF A PROCESS. In accordance with the lock relinquish rule of the original OSL protocol, a process P_i is finally allowed to commit if all its locks are shared in the correct order. Applied to transactional

processes and to the criterion of P-RC, a process must not commit if it has common locks (which correspond to abort dependencies) shared with older processes P_j, $ts(P_j) < ts(P_i)$. In this case, C_i must be deferred until all these P_j have committed. Note that all common locks shared with older processes may only be C locks. Otherwise, if no common locks with older processes exist, P_i is allowed to commit and to release all its locks; therefore, process locking follows the strict two phase locking (S2PL) paradigm [Eswaran et al. 1976].

Note that, although compensating activities are themselves required to be pivot, that is, compensation cannot be compensated, we do not demand them to acquire P locks since this would, due to the Piv-Rule, require C $\rightarrow$ P lock conversion for all C locks of aborting processes. Essentially, this would not be possible if older processes exist holding ordered shared C locks. Yet, it is sufficient for guaranteeing CT to abort only processes which have been executed conflicting activities between a regular and a compensating activity—which is already captured by requiring C locks for compensation (as part of the special treatment being part of the acquisition rules for C and P locks). In the case of retriable activities a^r, once they have acquired the adequate lock (either C or P), the corresponding transaction can be invoked in the subsystem; yet, even in the case of a failure of this transaction, the lock granted to a^r guarantees that it can be safely reinvoked.

Obviously, by allowing to share locks in timestamp order, a process may induce cascading aborts. However, due to the exclusive treatment of certain combinations of locks, it is ensured that cascading aborts are restricted to running processes. 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 [Rosenkrantz et al. 1978; Bernstein et al. 1987] 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.

Process locking supports the correctness criteria for transactional processes we have identified in Section 4. In particular, it provides P-SG-P-SR and P-RC process schedules and it guarantees that each process schedule can be terminated correctly (CT), thus also accounts for P-P-RED. More formally,

THEOREM 5.3 (PROCESS LOCKING). *Each process schedule $\mathcal{S}$ generated by process locking is P-SG-P-SR, P-RC, and P-P-RED. In addition, each completed process schedule $\mathcal{S}^*$ generated by process locking is CT.*

PROOF. The proof of Theorem 5.3 can be found in Schuldt [2001b]. $\square$

Due to the usage of process timestamps and the assignment of timestamps to processes when the latter are instantiated, process locking may rule out certain process schedules that are considered as correct, hence supports only a subset of P-SG-P-SR & P-RC & P-P-RED process schedules. This is shown in the following example:

Example 5.1 (*Restrictions of Process Locking*). Consider process schedule $\mathcal{S}^{10}$, depicted in Figure 18. Obviously, $\mathcal{S}^{10}$ accounts for P-SG-P-SR, P-RC, and

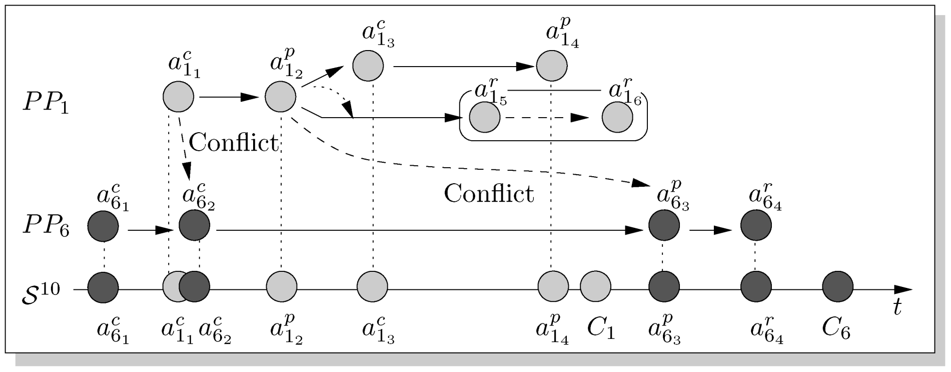


Fig. 18. CT process schedule $\mathcal{S}^{10}$ violating process locking rules.

P-P-RED as well as for CT. However, since the process timestamps do not coincide with the serialization order, the sharing of locks between P_1 and P_6 for $(a^c_{1_1}, a^c_{6_2})$ violates the Comp-Rule. Hence, $\mathcal{S}^{10}$ cannot be produced by process locking.

6. IMPLEMENTATION

Support for transactional processes has been implemented on top of the WISE system [Alonso et al. 1999c; Lazcano et al 2000], which acts as transactional process manager. The WISE system, in turn, builds upon the process support system OPERA [Hagen 1999]. In addition to the process manager functionality, WISE provides support for various practical problems:

—the *modeling and development* of process programs by integrating existing services as process activities and the specification of control flow dependencies (precedence and preference orders),

—the *monitoring* of the state of active processes,

—the facilities to make the state of each active process *persistent* so as to recover after system failures, and

—the possibility to integrate arbitrary transactional subsystems by appropriate *subsystem adapters*.

Figure 19 shows the overall architecture of the WISE framework. In what follows, we discuss only the two most important components, namely the process program specification tool and the WISE transactional process manager.

6.1 Process Program Specification

We use a commercial tool, IVYFRAME [IvyTeam 2001] for specifying process programs. This is done by graphically bringing activities as core building blocks together with control flow aspects, that is, precedence and preference dependencies, and data flow constraints between activities. The specification tool supports the full capabilities of the process programs model like

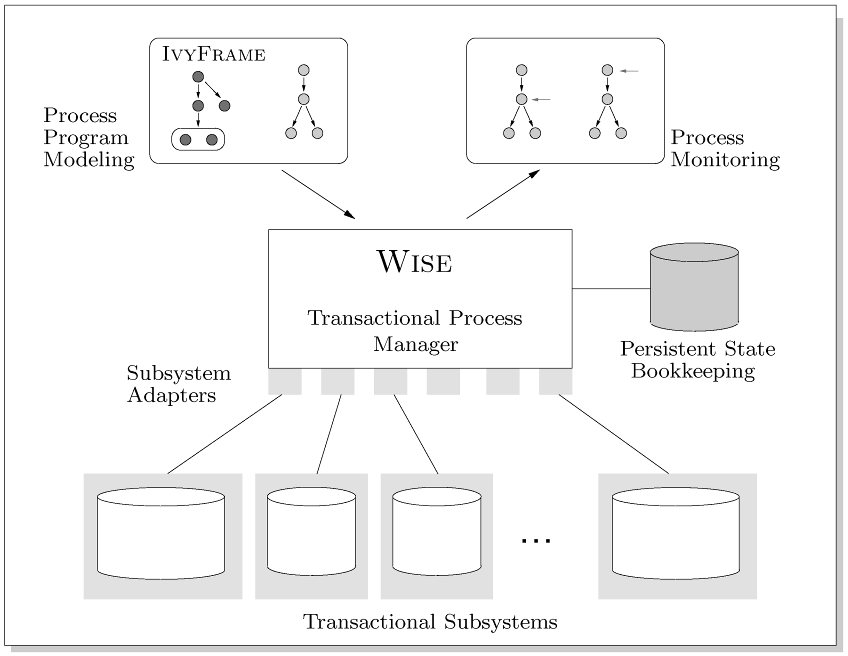


Fig. 19. General architecture of the Wise framework.

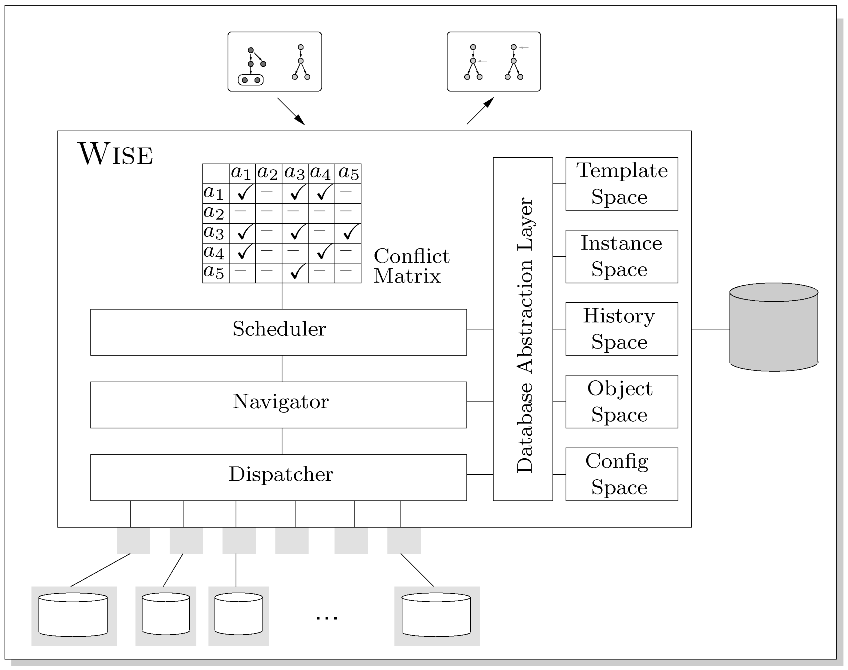
conditional branching and partially ordered activities of multiactivity nodes. In the latter case, this is modeled by encompassing the activities of such multiactivity nodes into a nested subprocess program that can be recursively embedded within a top-level process program. In addition, the process program specification in the extended IvyFrame system considers special runtime information about activities. This contains the location of the subsystem providing a service (IP address of the respective hosts), the parameters of the service invocation, and the service characteristics, that is, whether or not compensation exists, whether a service is retriable, and so forth. Essentially, this information is vital to the actual execution of the process programs.

An important extension that has been added to the IvyFrame system is the possibility to check whether single process programs are correctly defined, that is, whether they meet the guaranteed termination criterion. This verification is based on the control flow dependencies as well as on the individual properties of activities.

The graphical representation produced by IvyFrame is finally compiled into a language called OCR (Opera Canonical Representation) [Hagen 1999] that is understood by the Wise process manager and can be directly used for execution purposes.

6.2 Process Program Execution

The Wise engine is the core component of the complete framework orchestrating the concurrent execution of process programs, thereby acting as transactional



	a_1	a_2	a_3	a_4	a_5
a_1	✓	–	✓	✓	–
a_2	–	–	–	–	–
a_3	✓	–	✓	–	✓
a_4	✓	–	–	✓	–
a_5	–	–	✓	–	–

Fig. 20. General architecture of the WISE engine.

process manager. The main components of the WISE engine are depicted in Figure 20. In what follows, we concentrate on the discussion of the most crucial features for transactional process support. More details on the WISE engine and the OPERA process support kernel can be found in Lazcano et al. [2000], and Hagen [1999].

The most relevant components for the purposes of this article are the *navigator*, *scheduler*, and *dispatcher* modules. The navigator interprets the process program described in OCR and determines for each active process, according to the control flow dependencies, what activities have to be executed next, that is, it "navigates" through process programs. Prior to invoking the services corresponding to process activities, the navigator contacts the scheduler which determines whether activities are allowed to be executed in the presence of concurrency, whether they have to be deferred, or even whether certain processes have to be aborted. To this end, the scheduler implements the process locking protocol (see Section 5). In particular, the scheduler makes use of a conflict matrix containing the specification of the commutativity characteristics of all activities (which has to be provided by some administrator) as well as of the termination properties of individual activities as specified in OCR. In addition, the scheduler is responsible for assigning and for managing timestamps associated with processes. Once the navigator in accordance with the scheduler decide which activity to execute, the information is passed to the dispatcher which, in turn, associates it with a processing node in the cluster and a service of a particular application. Hence, the dispatcher deals with physical distribution by transparently managing the communication with remote system components.

All information necessary to execute processes is made persistent in the underlying WISE database(s). According to the characteristics and purpose of this information, it is subdivided into in separate spaces. The *template space* contains all process programs that have been loaded into the WISE engine. For each active process, a copy of the corresponding process program is made and placed in the *instance space*. This copy is used to record the process' state as execution proceeds. Storing instances persistently guarantees forward recoverability, that is, execution can be resumed as soon as the failure is repaired, which solves the problem of dealing with failures of long lived processes [Dayal et al. 1991]. In addition, the runtime information of the instance space also contains the timestamp and the currently held locks of each process. The *history space* manages information about terminated processes while the *config space* encompasses global system information, that is, it persistently stores the conflict matrix.

7. RELATED WORK

In this section, we introduce related work in the area of transactional workflows.

7.1 ConTracts and Spheres of Isolation

The *ConTract model* [Wächter 1991, 1996; Wächter and Reuter 1992; Reuter et al. 1997] aims at bringing together elements of programming languages (control flow specifications, iterations, conditional branching, etc.) and transaction processing. A ConTract is equivalent to a long-running transaction, or process, and consists of a set of *steps* that are combined by a *script* that specifies the execution dependencies between them and that allows to encompass single steps in atomic units of work. Associated with each single step, which is required to be compensatable, is an *entry invariant* and an *exit invariant*. The entry invariant reflects the conditions that must be true in order to start the execution of a step. The exit invariant of a step contains the post conditions that hold after its successful execution. If this condition is present in the entry invariant of a subsequent step, it must not be violated in between (= establishment of an invariant). The concurrency control mechanism exploited for ConTracts is therefore called *invariant-based serializability*. In spite of the existence of compensation for each step, the default strategy for dealing with failures is forward recovery, based on the persistently stored ConTract state and the context of the current execution. A joint criterion for fault-tolerant concurrent ConTract executions exists that exploits the notion of expansion of the original unified theory of concurrency control and recovery. However, forward recovery—although it is the default mechanism for recovery—is not present in this joint criterion where only backward recovery is considered.

The core model the *Spheres of Isolation* (SOI) approach [Schwenkreis and Reuter 1996] is an extension of the ConTract model. Basic elements are activities and transitions specifying the conditions for activities to be started.

Unlike the basic ConTract model, constraints are settled at object level. The goal of a concurrent execution of processes is to provide two properties: success and correctness. The *success* of a process execution denotes the possibility for correct termination even in the presence of concurrency, leading to the notion of "object-local" concurrency control. The notion of *correctness* guarantees the availability of compensation until the end of a process. Forward recovery is considered to be the default strategy for failure handling, although the correctness of a process only addresses backward recovery based on compensation. Correctness and successful termination is treated differently by assigning a (symbolic) *Sphere of Isolation* to each property.

7.2 Spheres of Joint Compensation

The *spheres of joint compensation* approach addresses the fault-tolerant execution of single processes [Leymann 1995]. This approach considers compensation not only for single activities but also allows to assign one single compensation activity to groups of activities. A sphere of joint compensation is such a set of activities of a process that either all together have to be executed successfully or that all have to be compensated. Different spheres may intersect or be even contained within each other. For recovery purposes, either pure backward recovery or partial compensation combined with the reinvocation of failed activities is possible. Concurrency control is not addressed by this approach.

7.3 Open Process Management (OPM)

Open process management (*OPM*) [Chen and Dayal 1996, 1997] brings together a nested activity model with a combination of closed nested [Moss 1985, 1987] and open nested transactions [Weikum and Schek 1992]. Each process consists of single activities and/or blocks; the latter consist recursively again of activities and blocks. Activities within a process are open, that is, they are allowed to commit prior to the commit of a process although these changes are only visible to activities of the same process and not to the outside (these activities are called *in-process open*). That is, in-process open activities are open nested transactions within the scope of a closed transaction, the associated process. The goal of these activities is to increase parallelism compared with closed nested transactions but, at the same time, to avoid the relaxation of atomicity as given by the open nested transaction model. Recovery encompasses partial backward recovery and alternative executions within blocks, if available. When an activity fails, recovery first addresses the affected block only and it will be checked whether in-block alternative executions exist. Otherwise, failure handling has to be extended to the block hierarchy. In addition to treating concurrency control and recovery independently, OPM differs from transactional processes in that the latter allow for a higher degree of parallelism. Essentially, the restriction that effects of activities are only visible within the corresponding process (as it is the case in OPM) does not exist in transactional process management.

8. CONCLUSION

Large-scale applications typically integrate several independent and distributed components rather than relying on a single centralized database and exploiting one global data model. Moreover, the development of such applications has to take into account that the individual computation steps are already in place but have to be glued together in a coherent way by means of control and data flow dependencies. Process programs allow for this kind of higher level application development. In this paper, we provide a framework to reason about correct executions of process programs, termed processes. To this end, we generalize the traditional notion of atomicity for single processes (leading to *guaranteed termination*) by allowing flexible failure handling and alternative executions being defined within process programs and by taking into account the different termination properties of single activities. Most importantly, we treat the problem of atomicity and isolation in transactional processes simultaneously within the same framework (*correct termination*) by extending and generalizing the unified theory of concurrency control and recovery. Yet, unlike other approaches, we jointly cover both atomicity and isolation and do concurrency control and recovery at the appropriate level, the scheduling of processes.

The framework established in this paper covers various applications such as workflow management, process support systems, and the provision of appropriate infrastructures for electronic commerce [Schuldt et al. 2000], virtual enterprises [Alonso et al. 1999c], and the coordination of subsystems [Schuldt et al. 1998]. In addition to covering such a large variety of applications, this framework is also completely transparent to the user. Based on the WISE process support system developed at ETH Zürich [Hagen 1999; Alonso et al. 1999c], we have implemented a transactional process scheduler based on the process locking protocol presented in this paper. This work, together with the correctness checking of single processes with respect to their guaranteed termination property, completes the effort to develop an infrastructure for supporting processes as applications at a higher level of semantics, that is, applications on top of independent component systems, and to provide transactional execution guarantees for these processes.

REFERENCES

AGRAWAL, D. AND EL ABBADI, A. 1990. Locks with constrained sharing. In *Proceedings of the 9th Annual ACM Symposium on Principles of Database Systems* (*PODS'90*) (Nashville, Tenn.). ACM, New York, pp. 85–93.

ALONSO, G. 1997. Processes + transactions = distributed applications. In *Proceedings of the 7th Annual International Workshop on High Performance Transaction Systems* (*HPTS'97*) (Asilomar, Calif.). California, USA.

ALONSO, G., BLOTT, S., FEßLER, A., AND SCHEK, H.-J. 1997a. Correctness and parallelism in composite systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Database Systems* (*PODS'97*) (Tucson, Az.). ACM, New York, pp. 197–208.

ALONSO, G., FEßLER, A., PARDON, G., AND SCHEK, H.-J. 1999a. Correctness in general configurations of transactional components. In *Proceedings of the 18th Annual ACM Symposium on Principles of Database Systems* (*PODS'99*) (Philadelphia, Pa.). ACM, New York, pp. 285–293.

ALONSO, G., FEßLER, A., PARDON, G., AND SCHEK, H.-J. 1999b. Transactions in stack, fork, and join composite systems. In *Proceedings of the 7th International Conference on Database Theory*

(*ICDT'99*) (Jerusalem, Israel). Lecture Notes in Computer Science, vol. 1540, Springer-Verlag, New York, pp. 150–168.

Alonso, G., Fiedler, U., Hagen, C., Lazcano, A., Schuldt, H., and Weiler, N. 1999c. Wise: Business to business e-commerce. In *Proceedings of the 9th International Workshop on Research Issues in Data Engineering. Information Technology for Virtual Enterprises* (*RIDE-VE'99*) (Sydney, Australia). IEEE Computer Society Press, Los Alamitos, Calif., pp. 132–139.

Alonso, G., Hagen, C., Schek, H.-J., and Tresch, M. 1997b. Distributed processing over stand-alone systems and applications. In *Proceedings of the 23rd International Conference on Very Large Databases* (*VLDB'97*) (Athens, Greece). Morgan-Kaufmann, San Mateo, Calif., pp. 575–579.

Alonso, G., Vingralek, R., Agrawal, D., Breitbart, Y., Abbadi, A. E., Schek, H.-J., and Weikum, G. 1994. Unifying concurrency control and recovery of transactions. *Inf. Syst. 19*, 1 (Mar.), 101–115.

Beeri, C., Bernstein, P., and Goodman, N. 1989. A model for concurrency in nested transaction systems. *J. ACM 36*, 2 (Apr.), 230–269.

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

Bernstein, P., Shipman, D., and Wong, W. 1979. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. SE-5*, 3 (May), 203–216.

Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M., and Silberschatz, A. 1991. On rigorous transaction scheduling. *IEEE Trans. Softw. Eng. 17*, 9 (Sept.), 954–960.

Camp, J., Harkavy, M., Tygar, D., and Yee, B. 1996. Anonymous atomic transactions. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce* (Oakland, Calif.). The USENIX Association. pp. 123–133.

Chen, Q. and Dayal, U. 1996. A Transactional Nested Process Management System. In *Proceedings of the 12th International Conference on Data Engineering* (*ICDE'96*) (New Orleans La.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 566–573.

Chen, Q. and Dayal, U. 1997. Failure handling for transaction hierarchies. In *Proceedings of the 13th International Conference on Data Engineering* (*ICDE'97*). (Birmingham, England). IEEE Computer Society Press, Los Alamitos, Calif., pp. 245–254.

Dayal, U., Hsu, M., and Ladin, R. 1991. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Databases* (*VLDB'91*) (Barcelona, Spain). Morgan-Kaufmann Publishers, Reading, Mass., pp. 113–122.

Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. 1990. A multidatabse transaction model for interBase. In *Proceedings of the 16th International Conference on Very Large Databases* (*VLDB'90*) (Brisbane, Australia). Morgan-Kaufmann Publishers, Reading, Mass. pp. 507–518.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

Hagen, C. 1999. A generic kernel for reliable process support. Ph.D. dissertation. Swiss Federal Institute of Technology (ETH) Zürich. Diss. ETH Nr. 13114.

IvyTeam. 2001. IvyFrame—Process Modeling & Simulation Tool. IvyTeam, Zug, Switzerland, `http://www.ivyteam.com`.

Lazcano, A., Alonso, G., Schuldt, H., and Schuler, C. 2000. The Wise Approach to Electronic Commerce. *Int. J. Comput. Syst. Sci. Eng. 15*, 5 (Sept.), 343–355. Special Issue on Flexible Workflow Technology Driving the Networked Economy.

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

Lynch, N., Merritt, M., Weihl, W., and Fekete, A. 1994. *Atomic Transactions*. Morgan-Kaufmann, San Mateo., Calif.

Mehrotra, S., Rastogi, R., Silberschatz, A., and Korth, H. 1992. A transaction model for multidatabase systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems* (*ICDCS'92*) (Yokohama, Japan). IEEE Computer Society Press, Los Alamitos, Calif., pp. 56–63.

Moss, J. 1985. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Mass.

Moss, J. 1987. Nested transactions: An introduction. In *Concurrency Control and Reliability in Distributed Systems*, B. Bhargava, Ed. Van-Nostrand Reinhold Company, New York, Chap. 14, pp. 395–425.

O'Neil, P. 1986. The escrow transaction model. *ACM Trans. Datab. Syst.* (*TODS*) *11*, 4 (Dec.), 405–430.

Papadimitriou, C. 1979. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct.), 631–653.

Raz, Y. 1992. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Databases* (*VLDB'92*) (Vancouver, B.t., Canada). Morgan-Kaufmann Publishers, 292–312.

Reuter, A., Schneider, K., and Schwenkreis, F. 1997. ConTracts revisited. In *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds. Kluwer Academic Publishers, Chap. 5, pp. 127–151.

Rosenkrantz, D., Stearns, R., and Lewis, P. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Trans. Datab. Syst.* (*TODS*) *3*, 2 (June), 178–198.

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

Schek, H.-J., Weikum, G., and Ye, H. 1993. Towards a unifying theory of concurrency control and recovery. In *Proceedings of the 12th Annual ACM Symposium on Principles of Database Systems* (*PODS'93*). (Washington D.C.). ACM, New York, pp. 300–311.

Schuldt, H. 2001a. Process locking: A protocol based on ordered shared locks for the execution of transactional processes. In *Proceedings of the 20th Annual ACM Symposium on Principles of Database Systems* (*PODS'01*) (Santa Barbara, Calif.). ACM, New York, pp. 289–300.

Schuldt, H. 2001b. Transactional process management over component systems. Ph.D. dissertation, Swiss Federal Institute of Technology (ETH) Zürich. Diss. ETH Nr. 13976. Akademische Verlagsgesellschaft, Berlin, Germany and IOS Press, Amsterdam, The Netherlands.

Schuldt, H., Alonso, G., and Schek, H.-J. 1999a. Concurrency control and recovery in transactional process management. In *Proceedings of the 18th Annual ACM Symposium on Principles of Database Systems* (*PODS'99*) (Philadelphia, Pa.). ACM, New York, pp. 316–326.

Schuldt, H., Popovici, A., and Schek, H.-J. 2000. Automatic generation of reliable e-commerce payment processes. In *Proceedings of the 1st International Conference on Web Information Systems Engineering* (*WISE'00*) (Hong Kong, China). IEEE Computer Society Press, Alamitos, Calif., pp. 434–441.

Schuldt, H., Schek, H.-J., and Alonso, G. 1999b. Transactional coordination agents for composite systems. In *Proceedings of the 3rd International Database Engineering and Applications Symposium* (*IDEAS'99*) (Montréal, Ont. Canada). IEEE Computer Society Press, Los Alamitos, Calif., pp. 321–331.

Schuldt, H., Schek, H.-J., and Tresch, M. 1998. Coordination in CIM: Bringing database functionality to application systems. In *Proceedings of the 5th European Concurrent Engineering Conference* (*ECEC'98)* (Erlangen, Germany). pp. 223–230.

Schuler, C., Schuldt, H., Alonso, G., and Schek, H.-J. 1999. Workflows over workflows: Practical experiences with the integration of SAP R/3 business workflows in Wise. In *Proceedings of the Informatik'99 GI-Workshop Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications* (Paderborn, Germany). pp. 65–71. Tech. Rep. Nr. 99-07, University of Ulm, Department of Computer Science.

Schwenkreis, F. and Reuter, A. 1996. Synchronizing long-lived computations. In *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. V. Kumar, Ed. Prentice-Hall, Englewood Califs. N.J., Chap. 12, pp. 336–355.

Thomas, R. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Datab. Syst.* (*TODS*) *4*, 2 (June), 180–209.

Türker, C., Schwarz, K., and Saake, G. 2000. Global transaction termination rules in composite database systems. In *Proceedings of the 17th British National Conferenc on Databases* (*BNCOD 17*) (Exeter, U.K.). Lecture Notes in Computer Science, vol. 1832. Springer-Verlag, New York, 122–139.

Tygar, D. 1996. Atomicity in electronic commerce. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (*PODC'96*) (Philadelphia, Pa.). ACM, New York, pp. 8–26.

Tygar, D. 1998. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *Proceedings of the 24th International Conference on Very Large Databases* (*VLDB'98*) (New York, N.Y.). Morgan-Kaufmann, San Mateo, Calif., pp. 1–12.

Vingralek, R., Hasse-Ye, H., Breitbart, Y., and Schek, H.-J. 1998. Unifying concurrency control and recovery of transactions with semantically rich operations. *Theoret. Comput. Sci. 190*, 2 (Jan.), 363–396.

Wächter, H. 1991. ConTracts: A means for improving reliability in distributed computing. In *Proceedings of the 36th IEEE Computer Society International Conference* (*COMPCON SPRING'91*) (San Francisco, Calif.). IEEE Computer Society Press, Los Alamitos Calif., pp. 574–578.

Wächter, H. 1996. An architecture for the reliable execution of distributed applications on shared resources. Ph.D. dissertation. University of Stuttgart, Stuttgart, Germany (In German).

Wächter, H. and Reuter, A. 1992. The ConTract Model. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Ed. Morgan-Kaufmann Chap. 7, Reading, Mass., pp. 219–263.

Weihl, W. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput. 37*, 12 (Dec.), 1488–1505.

Weikum, G. and Schek, H.-J. 1992. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Ed. Morgan-Kaufmann, Reading MA., Chap. 13, pp. 515–533.

Zhang, A., Nodine, M., and Bhargava, B. 2001. Global scheduling for flexible transactions in heterogeneous distributed database systems. *IEEE Trans. Knowl. Data Eng.* (*TKDE*) *13*, 3 (May/June), 439–450.

Zhang, A., Nodine, M., Bhargava, B., and Bukhres, O. 1994. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD'94*) (Minneapolis, Minn.). ACM, New York, pp. 67–78.

Received March 2001; revised August 2001; accepted January 2002