Department of Computer Science
Series of Publications A
Report A-1999-2

Transactional Workflows

Juha Puustjärvi

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1999-2

Transactional Workflows

Juha Puustjärvi

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium III, Porthania, on August 19th, 1999, at 12 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-8642-5
Computing Reviews (1998) Classification: H.2.4, H.4.1
Helsinki 1999
Helsinki University Printing House

Transactional Workflows

Juha Puustjärvi

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
juha.puustjarvi@cs.helsinki.fi, http://www.cs.helsinki.fi/~puustjar/

PhD Thesis, Series of Publications A, Report A-1999-2
Helsinki, August 1999, 104 pages
ISSN 1238-8645, ISBN 951-45-8642-5

Abstract

The basic transaction model has evolved over time to incorporate more complex transaction structures and to selectively relax the atomicity and isolation properties. We study the relaxation of atomicity and isolation in the context of workflow systems.

For long lasting transactions the atomicity property is usually enforced by some form of compensation. A problem with compensation is that other transactions may access dirty (i.e., uncommitted) data. In order to avoid the problems of dirty data we introduce the notion of *options*. Instead of making dirty updates, workflows acquire options. An option is a certification of the success of a possible later update.

The implementation of options is based on the notion of a *two-phase task*. Such a task can also be used to support co-operative work. This leads to the more general notion of a *negotiation task*. Negotiation tasks are managed by a protocol which is similar to the two-phase commit protocol in that there is a request phase and a decision phase; the spectrum of the negotiation is not restricted to atomic commitment, however.

Serializability as an isolation criterion though well suitable for traditional transactions would overly restrict the concurrency of long lasting transactions. By using semantic information it is possible to weaken, or give up altogether, the serializability criterion, and yet ensure workflow execution correctness. The problem, however, is that the use of semantic information in concurrency control makes the specification as well as the management of workflows more complex. We alleviate this problem by introducing *self-isolating workflows*, which set and unset consistency constraints in the database in a way analogous to the way in which traditional transactions obtain and release locks. The advantages of using consistency constraints in workflow isolation are twofold. First, the expression power of consistency constraints in a database query language (SQL) is high, which in turn increases the potential degree of concurrency. Second, workflow concurrency control can be implemented by using the services of the underlying database management system.

Computing Reviews (1998) Categories and Subject Descriptors:
H.2.4 Systems: Concurrency, Transaction processing
H.4.1 Office automation: Workflow management

General Terms:
Workflows, Databases

Additional Key Words and Phrases:
Advanced transactions, Atomicity, Isolation, Consistency

Acknowledgements

Professor Martti Tienari originally directed my research and teaching to transaction processing. The TRANSCOOP (Transaction Management Support for Cooperative Applications) project, which was partially funded by the European Commission, gave me an opportunity of doing research work on advanced transaction processing.

My thesis continues the research on workflow systems started at the TRANSCOOP project. During this study, I have had valuable and stimulating discussions with Professors Hannu Erkiö and Seppo Sippu. I am grateful to them for their advice and support in both scientific and practical matters related to this thesis.

Contents

Chapter 1

Introduction

Business processes, or organizational processes as they are also called, are collections of activities that support critical organizational and business functions. Processing a purchase order, processing an insurance claim and processing a loan request are typical business processes.

Business processes are implemented as information processes and material processes. Material processes relate to human tasks and physical products. Information processes consist of tasks that create, process, manage, and provide information. The tasks may be more or less automated, e.g., a task performed by a human interacting with a computer is a partially automated task while a program taking order information as input and generating a bill is a fully automated task. Database systems, transaction processing monitors, and distributed systems provide the basic infrastructure for supporting tasks.

Today's business processes are dynamic: competition and economic pressures may require the cost of business processes to be reduced. Moreover, in order to increase customer satisfaction an enterprise may be forced to rapidly change the existing business processes or to develop new ones to better utilize the new information technology.

Reconsideration and redesign of business processes is called *business process reengineering*. Its complementary activity is called *information process reengineering*, which involves determining how to use legacy and new information systems and computers to automate the reengineered business processes. Both these processes can be performed iteratively, so as to provide mutual feedback.

The concept of a *workflow* was introduced to facilitate business process and information process reengineering and automation [GHS95]. A workflow specification describes business process tasks and their dependencies, and the requirements these tasks impose on information system functionality and human skills. For example, in Figure 1.1 the tasks of the workflow *Purchase order* and their dependencies are presented. The workflow starts when a phone call is received. In the first task, *Enter order*, a clerk enters the required information into an electronic form and records a new order. Then, two parallel tasks are processed: an *Inventory* task updates the inventory and a *Creditability* task checks the customer's credit information. After their successful processing another task, *Shipping*, generates the appropriate shipping orders, and finally, a *Billing* task charges the customer. Each of these tasks involves a transaction that reads and updates a database.

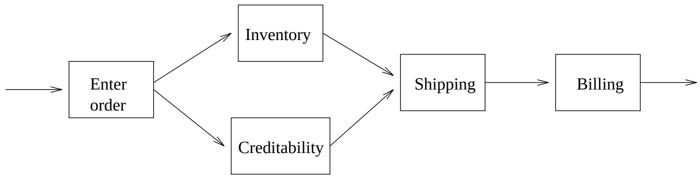

Figure 1.1: The structure of the workflow *Purchase order*.

The idea behind workflows can be traced back to job control languages of batch operating systems [RS95b, MSK+95]. Job control languages allowed the user to specify a job as a collection of tasks (or steps) such that each task was an invocation of a program, and the tasks were executed in sequence and some tasks could be executed conditionally. The idea of workflows also appears in office automation systems, which also separate work activities into well-defined tasks.

Early workflow models (e.g., [CC82]) included the notion of a task (also called an action, a step or a procedure). The models addressed neither isolation nor atomicity issues. The control flow between tasks was specified by triggers or Petri nets. Then, with the development of the information technology, the range of automated tasks was still extended. And now, workflow management technology (e.g., FlowMark [AKA+94, MAGK95], InConcert [Sar96], Panta Rhei [EGL98], MENTOR [WWW+97] and METEOR [KS95]) provides graphical editors for specifying workflows and facilities for atomic executions of workflows.

In the future, the effective use of the workflow concept will require that the computing environment should support the interoperability of heterogeneous, autonomous and distributed systems. This requires workflow technology to utilize the standards of distributed computing [GHM+93, MSKW96]. In addition, automating still more tasks requires addressing issues of workflow execution correctness and reliability, i.e., issues related to the transactional properties of workflows.

Most workflow-related research done in the past few years can be categorized into

- workflow design and specification (e.g., [EL95, FKB95, SR93, KS95]),
- inter-task dependency specification and management (e.g., [Kle91, CR92, CR93, DHL91, ANRS92, KRSRS96a]), and
- workflow management system design (e.g., [AMG+94, Sch93, Sar96, WWW+97, EL96, KS95, Kam98]).

In this thesis, we will restrict ourselves to workflow specification and workflow management system design. Particularly, we focus on the specification of the transactional properties of workflows and how they can be enforced by the workflow management system. The core of the problem is that guaranteeing transactional properties requires (independently of the method used) workflows to hold their data resources, causing other workflows that wish to use the same resources to wait. We therefore focus on developing transaction management methods having two important goals. First, the time a data resource is reserved should be minimized. Second, as few data resources as possible should be reserved. We aim to achieve both these goals by developing methods that exploit application semantics. We aim at methods that

- do not overly burden the workflow designer, and
- are implementable by using the services of existing database systems.

We therefore present protocols for stating the transactional properties of workflows. We also show how our protocols can enforce the transactional properties by utilizing features of SQL systems [Dat86].

Before presenting our method for managing transactional workflows we will review in Chapter 2 the problems related to transactional workflows as well as the proposed solutions. Managing transactional properties of workflows cannot be considered as an independent problem because it is in relation to other aspects including workflow modularity and reusability. Consequently in Chapter 3 we present our view of workflow modularity. Particularly, we focus on workflow structures and the notion of *task variations*. Ideally, a variation is a modification of a task that matches better the requirements of a particular workflow. The function of variations is twofold. First, task variations increase task reusability, and thereby decrease redundant workflow design. Second, task variations form the basic unit for transactional task integration. Particularly, we utilize a variation that externalizes certain states, and thereby allows the tasks to be integrated into larger transactional units.

In Chapter 4 a failure atomicity model is presented. A corner-stone in the model is the notion of an *option*. An option is a certification of the success of a potential later update. Instead of performing updates a workflow instance acquires options, and if the workflow instance can be committed then the options are eventually realized by installing the updates to the database. Through options we avoid the problems of dirty data.

Options are implemented by specific tasks, called *two-phase tasks* or 2P-tasks for short. A 2P-task divides task execution into two short-lasting ACID transactions, called the *preliminary transaction* and the *complementary transaction*, and thereby decrease the time the tasks hold their data resources. The only function of the preliminary transaction is to ensure that the complementary transaction will not fail due to failures or execution of other concurrent transactions.

We make a distinction between two kinds of options, called *public options* and *internal options*. In a public option, the effects of the preliminary transaction are made visible, meaning that the intermediate states of the tasks are modelled by the database. In contrast, in using internal options the effects of the preliminary transaction are hidden from other concurrent users. From the point of view of implementation, a public option and an internal option differ in that the former is implemented in an application while the latter is implemented by the workflow management system.

The coordination of 2P-tasks is carried out by a failure atomicity protocol presented in Chapter 5. The protocol is similar to the two-phase commit (2PC) protocol [BHG87] in that it consists of a vote phase and a decision phase. It differs from the 2PC protocol in that its communication topology is not restricted to two levels. Nor does the commitment require yes-votes from all the participants. In addition, a modification of the failure atomicity protocol is presented in which a 2P-task can be replaced by a compensating transaction, a long lasting ACID-transaction, or a manually executed task. So the transactional properties of each task are determined according to the implementation method used. Which method is most appropriate depends on the semantics of the corresponding task.

The range of the applicability of 2P-tasks is wide. In the context of workflow recovery (i.e., in ensuring failure atomicity) we attach two complementary tasks to each preliminary task; which one is executed depends on whether the workflow is committed or aborted. By allowing more than two complementary tasks we can support *negotiation workflows*, to be presented in Chapter 6. For example, a business process where dealing is made by first requesting an offer and then making a deal can be carried out by such a negotiation. The function of 2P-tasks in negotiation is to process the database operations of the participating tasks: the preliminary task is executed before the response is sent, and after the coordinator's decision is received, an appropriate complementary task is executed.

Mere failure atomicity is not a sufficient correctness criterion for workflows. Workflows should also behave correctly in the presence of other concurrent workflows. Our approach for workflow concurrency control is presented in Chapter 7. Particularly, the notion of a *self-isolating workflow* is introduced. Such a workflow sets and unsets *restrictions* in a way similar to the way in which traditional transactions obtain and release locks. A restriction is a temporary consistency constraint imposed on the data items on the database (a data restriction) or on the state information of tasks (a task restriction). Data restrictions are semantic locks that control the modifications of data items, whereas task restrictions are used to control the execution of the programs behind the tasks. Thus the interleaving of workflows is carried out by controlling the accesses to the database as well as by restricting the execution of tasks. An important aspect is that both kinds of restrictions can be expressed as assertions in the SQL database language. As a result, restrictions have a strong expression power, and restrictions can be enforced by the services of the underlying database system.

Finally, Chapter 8 gives some concluding remarks, and some suggestions for further research.

Parts of the research reported in this thesis have been presented in separate papers. The issues of task reusability and task variations presented in Chapter 2 are also discussed in [PTV96a, PTV96b, PTV97]. The negotiation workflow model presented in Chapter 6 is also discussed in [Puu97]. The early ideas of using variations of the 2PC protocol in relaxing the isolation properties of long lasting transactions can be traced back to [Puu91, Puu92].

Chapter 2

Overview of transactional workflows

In this chapter we will briefly discuss the evolution of transaction models and recognize the relevance of transactions to workflows. Particularly workflow recovery and isolation are discussed, and different approaches to these issues are overviewed.

2.1 Advanced transactions and workflows

The concepts of an atomic transaction and a serializable history were introduced when only centralized database systems existed. As distributed database systems developed, these concepts were extended to distributed systems as well. This traditional approach is ideal in that it ensures that transactions behave correctly in the presence of failures and other concurrent transactions. In addition, serializability and atomicity as correctness criteria are intuitive and clear.

Fundamentally, the only problem with atomic transactions and serializable histories is that they overly restrict the performance of the system, and thus are inadequate for many applications. Motivated by this, several *advanced transaction models* (ATMs) have been proposed. Advanced transactions differ from traditional (flat) transactions in that they structure the transaction as a set of subtransactions, thus forming a transaction tree.

Tree-structured transactions exhibit two important properties. First, they allow to specify appropriate execution structures: subtransactions can be executed in parallel or in a predefined order. Second, a tree structure allows finer specification of isolation and atomicity: the unit of atomicity as well as the unit of isolation may concern the whole transaction or one or more of its subtransactions.

Modelling a workflow as an extended transaction means that the subtransactions correspond to the tasks of the workflow and the execution structure of the extended transaction (augmented by control-flow specification features if needed) corresponds to the control-flow of the workflow. Hence, the isolation and atomicity properties of the workflow are determined according to the ATM used.

Many ATMs have been proposed for modelling workflows, e.g., nested transactions in [DHL90, DHL91, KLLO96], multi-parent subtransactions in [KRSRS96b, KRSRS96c], open nested transactions in [BDS+93, Wei93], multilevel transactions in [BÖH+92], sagas in [GMGK+90] and flexible transactions in [ANRS92].

Nested transactions [Mos85] provide full isolation and atomicity with respect to a whole transaction, but internally they provide finer control over failures. For the recovery of subtransaction failures the model provides four alternatives:

- the execution of the failed subtransaction can be resumed,
- the whole transaction can be aborted,
- the failure can be ignored, i.e., the subtransaction is considered *non-vital*, or
- another subtransaction may be initiated, i.e., the subtransaction is considered as a *contingency subtransaction.*

Multi-parent subtransactions [KRSRS96b, KRSRS96c] extend the nested transaction model by allowing every subtransaction to have an arbitrary number of parent transactions. A subtransaction may be executed in parallel to some or all of its parent transactions. So the model allows multiple transactions to start a subtransaction in cooperation. The main advantage of the model is that composite events (i.e., events representing a combination of other events) can be embedded into transactions.

Open nested transactions [WS92, WDSS93] still relax the isolation of the subtransactions. Ideally, an uncommitted open nested transaction allows other transactions to access the results of its committed subtransactions if the access does not exclude the possibility to abort the nested transaction. This is the case if the committed subtransaction and the transaction accessing the data are *commutative*, meaning that the final state of the database is independent of their execution order.

Multilevel transactions [Wei91, WS91] still relax the requirement of isolation by not requiring the commutativity. Subtransactions can commit and release their resources before the whole transaction commits. If the whole transaction (root) aborts then the (semantic) atomicity is ensured by executing compensating transactions which semantically undo the effects of the committed subtransactions.

Also *sagas* [GMS87] and *flexible transactions* [ELLR90, RELL90, ANRS92, BEK93, KPE92] relax strict atomicity by introducing compensating transactions. Flexible transactions may also incorporate alternative subtransactions. In addition the designer can specify the acceptable termination states for flexible transactions. With sagas the application programmer defines a linear execution sequence of the subtransactions consisting of traditional flat transactions, and if the execution of a subtransaction fails then the executed subtransactions are compensated in the inverse order. *Nested sagas* [GMGK+91] as their name indicates also allow nesting structures within a saga.

Asynchronous transactions [MD89, CW90, CW92b, CW92a] are developed for loosely-coupled multidatabases which may be temporarily in an inconsistent state. Asynchronous transactions require that inter-database consistency requirements are specified by a collection of descriptors. These descriptors are used to convert a transaction updating a data item in a single database into transactions that spawn or trigger various activities needed to maintain inter-database consistency. The concept *polytransactions* [SRK92] is developed for such an environment.

Cooperative transaction models are related to groupware systems [EGR91, Ell92]. A groupware system supports a group of users engaged in a common task. It aims to assist groups in communication, in collaborating and in coordinating their activities. Cooperative software engineering, engineering design and text editing are typical applications of groupware systems. In such cooperative environments transactions cooperate by having intersecting views, and by allowing the effects of other's operations to be visible without producing conflicts. Cooperative transactions do not provide compensation. Concepts such as *split transactions* [Kai90, KP92, PH88], the *cooperative transaction model* [NRZ92] and *lazy consistency criterion* [NG92] are developed for transaction management in cooperative environments.

Transaction metamodels [CR90, Kle91] provide a common framework within which one can specify and reason about the nature of interactions between transactions in a particular model. Using a metamodel existing transaction models can also be described and compared.

Due to the limited execution structure (control-flow) specification capabilities of most ATMs, workflow models based on an ATM have been augmented by control-flow specification features. In general, control-flow specification can be represented by a set of dependencies concerning the abortion and commitment of transactions [Kle91]. However, a fundamental problem in modelling workflows by ATMs is that ATMs provide a predefined set of properties, which are neither necessary nor meaningful for all workflow applications [WS96]. For example, viewing purchase order processing in an ATM would produce the transaction tree with the subtransactions *Enter order*, *Inventory*, *Creditability*, *Shipping* and *Billing* (Figure 2.1). Further,

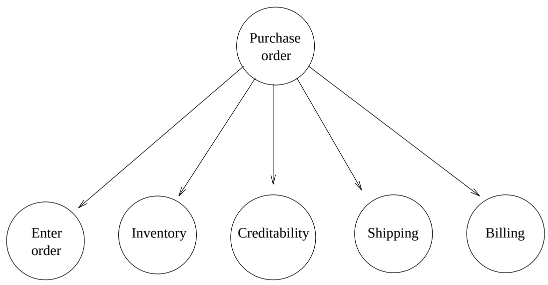

Figure 2.1: The workflow *Purchase order* structured as a transaction tree.

assuming that the transaction tree represents a saga, the execution structure is a predefined linear execution order of the tasks. However, this is not the case with many workflows, such as the Purchase order. Motivated by this also frameworks for tailoring appropriate ATMs for workflows have been proposed [GHM+93, KU93, BDG+94]. In such frameworks ATMs' structures are described in terms of dependency descriptors between constituent transactions [GH94]. Further, new ATMs can be developed from existing ATMs by composing their dependency descriptors.

2.2 Transactional workflows

The term *transactional workflow* is used to emphasize the relevance of transactional properties for workflows. A transactional workflow involves coordinated execution of tasks which may require access to heterogeneous, autonomous and distributed database systems. The coordination requirements are expressed by control-flow dependencies, which specify a precondition for the execution of each task. The preconditions are based on

- the execution states of other tasks (i.e., whether they are active, committed or aborted),
- output parameters of other tasks, or

- on external variables (e.g., on time).

For example, the precondition of the task *Shipping* could be the commitment of the task (transaction) *Inventory*, and a positive value of the output parameter "creditable" from the task *Creditability*.

Due to the expression power of control-flow dependencies, the semantics of transactional workflow models are richer than those of ATMs. As a matter of fact, workflow systems can be used to implement ATMs. For example, FlowMark [MAGK95, MAA+95, AAEA+96] has been used in implementing sagas and flexible transactions. However, workflow models are not only general frameworks for implementing ATMs, as they complement many aspects (e.g., interaction with manual activities) of ATMs, which makes them more suitable for building workflow applications.

A salient feature of transactional workflows is that the workflow designer has to specify appropriate recovery and isolation properties for workflows. The nature of the specification depends on the recovery and isolation capabilities the workflow system provides.

2.2.1 Workflow recovery

The traditional notion of recovery would require that a failure of any task result in the failure of the workflow. However, analogous with ATMs, a workflow may survive certain failures. In addition, some failures may require human intervention (exception handling). For example, during the execution of a long-lasting workflow the underlying database schema may change (e.g., a relation is dropped), and as the workflows are not designed to cope with such modifications human intervention is required for the recovery. Consequently workflows may require much more comprehensive solutions to recovery than ATMs.

In the context of workflows it is necessary to distinguish semantic failures (also called logical failures) from system failures. A semantic failure occurs when a task cannot reach its objective. For example, the task *Inventory* would fail semantically when a non-existent product is tried to be removed from the database. With semantic failures the transaction itself issues an abort-operation as it fails in achieving its objective.

A system failure causes the loss or corruption of main memory. It may occur, e.g., when electric power fails. Analogous to how transactions use a transaction log to cope with system failures, workflows use a workflow log to cope with system failures. A transaction log is used to restore the last committed values of data items while a workflow log is used to guarantee the continuation of interrupted workflows. This requires that the workflow log contains information of the executed tasks. In addition, aborting tasks or workflows that have read data values updated by aborted tasks or workflows (i.e., processing cascading aborts) requires that the log contains information about the workflows that have accessed the data updated by active workflows.

To illustrate the notions related to recovery we have augmented the *Purchase order* workflow in Figure 2.2. The task *Inventory order* is added, its precondition being the semantic failure of the task *Inventory*. Further, an order can be sent to two producers, represented as tasks *Producer A* and *Producer B*. If *Producer A* fails then the order is sent to *Producer B* (an alternative task).

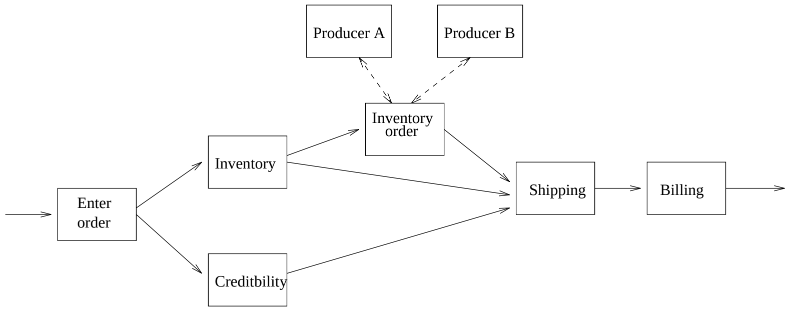

Figure 2.2: The execution structure of the augmented workflow *Purchase order*.

If a workflow does not survive a task failure, then the workflow is aborted. As the tasks may have externalized their updates they cannot be simply rolled back by restoring before images [BHG87]; instead, they have to be undone by executing compensating actions. Compensating actions may also be needed although the workflow would not be aborted. This happens if the workflow has alternative paths, and a task in a path fails causing the compensation of preceding tasks in the path. The compensation of a task may also require a cascading abort, i.e., abortion of the workflows which have accessed the data updated by the task to be compensated. Further, the success of the compensation should be ensured. For example, if withdrawing is the compensating activity of deposit, then it should be ensured that the balance of the account is at least the amount of the withdrawing.

2.2.2 Workflow isolation

A natural and sufficient criterion for workflow isolation correctness is that the execution of workflows is serializable, i.e., equivalent to a serial execution. However, the serializability requirement of long-lasting activities overly restricts the performance of the system.

Generally, workflow isolation (execution atomicity) is a tradeoff between

- workflow execution correctness,
- workflow system performance, and
- the simplicity of workflow specification and management.

By using semantic information it is possible to weaken, or give up altogether, the serializability criterion, and yet ensure workflow execution correctness. However, analogous with traditional semantic concurrency control models [Lyn83, GM83] the use of semantic information makes the specification as well as the management of the system more complex.

To illustrate workflow isolation requirements, we give three isolation requirements for our purchase order workflow and consider the effects of their violations.

- Businesses often enumerate events with unique sequence numbers, and so there may be a requirement that those numbers (given in the task *Enter order*) must constitute a monotone series with no gaps. So a new order number cannot be issued until it is sure that the previous workflow will not fail. However, as a workflow may fail at any time during its execution, the only way to ensure that there are no gaps is to execute the workflows serially.
- Assume that the balance of unpaid bills of a client has a predefined upper limit. The balance is checked in the task *Creditability*, and if the new order would cause an overdraft then the workflow is aborted (a semantic failure). Otherwise, the new balance is updated in the task *Billing*. Now, to ensure that the limit will not be exceeded as a result of two or more concurrent workflows of the same customer, the workflows pertaining to the same customer should be processed in a serializable way, or at least the tasks *Creditability*, *Shipping* and *Billing* pertaining to the same client should be processed in a serializable way.
- Assume that the existence of an ordered product is checked in the task *Inventory*, and that the ordered product is not removed from the inventory database until in the task *Shipping* (after the customer's creditability has been checked). Now, to ensure that the ordered product is still in the inventory the tasks *Inventory* and *Shipping* should be executed as one transaction, or at least they should constitute a unit

of isolation. However, this is not necessary if the workflow anticipates this case. That is, the workflow is either aborted as a result of a semantic failure or the task *Inventory order* is executed.

We make the distinction between four approaches for specifying isolation properties for workflows. We call them: *ATM-approach*, *transactional task integration*, *semantic locking*, and *conflict specification*. The two first represent syntactic while the two latter represent semantic approaches to concurrency control.

ATM-approach. By the ATM-approach we refer to the isolation features provided by the ATMs. That is, each workflow is modelled as a transaction. Therefore, the isolation and the atomicity properties of a workflow are intuitive and clear. However, workflows are so diverse in their requirements that no single ATM is sufficient to meet the needs of all workflows. In addition, different workflows often have incompatible requirements which makes using a single ATM impossible.

Transactional task integration. To better match the application's transactional requirements some workflow models [TV95, WR92] allow one or more tasks to form an ACID-transaction. So using transactional task integration the unit of isolation may vary from a single task to the whole workflow. We also regard this approach as syntactic as no application semantics are used in the specification of the isolation constraints.

Semantic locking. By semantic locking [PTV96a] we refer to the approaches in which some kind of semantic lock is expressed in the terms of an application. The isolation constraint is expressed as a predicate, which has to be valid during the execution of the workflow. Generally, using predicates instead of requiring transactional properties a much higher degree of concurrency can be achieved. For example, a predicate could be set which ensures that at least one unit of a certain product will be in the inventory. This is a much more liberal constraint than locking a part of the inventory database.

Conflict specification. By conflict specification we refer to the isolation approaches (e.g., [WDSS93, BDS$^+$93]) where a workflow designer specifies a conflict matrix concerning the tasks of the workflows. That is, two tasks conflict if their order of execution affects either the state of the database or the value that one of them returns. The matrix and its use is analogous with the Read/Write conflict matrix [BHG87] used with the traditional syntactic concurrency control model. Consequently, the scheduler produces conflict serializable executions. That is, execution of workflows is serializable, if an equivalent serial execution exists with the same ordering of conflicting tasks. As the conflict specification utilizes the semantics of the application, this approach allows more concurrency than the traditional syntactic (Read/Write) concurrency control model [LMWF94, BHG87].

A desirable feature of workflow concurrency control is its ability to support more than one approach for the specification of workflow's isolation requirements. For example, the contract model [Wäc91, WR92, SR96] supports transactional task integration and semantic locking (invariance predicates).

2.2.3 Characterizing transactional workflows

A generally used taxonomy distinguishes between administrative, ad hoc, collaborative and production workflows [AAEAM97]. This classification is mainly based on

- workflow complexity, and
- repetitiveness and predictability of workflows and tasks.

Administrative workflows involve repetitive, predictable processes where the steps to follow are well established. For example, processing a purchase order of Figure 1.1 is a typical administrative workflow. *Ad hoc workflows* differ from administrative workflows in that they are developed to deal with unique situations which require rapid workflow solution [GHS95]. Workflow management systems that support ad hoc workflow are also called groupware.

A characteristic feature of *collaborative workflows* is that they may involve several iterations, whereas other types of workflows are based on a forward progress. For example, writing an article by several authors may require several writing iterations before being complete, and thus it can be classified as a collaborative workflow.

In this thesis, we will restrict ourselves to *production workflows*. They involve repetitive, predictable and complex processes. Furthermore they typically access many heterogeneous and autonomous information systems. Also the nature and the transactional requirements of interoperability may be diverse. To illustrate this, we have augmented the purchase order example in Figure 2.3 by three interoperable tasks, each having different kinds of transactional requirements.

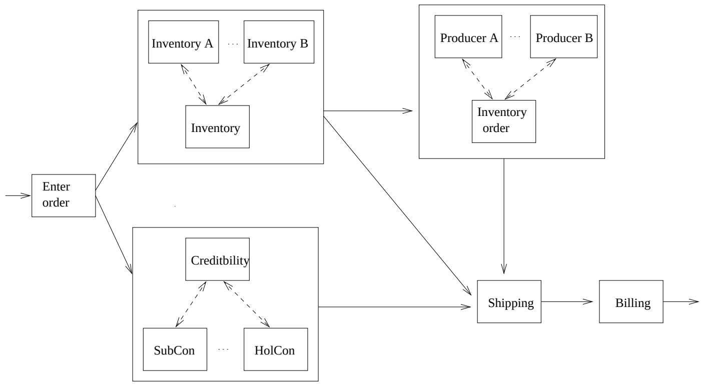

Figure 2.3: Distributed structure of the workflow *Purchase order*.

- Assume that the enterprise has many inventories, each having a local inventory system. So, the order may be comprised of several products located in different inventories. Further, due to the atomicity of the order, also the updates of the local databases should comprise an atomic distributed transaction.

- Assume that if the value of the order exceeds a predefined limit and the creditability of the customer has not previously been assessed, then the assessment is done by communicating subsidiaries and holding companies. In this case the interoperability does not require distributed updates, but only the activation of remote activities.

- The processing of the task *Inventory order* may require different kinds of communication between two or more producers. In Figure 2.3 the tasks *Producer A* and *Producer B* represented alternative tasks. However, the required interoperability may be more complex than that. For example, ordering may consist of first sending an offer request to certain producers, and then making the order based on the received offers.

The form of task interoperability depends on the support of the distributed system. For example, DOMs (Distributed Object Management Systems) [GHM+93] support distributed transactions, FlowMark [KAGM95, AAEAM97] uses the 2PC-protocol [BHG87] for managing replicas, whereas MENTOR [WWW+97, WWWD96] and METEOR [KS95] use remote procedure calls (RPC) [GR93] and TP-monitors [GR93] for supporting task interoperability.

Chapter 3

Structuring workflows

We shall view a workflow as a tree structure. Such a structure also nicely fits with the goals of task reusability and task variations, as will be illustrated in Section 3.1. Particularly the notions of workflow tree and workflow instance tree are introduced in Section 3.2. The relationship of workflow trees and hierarchical workflow management is briefly discussed in Section 3.3. The relationship of the hierarchical workflow management with the other components of the workflow system is discussed in Section 3.4.

3.1 Task reusability and task variations

An appropriate programming model should support code reusability. In the context of workflows this means that a given task may be used in several workflows. However, different workflows may have minor differences in their requirements on the functionality of a task, which in turn may restrict the reusability of a task. This suggests that task reusability can be increased by allowing *task variations*. As we will illustrate the variations of a task may differ either in the unit of work they have to do or in their transactional properties.

From an architectural point of view we model workflows by a modular structure consisting of a *workflow pool* and a *task pool* (Figure 3.1). Modular workflow specification also applies to a distributed environment. Then each site has its own workflow and task pools, but a task may also belong to a workflow locating in the workflow pool of another site.

Each task belongs at least to one workflow. If a task belongs into more than one workflow, then the task is *reused*. A task may also be comprised of subtasks as presented in Figure 3.2. Therefore a task may also be a *subworkflow*. Respectively, if a subworkflow belongs into more than one

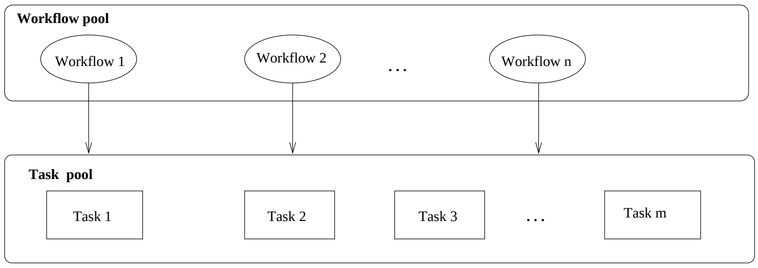

Figure 3.1: Modular component architecture.

workflow, then the subworkflow is reused.

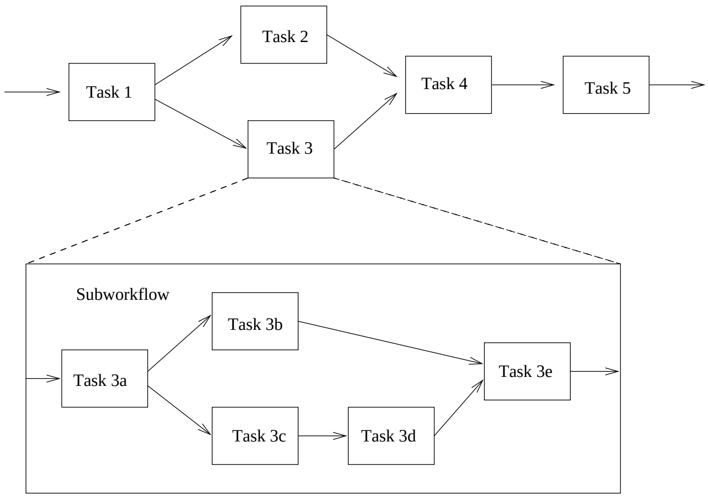

Figure 3.2: A task representing a subworkflow.

We will illustrate task variations by two workflows in a banking environment. These workflows share a task named *Credit check*, which as its name suggests, evaluates client creditability. However, as all the variations are comprised of subtasks we could also call them subworkflows.

A variation *Credit check variation A* is presented in Figure 3.3. Its task *Enter client information* accepts the information of the client. Based on this information, the task *Client's data access* retrieves from the bank's database the relevant client information. This information includes, among other things, the amount of loan the client currently holds in the bank and the balances of the client's accounts. Then the task *Client's credit evaluation* determines the creditability of the client.

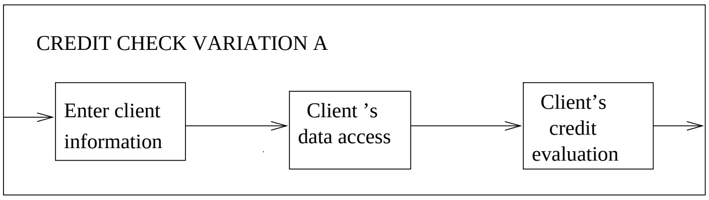

Figure 3.3: Workflow *Credit check variation A*.

Another variation, named *Credit check variation B* (Figure 3.4), has an additional functionality: the subsidiaries and holding companies in which the client has liabilities are also retrieved, and if such liabilities exist the subsidiaries or holding companies are contacted to get their assessments of the client. These are performed by the tasks *Subsidiaries' contacts* and *Holding companies' contacts*, which also provide input for the task *Client's credit evaluation*.

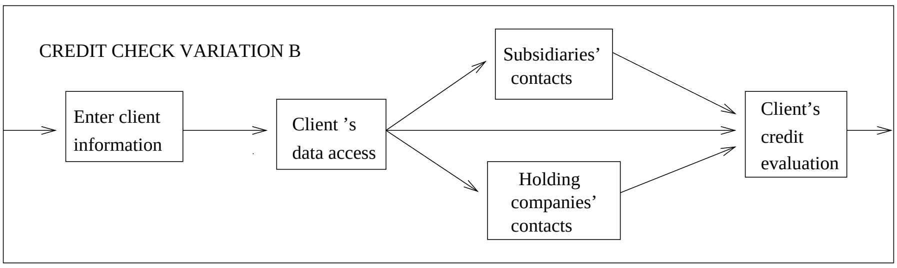

Figure 3.4: Workflow *Credit check variation B*.

We now consider the use of these two variations in two workflows named *Credit account request* (Figure 3.5) and *Loan request* (Figure 3.6).

In the workflow *Credit account request* the first task *Enter credit account request* accepts the requested limit and the name and SSN (social security number) of the client. In the second task *Client's accounts* it is checked whether the client has an existing credit account in the bank. If the client has such an account then in the task *Rejection* the request is returned and the client is advised to request the increment of the credit limit, i.e., a client is not allowed to increase his/her credit limit by acquiring many credit accounts. Otherwise, the task *Credit check* is executed. Which of its two variations, A or B, is executed depends on the credit limit of the request. Finally, if the credit account is granted in the task *Enter credit account decision*, then the client's data is updated by the given limit in the task *Client's data update*.

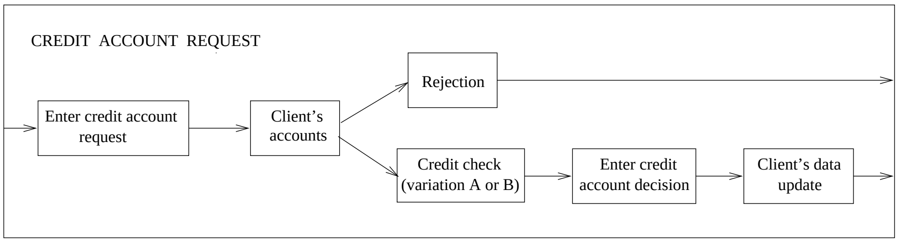

Figure 3.5: Workflow *Credit account request*.

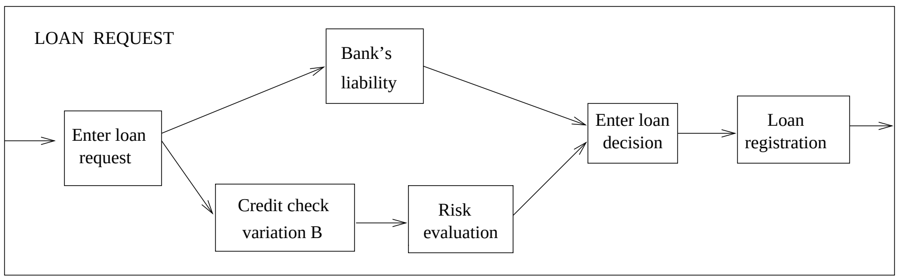

Figure 3.6: Workflow *Loan request processing*.

In the workflow *Loan request* the first task *Enter loan request* accepts the requested loan amount and the name and SSN of the client. Based on that information the subworkflow *Credit check variation B* is processed, which creates input for the task *Risk evaluation*. In risk evaluation, the interest for the loan is computed from the amount of the loan and the client's creditability. Then the task *Enter loan decision* makes the decision to either grant or refuse the loan request. However, before the loan can be granted the task *Bank's liability* checks that the bank does not exceed its liability limit, and if the limit will not be exceeded, the bank's total liability will be increased by the amount of the loan. If the loan is granted, the task *Loan registration* finally adds the information of the granted loan to the client's and the bank's data, and increments the bank's total liability by the amount of the loan.

The variations A and B of the task *Credit check* differ in their functionality. In addition, there may be a need to introduce new variations which differ as regards their transactional properties. To see this assume that a client (which may be a human or an enterprise) has two concurrent loan requests, say *loan request 1* and *loan request 2*. The decision of granting a loan is based among other things on the information of the client's existing loans, which are retrieved in the subworkflow *Credit check*. Therefore, if both the loan requests are granted, then in granting the latter loan the first loan should be known. However, this will not be the case if the requests were processed simultaneously.

This example illustrates that neither of the variations A or B of *Credit check* is sufficient for the workflow *Loan request*. A simple solution is to lock the client's data at the task *Credit check* and to hold the locks until the end of the workflow. However, such a solution would not work in practice as it would prevent the access to the client's account and the payings of the loans. We will consider more appropriate solutions to this problem in the context of *self-isolating workflows* (Chapter 7), which allow us to specify new variations that deviate only in their isolation properties.

3.2 Workflow trees and workflow instance trees

So far we have represented a workflow's structure by a task precedence graph. As such a graph does not express the tasks' internal structure, we will also use tree structures for representing workflows.

In a *workflow tree* the root represents the whole workflow, the nodes represent subworkflows and the leaves represent tasks. As a subworkflow may have another subworkflow as its component, the tree structure may be arbitrarily high. In Figure 3.7 a three-level tree representing the *Purchase order* workflow is presented (c.f. Figure 2.3).

We also represent the structure of a workflow instance by a tree, called an *instance tree*. As some tasks of a workflow may be optional, the execution of a workflow may yield different kinds of instance trees. For example, in the workflow *Purchase order* the subworkflow *Inventory order* is executed only if the subworkflow *Inventory* fails semantically. Further, in the subworkflow *Creditability* the tasks *Subsidiaries' contacts* and *Holding companies' contacts* are executed only if the client has certain liabilities. So an instance

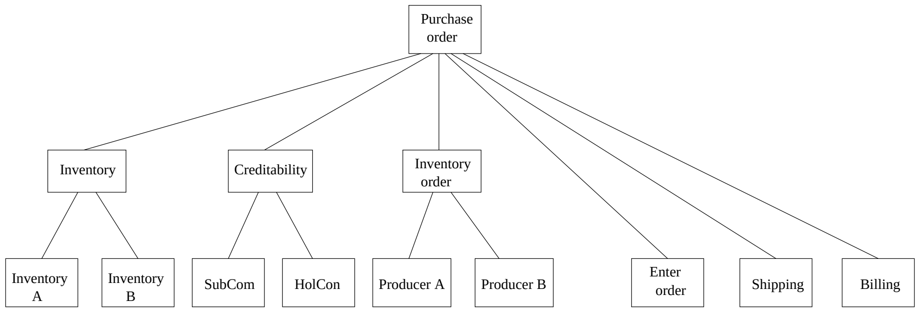

Figure 3.7: A workflow tree.

tree could be the one presented in Figure 3.8.

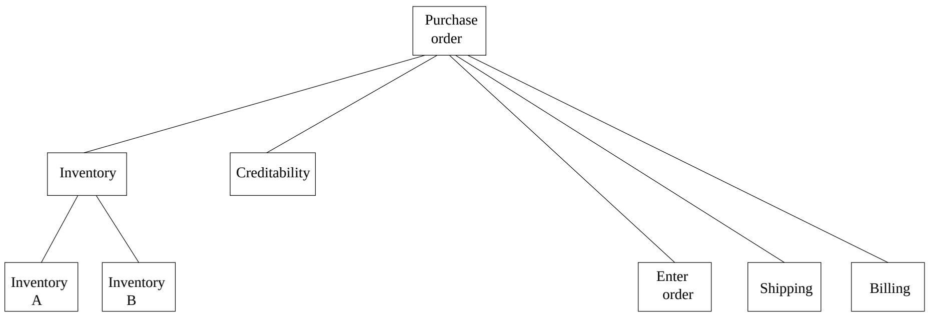

Figure 3.8: A workflow instance tree.

The presence of a task (leaf), a subworkflow (node) or a workflow (root) in an instance tree only indicates that such a component has been in execution. It does not indicate whether it failed or not.

3.3 Hierarchical workflow management

In our workflow model, there are two kinds of tasks:

- *actual tasks*, which comprise the function of the workflow, and
- *transactional tasks*, whose function is to provide transactional properties for workflows.

The transactional tasks are further divided into two groups:

- *communicating tasks*, whose function is to ensure failure atomicity, and
- *isolating tasks*, whose function is to ensure execution atomicity.

The communicating tasks of each workflow instance comprise a hierarchical communication structure, similar to the workflow's instance tree. The task attached to the root is called the *coordinator*, the tasks attached to subworkflows are called *inter-coordinators* and the tasks attached to the leaves are called *participants*. So, in the communication structure the coordinator may have inter-coordinators and participants as its children, and each inter-coordinator may have other inter-coordinators and participants as its children. For example, the workflow instance tree of Figure 3.8 would give rise to the communication structure presented in Figure 3.9.

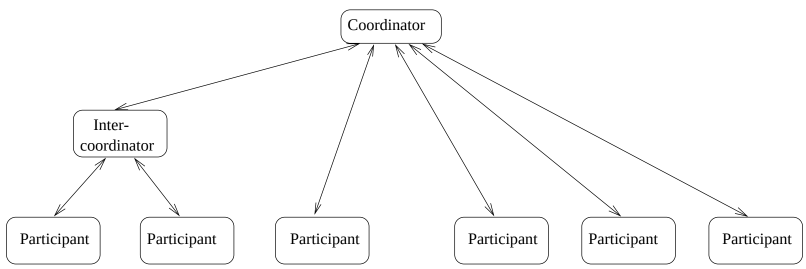

Figure 3.9: The communication structure of a workflow instance.

The communicating tasks differ from other tasks in that the workflow designer does not have to specify their functions; they are standard tasks that the workflow system adds to workflow instances. In contrast, the workflow designer has to specify the isolating tasks by giving the isolation constraints. As isolating tasks are subtasks of other workflow components, they are not presented in the hierarchical structure.

The communicating tasks are the main topic of Chapters 4 and 5, whereas the isolating tasks are the main topic of Chapter 7.

3.4 Workflow system architecture

Our distributed workflow system architecture incorporates the following functions:

- workflow specification,

- workflow scheduling (enforcement of inter-task dependencies),
- workflow concurrency control (execution atomicity), and
- workflow recovery (failure atomicity).

The *workflow specification interface* is a module which provides the interface between the workflow designer and the workflow system (Figure 3.10). A *workflow designer* specifies a workflow. The designer may use a specific language or a graphical editor. The specification is then handled by a *workflow specification interface*, which generates ECA-rules [UW97] to represent the intertask dependencies; these rules are then stored in a workflow pool and a task pool.

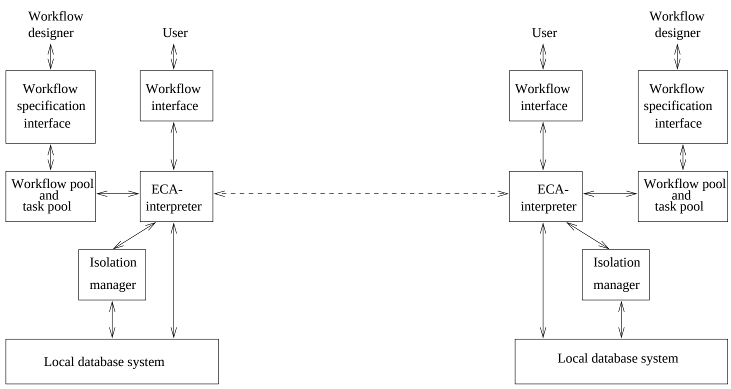

Figure 3.10: Distributed workflow system architecture.

The workflow specification interface may also analyze the specification to determine whether such a workflow can be implemented. For example, if the specification includes a reusable task, but such a task is not found in the task pool, then a mismatch is reported to the workflow designer.

Based on the ECA-rules stored in the workflow pool the *ECA-interpreter* schedules the tasks of each workflow instance. That is, when the precondition of a task is satisfied, then the ECA-interpreter starts the execution of the task. The preconditions may be defined through dependencies involving the state of other tasks or the output values of other tasks. So all the possible tasks in a workflow and their dependencies must be known in advance, but only those tasks whose preconditions are satisfied are started by the ECA-interpreter.

The *workflow interface* is a module which provides the interface between the user and the workflow system. A user activates a workflow instance by passing a workflow name and its parameters to the workflow interface, which passes them to the ECA-interpreter. The ECA-interpreter activates the tasks by communicating with the entities that process the tasks. Typically the processing entity is a database system or a human using the database system. However the isolation tasks are processed by the *isolation manager*, which in turn uses the services of the database system. Note that in the architecture there is no system module for ensuring failure atomicity, since in our approach the functions required for failure atomicity are included in the communicating tasks.

As a subworkflow may be a component of a workflow, the ECA-interpreter may also activate another workflow. If that workflow is located at another site, then the task is activated through the ECA-interpreter of that site.

Chapter 4

Two-phase tasks and options

The corner-stone in our approach for achieving failure atomicity is the notion of an option. An option is a certification of the success of a possible later activity. Typically such an activity is a database update but it may also be a manual activity. A task may request an option from the workflow system; the option may be granted only if the system is sure that the option can indeed be realized (e.g., an update can be executed).

Section 4.1 motivates the use of options by showing the drawbacks of other related approaches. A decomposition of a task suitable for supporting options is presented in Section 4.2. This leads to the introduction of the notion of a two-phase task. The effects of two-phase tasks upon subworkflow and workflow structures are considered in Sections 4.3 and 4.4.

A two-phase-task can be implemented by an application or by the workflow system. Depending on the implementation strategy we make a distinction between two kinds of options: public options (Section 4.5) and internal options (Section 4.6). As their names suggest, the two kinds of options differ in whether the effects of options are visible to other concurrent activities or not. We also show how the both types of options can be implemented in a SQL-system.

4.1 Motivation

The transactional approaches for failure atomicity can be categorized into two classes:

- A workflow is executed *atomically*, if other workflows are not allowed to see the effects of an active (uncommitted) workflow. The solutions based on open nested transactions [WS92, WDSS93] and traditional atomic transactions [BHG87], for example, belong to this class.

- A workflow supports *semantic atomicity* [GM83], if immediately after the commitment of a subtransaction (e.g., a task) other transactions are allowed to access the updated data (dirty data) [BHG87], though the workflow may still be aborted. Abortion is processed by executing compensating transactions for each subtransaction that has already committed. So each compensating transaction "undoes', according to the semantics of the application, the effects of the subtransaction being compensated. For example the transactional workflows based on flexible transactions [ELLR90, RELL90, ANRS92, BEK93, KPE92] and sagas [GMS87] support semantic atomicity.

A problem with semantic atomicity is that the access to dirty data may yield inconsistent results, or may cause a subsequent compensating action to fail. On the other hand, if the effects of dirty data are avoided by cascading aborts, then the workflow management system has to keep track of all workflows which have accessed dirty data. In the case of workflow abortion, all the tasks which accessed dirty data and all their successors have to be aborted or compensated.

Our main idea is to support failure atomicity of workflows, and yet avoid the problems of dirty data and cascading aborts.

First we will motivate our ideas by a variation of the traditional travel-planning example [Gra81], which has become the example of failure atomicity choices. The example was originally presented by Gray [Gra81] to illustrate that the subtransactions of a transaction may have different kinds of transactional requirements, so that some subtransactions may be undone or redone, while other subtransactions may be deferred but not undone. Later on, the contract model [WR92], alternative subtransactions [ELLR90] and contingency transactions [BÖH+92] have been motivated by that example. Again, we motivate our two-phase tasks using the same example.

The execution structure of the workflow *Travel-planning* is presented in Figure 4.1. The workflow starts when a travel agent receives a call to reserve tickets for a travel. An *Enter order* task involves recording the new order. Then two parallel tasks, *Flight reservation* and *Hotel reservation*, are executed. A *Car reservation* task takes as input the flight's arrival time, and thus it is executed after the *Flight reservation* task. Finally, a *Billing* task is executed in order to charge the customer.

If a vital task fails, then the workflow will be aborted and the tasks executed should be compensated. To model this feature, the task *Compensations* in Figure 4.1 denotes the set of compensations to be executed in the case of workflow abortion. In this example, only the *Car reservation* task is non-vital (i.e., its failure does not cause the abortion of the workflow), while the other tasks are vital. For example, if the *Billing* task fails (e.g., as a result of the absence of the customer's billing information, or as a result of previous unpaid bills), then the *Flight reservation* task, the *Hotel reservation* task and the *Car reservation* task (assuming that it was executed) will be cancelled in the *Compensations* task.

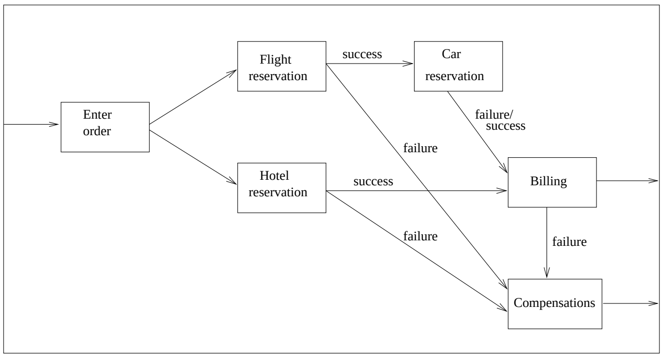

Figure 4.1: The execution structure based on semantic atomicity.

In Figure 4.1, the failure atomicity in the travel-planning workflow is based on semantic atomicity, as in sagas or flexible transactions. A problem here is that other transactions or workflows accessing the data updated by an uncommitted travel-planning workflow do not make a distinction between a preliminary reservation and a final reservation, i.e., whether the reservation is made by an active or a committed workflow. The access of dirty data may cause inconsistent results, when the number of reservations were not to include the reservations of active workflows. Also the implementation of compensations may be problematic in practice, e.g., if the cancellation of a room reservation is not free of charge.

4.2 Two-phase tasks

By setting an option a workflow instance ensures that a certain potential later update of the data item will not fail as a result of other concurrent activities. Options are implemented by two-phase tasks. A *two-phase task* (or a *2P-task* for short) is a decomposition of a task into two tasks: the first one, called the *preliminary task*, ensures that the second one, called the *complementary task*, is always executable, i.e., the complementary task will not fail as a result of other concurrent activities. The exact function of the complementary task is not determined until all the preliminary tasks of the workflow instance have been executed, i.e., until it is known whether the workflow instance will be committed or aborted. Consequently, no specific compensating mechanism to undo the effects of the aborted workflows is needed.

Each complementary task comes in two versions: a *positive complementary task* and a *negative complementary task*; only one of these is executed in a specific task instance (Figure 4.2).

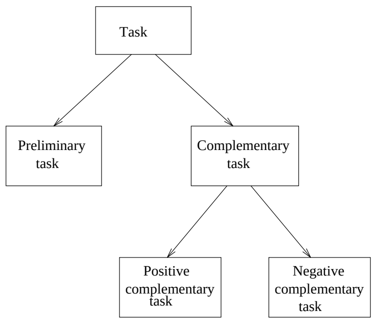

Figure 4.2: The structure of a 2P-task.

A preliminary task may succeed or fail (Figure 4.3). The success of a preliminary task ensures that its complementary task (be it positive or negative) will not fail. If the preliminary task succeeds and the workflow will be committed (i.e., all other preliminary tasks of the vital tasks have succeeded as well), then the positive complementary task will be executed. If the preliminary task fails (meaning that the requested option was not granted), then neither the positive nor the negative complementary task is executed. Otherwise, that is, when the preliminary task succeeds but the workflow has to be aborted because of a failure of some other vital task, the negative complementary task is executed.

Even if a task does not do any update we model it as a 2P-task. In such a case the task is essentially equivalent to its preliminary task, and the complementary task represents an "empty task" which is skipped over in the execution precedence graph.

As stated in Section 3.3, we use communicating tasks to model the communication required for achieving failure atomicity. Therefore we augment the structure of a 2P-task by a *participant task* (Figure 4.4). The participant task is a standard task attached to each 2P-task. The algorithm behind the participant task, as well as those behind the other communicating tasks, are

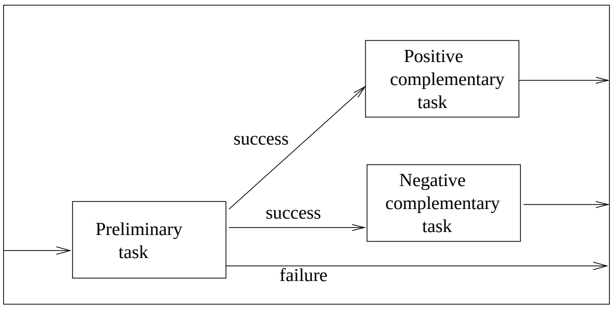

Figure 4.3: The execution graph of a 2P-task.

presented in Chapter 5.

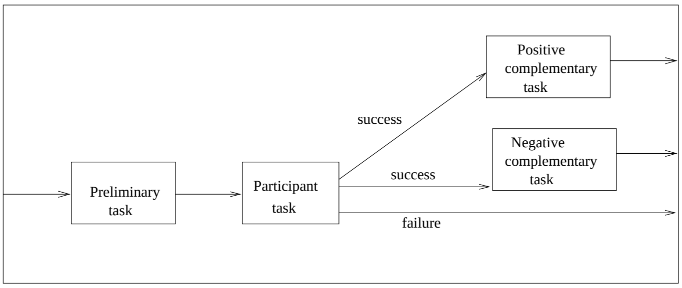

Figure 4.4: The execution graph of a 2P-task including the participant task.

4.3 Two-phase subworkflows

We now consider how the decomposition of a task into a 2P-task affects the execution precedence graph of a subworkflow. We call a subworkflow consisting of 2P-tasks a *two-phase subworkflow*, or *2P-subworkflow*, for short. In its first phase, all the preliminary tasks are executed, and in its second phase either the positive or the negative complementary tasks are executed.

We now illustrate by an example the difference between a subworkflow and a two-phase subworkflow. The task *Flight reservation* is a subworkflow of the travel-planning workflow. It consists of three tasks (Figure 4.5). First, the flight schedules are checked in the task *Check schedules*. If there are no flights fitting the customer's needs, then this task semantically fails, causing the entire subworkflow to fail. Otherwise, in the task *Own reservations* the agent checks whether he/she has allocated vacant seats for the flight. If there are vacant seats, the preliminary reservation for the customer is done. Otherwise, in the task *Borrow reservations* the agent tries to borrow vacant seats from other local branches. If this task fails, the entire subworkflow also fails.

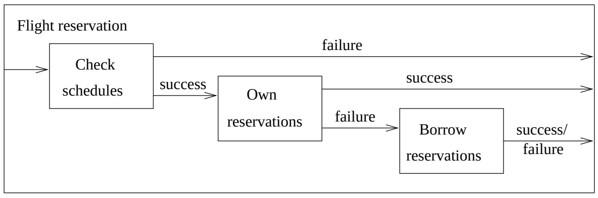

Figure 4.5: Subworkflow *Flight reservation.*

In Figure 4.6, the execution precedence graph of the two-phase subworkflow *Flight reservation* is presented. In the first phase the preliminary tasks are executed. In the second phase, first an *Inter-coordinator* task is executed and then the complementary tasks are executed.

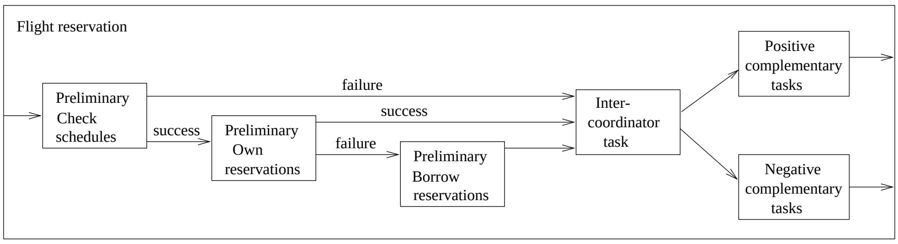

Figure 4.6: *Flight reservation* as a two-phase subworkflow.

Note that the precedence graph only illustrates the temporal ordering of the tasks: the arcs from the *Inter-coordinator* task to the *Positive complementary tasks* and the *Negative complementary tasks* only indicate a temporal dependency. Individual complementary tasks are activated through the *Participants tasks*, i.e., the *Inter-coordinator* task sends the decision to the *Participant* tasks, and then the appropriate complementary tasks are activated. However, the decision may proceed through many *Inter-coordinators* if the high of the corresponding workflow tree is more than three. So, an important aspect is that the *Participant* tasks and the *Inter-coordinator* tasks provide an equal control interface. Therefore, the parent (inter-coordinator or coordinator) does not have to know whether its children are tasks or subworkflows.

4.4 Two-phase workflows

In our approach a workflow may have a variation which can be used as a subworkflow in other workflows. This difference can be illustrated by the external control interfaces they provide (Figure 4.7): a workflow can independently commit or abort but a subworkflow cannot commit without an external request. Their execution precedence graphs differ only in that the communicating task in a workflow is the *Coordinator* task while in the subworkflow it is the *Inter-coordinator* task, i.e., they are variations differing only in their transactional properties.

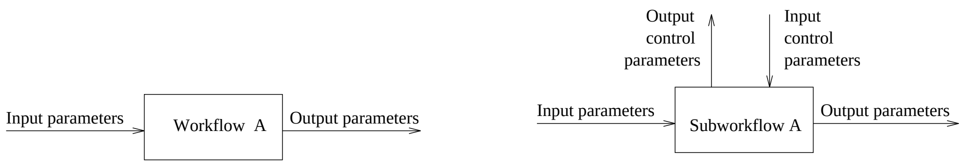

Figure 4.7: Workflow A and its subworkflow variation.

Analogous with subworkflows also workflows are divided into two phases. In the first phase, all the preliminary tasks are executed. In the second phase first it is checked whether the workflow will be committed or aborted, i.e., the Failure Atomicity Protocol (presented in Chapter 5) is executed. And then, according to the output of the protocol the positive or the negative complementary tasks are started.

In Figure 4.8, the execution precedence graph of the *Travel-planning* workflow following the two-phase decomposition is presented. The function of the *Coordinator* task is to execute the Failure Atomicity protocol.

4.5 Public options

A salient feature of *public options* is that the intermediate results of long lasting activities are presented in the database. Therefore they also have

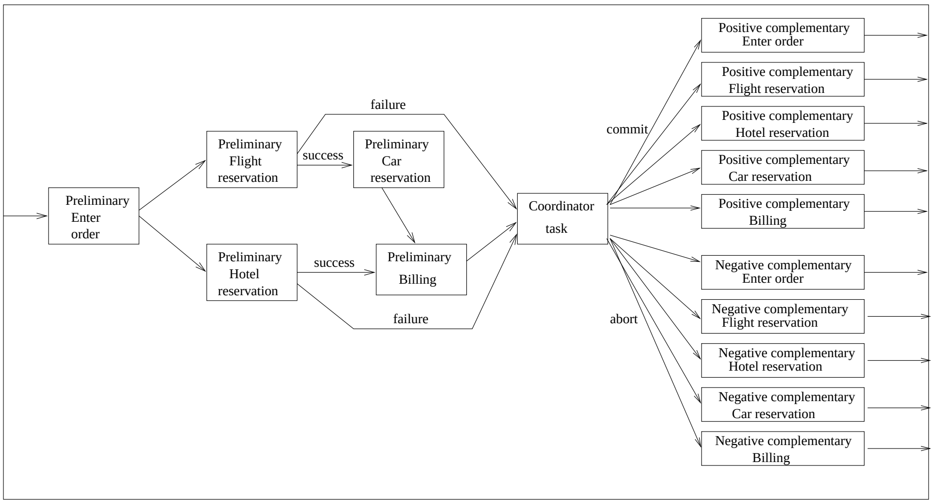

Figure 4.8: An execution precedence graph of the *Travel-planning* workflow using 2P-tasks.

specific requirements on the database schema. Typically a new relation or a new column to some relation is needed.

The specification of a public option is the following:

PUBLIC OPTION <task-name>
    Preliminary task: <transaction-call>
    Positive complementary task: <transaction-call>
    Negative complementary task: <transaction-call>

An option is a certification of the success of a potential later update. An option arises as a result of the successful execution of the preliminary task. The positive complementary task realizes the option and the negative complementary task cancels the option.

4.5.1 An example of public options

We will illustrate the use of public options by the *Travel-planning* workflow. Particularly we specify the task *Flight reservation* as a public option.

In order to illustrate what additional requirements the introduction of public option requires we first present the SQL-specifications of the original relations *Reservations* and *ReservationStates*:

```
CREATE TABLE Reservations (
    customer CHAR(20),
    flight CHAR(5),
    resSeats INTEGER);

CREATE TABLE ReservationStates (
    flight CHAR(5),
    resNum INTEGER,
    maxRes INTEGER,
    CONSTRAINT BookingWatch
    CHECK (resNum ≤ maxRes));
```

To prevent overbookings the consistency constraint *BookingWatch* is included in the specification of the relation *ReservationStates*. It states that for each tuple the value of the attribute *resNum* is at most the value of the attribute *maxRes*. Specific instances of these relations are presented in Figures 4.9 and 4.10.

Reservations	customer	flight	resSeats
	Smith	A-123	1
	Jones	C-345	3
	Taylor	B-234	1
	Cooper	A-123	2

Figure 4.9: Relation *Reservations*.

ReservationStates	flight	resNum	maxRes
	A-123	3	60
	B-234	1	50
	C-345	3	90

Figure 4.10: Relation *ReservationStates*.

In the *Flight reservation* task setting an option means to make a preliminary reservation. The option is realized by changing the preliminary reservation to a real reservation, and the option is canceled by deleting the preliminary reservation. These operations require the creation of a relation *PreliminaryReservations* and the addition of a new column *preResNum* into the relation *RelationStates*. These modifications can be done by the following SQL-statements:

```
CREATE TABLE PreliminaryReservations (
    customer CHAR(20),
    flight CHAR(5),
    preResSeats INTEGER);

ALTER TABLE ReservationStates
    ADD preResNum INTEGER;
ALTER TABLE ReservationStates
    DROP CONSTRAINT BookingWatch;
ALTER TABLE ReservationStates
    ADD CONSTRAINT NewBookingWatch
    CHECK (resNum + preResNum ≤ maxRes);
```

The constraint *NewBookingWatch* states that for each tuple of the relation *ReservationStates* the sum of the attributes *resNum* and *preResNum* is at most the value of the attribute *maxRes*. Specific instances of the relation *PreliminaryReservations* and the modified relation *ReservationStates* are presented in Figures 4.11 and 4.12.

PreliminaryReservations	customer	flight	preResSeats
	Robinson	A-123	1
	Harrison	C-345	2

Figure 4.11: Relation *PreliminaryReservations*.

Now we can define the public option *Flight reservation* as follows:

PUBLIC OPTION *FlightReservation*
Preliminary task: PreRes(customerID, flightId, resSeats)
Positive complementary task: Confirm(customerID, flightId, resSeats)
Negative complementary task: Cancel(customerID, flightId, resSeats)

The three transactions of the option are given below. Here as well as in

ReservationStates	flight	resNum	preResNum	maxRes
	A-123	3	1	60
	B-234	1	0	50
	C-345	3	2	90

Figure 4.12: Relation *ReservationStates*.

the subsequent examples we use an SQL-like notation in presenting transactions. It deviates from SQL in that we omit certain features which are irrelevant from the illustrative point of view. For example, we omit the checks of the SQLSTATE-variable.

```
PreRes(customerId, flightId, resSeats)
Begin transaction
   INSERT INTO PreliminaryReservations
         VALUES (:customerId, :flightId, :resSeats);
   UPDATE ReservationStates
         SET preResNum = preResNum + :resSeats
         WHERE flight = :flightId;
End transaction


Confirm(customerId, flightId, resSeats)
Begin transaction
    UPDATE ReservationStates
          SET preResNum = preResNum − :resSeats,
          resNum = resNum + :resSeats
          WHERE flight = :flightId;
    INSERT INTO Reservations
          VALUES (:customerId, :flightId, :resSeats);
    DELETE FROM PreliminaryReservations
          WHERE flight = :flightId AND customer = :customerId;
End transaction


Cancel(customerId, flightId, resSeats)
Begin transaction
    UPDATE ReservationStates
```

```
        SET preResNum = preResNum − :resSeats
        WHERE flight = :flightId;
    DELETE FROM PreliminaryReservations
        WHERE flight = :flightId AND customer = :customerId;
End Transaction
```

The constraint *NewBookingWatch* ensures that the transaction PreRes fails if there are not enough vacant seats on the requested flight. If PreRes does not fail then neither can the transaction Confirm nor the transaction Cancel fail semantically. For example, assume that Harrison's preliminary reservation has been successfully executed. Then the transaction Confirm would decrease the preliminary reservations of the flight C-345 by 2, and increase the reservations of the same flight also by 2. Consequently, the sum of the preliminary reservations and the reservations is not changed, and so it cannot violate the constraint *NewBookingWatch*.

4.6 Internal options

Analogous to public options *internal options* are specified as follows:

```
INTERNAL OPTION task-name
    Preliminary task: <transaction-call>
    Positive complementary task: <transaction-call>
    Negative complementary task: <transaction-call>
```

An internal option differs from a public option in that the effects of the preliminary task (i.e., the arising of the option) are not externalized. Hence, with an internal option the preliminary task updates a variable whose only use is in controlling the interleaving of workflow instances. In contrast to an internal option the preliminary task of a public option updates attributes that model the reality as any other attribute of the database.

4.6.1 Transaction coupling rules

The function of the preliminary task is to ensure that the complementary task will not fail as a result of other concurrent transactions. So, other concurrent transactions may not change the state of the database such that a consistency constraint would prevent the execution of the complementary task.

With internal options the preliminary task prevents the execution of such concurrent transactions by temporarily changing an argument of the constraint. The constraints are specific items of the database in that only the preliminary and the complementary tasks can access them. The changing of an argument in the preliminary task (transaction) effectively "tightens" the constraint, while the changing of an argument in the complementary task (transaction) effectively "loosens" the constraint, and thereby ensures semantic success of the complementary task. We call this kind of action between two transactions *transaction coupling*, and the rules specifying correct actions *transaction coupling rules*.

A problem with transaction coupling is that there may be many concurrent couplings active on the same constraint. This implies that "tightening' and "loosening' must be commutative operations. This in turn requires that the constraint to be modified must be expressed as a comparison statement. Accordingly we assume that the consistency constraint is expressed as follows:

<data-object or function><comparison-operator><numerical-value>

Typical consistency constraints can be expressed by one or more comparison statements. In terms of the relational model the constraint can concern an attribute, a tuple, a set of tuples or a whole relation. For example, the constraint *BookingWatch* presented in the previous section concerned each tuple of a relation. Using one or more comparison statements one can specify, e.g., that a certain attribute stays within a specified range, certain tuple is not deleted and that the number of tuples of a certain type is restricted. In contrast, dynamic consistency constraints [CGM88] cannot be expressed by comparison statements. They determine a set of admissible transitions between database states, e.g., successive values of a loan can only be decreasing.

The coupling of preliminary and complementary transactions is achieved by executing *increment* and *decrement operations* on the *numerical-value*, denoted by *NV* for short.

The *transaction coupling rules* are:

1. Only preliminary and complementary transactions are allowed to modify the *NV*.

2. If the comparison operator is $\leq$ or $<$ (resp. $\geq$ or $>$) then the preliminary transaction can try to decrease (resp. increase) *NV*.

3. The preliminary transaction is allowed to decrease or increase *NV* if it does not result in an inconsistent state of the database.

4. If the preliminary transaction has decreased (increased) *NV* by n, then

the corresponding complementary transaction increases (decreases) NV by n.

5. Other transactions are allowed to access the data-object as far as the access does not violate the current consistency constraint.

4.6.2 An example of internal options

Next we illustrate the use of internal options in an SQL system. As with public options we will consider the implementation of the task *Flight Reservation* of the workflow *Travel-planning*. Now the intermediate results of reservations (i.e., preliminary reservations) are not visible to the users. The users only see the relations containing the real reservations (Figure 4.13) and the reservation state of each flight (Figure 4.14). The SQL specification statements of these relations are the same than those given in Section 4.5.1.

Reservations	customer	flight	resSeats
	Melton	B-234	1
	Jones	C-345	3
	Taylor	B-234	1
	Cooper	C-345	2

Figure 4.13: Relation *Reservations*.

ReservationStates	flight	resNum	maxRes
	A-123	0	60
	B-234	120	120
	C-345	92	100

Figure 4.14: Relation *ReservationStates*.

The first tuple of the relation *ReservationStates* indicates that there are no reservations on flight A-123, the tuple is in its initial state. The second tuple indicates that flight B-234 is entirely booked. A new booking on that flight would violate the constraint *BookingWatch*, and would thereby be rejected by the SQL system. The third tuple indicates that there are 8 vacant seats on flight C-345.

For specifying the task *Flight reservation* as an internal option, we need some additional data structures. We present the required data structure by extending the relation *ReservationStates* by a new column, called *variableConstraint*. In addition to the constraint *BookingWatch* we define a constraint, called *OptionWatch*, to be used in transaction coupling. These modifications can be done by the following SQL statements:

ALTER TABLE *ReservationStates*
    ADD *variableConstraint* INTEGER;
UPDATE *ReservationStates*
    SET *variableConstraint* = *maxRes*;
ALTER TABLE *ReservationStates*
    ADD CONSTRAINT *OptionWatch*
    CHECK (resNum $\leq$ variableConstraint);

The attribute *variableConstraint* is only introduced for implementing transaction coupling. Therefore, we will define a view *ResStates* based on the relation *ReservationStates* as follows:

CREATE VIEW *ResStates* AS
    SELECT flight, resNum, maxRes
    FROM ReservationStates;

Now we can specify that the user can only access the table via the view *ResStates*, which only contains committed reservations, and so the intermediate results (i.e., internal options) are hidden from the user. Instances of the relation *ReservationStates* and the view *ResStates* are presented in Figures 4.15 and 4.16.

ReservationStates	flight	resNum	maxRes	variableConstraint
	A-123	0	60	60
	B-234	105	120	116
	C-345	92	100	100

Figure 4.15: Relation *ReservationStates*.

Now, according to the transaction coupling rules, the preliminary and complementary transactions modify the value of the attribute *variableConstraint*. For example, in Figure 4.15 we see that for flight A-123 the values of the attributes *maxRes* and *variableConstraint* are equal meaning that there is no internal option on that flight. In contrast, for flight B-234 the difference, 4, of the values of these attributes means that there are one or more

ResStates	flight	resNum	maxRes
	A-123	0	60
	B-234	105	120
	C-345	92	100

Figure 4.16: View *ResStates*.

options on that flight.

After these extensions we can define the task *Flight Reservation* as an internal option:

INTERNAL OPTION FlightReservation
Preliminary task: PreRes(customerID, flightId, resSeats)
Positive complementary task: Confirm(customerID, flightId, resSeats)
Negative complementary task: Cancel(customerID, flightId, resSeats)

The transactions of the internal option are:

```
PreRes(customer, flightId, resSeats)
Begin Transaction
   UPDATE ReservationStates
         SET variableConstraint = variableConstraint − :resSeats
         WHERE flight = :flightId;
End Transaction


Confirm(customerId, flightId, resSeats)
Begin Transaction
    UPDATE ReservationStates
          SET variableConstraint = variableConstraint + :resSeats
          SET resNum = resNum + :resSeats
          WHERE flight = :flightId;
    INSERT INTO Reservations (:customerId, :flightId, :resSeats);
End Transaction


Cancel(customerId, flightId, resSeats)
Begin Transaction
    UPDATE ReservationStates
         SET variableConstraint = variableConstraint + :resSeats
```

```
        WHERE flight = :flightId;
End Transaction
```

Now we consider the behavior of several simultaneous options on a flight. Assume that there are 92 reservations and no options on flight C-345. So there is a tuple

(C-345, 92, 100, 100)

in the relation *ReservationStates*. Then a preliminary task requesting an option concerning three reservations can commit; therefore, the option is granted, and the value of the tuple will be

(C-345, 92, 100, 97).

The SQL system accepts such an update as it violates neither the constraint *OptionWatch* nor the constraint *BookingWatch*. Now assume that another option concerning two reservations on the same flight is started. The SQL system will accept the request, and the new value of the tuple will be

(C-345, 92, 100, 95).

After this, the positive complementary task of the first option is executed, changing the tuple to

(C-345, 95, 100, 98).

The transaction coupling rules ensure that the complementary task cannot be rejected as a result of violating the constraint *OptionWatch*. In particular, the complementary task first updates the value of *variableConstraint*, which ensures that the update of *resNum* cannot violate the constraints. Then assume that a normal transaction (i.e., outside the control of the workflow management system) reserving three seats is started, resulting in the tuple

(C-345, 98, 100, 98).

Now, only the active option for two reservations can make reservations, as its complementary transaction is allowed to increase the attribute *variableConstraint* by 2.

Recall that an option is a certification of the success of a potential later update. If there are no constraint which could prevent the later update neither the option is needed. For example, this is the case with the tasks which revoke reservations.

4.7 General treatment of options

So far we have discussed options from an illustrative point of view. We next work out the concept of database consistency so as to be more suitable for analyzing the correctness of options. In particular, the notion of an argument of a consistency constraint is introduced. Then by considering the way consistency constraints are expressed we then make a distinction between explicit and implicit consistency constraints. And, finally, we give an interpretation of an option using the notion of a constraint argument.

4.7.1 Database consistency

The consistent states of a database are specified by consistency constraints: a database state is consistent if all the consistency constraints are met [Pap86]. Although this definition is intuitive and clear, it is not appropriate for our further analysis. Before giving an equivalent definition, we present some notations.

Let S be a database state and let CC be a consistency constraint. A *constraint argument* of CC on S is each minimal subset of S on which the validity of CC can be evaluated. For example, the attributes *resNum* and *maxRes* of each tuple of the relation *ReservationStates* (Figure 4.10) is a constraint argument of the constraint *BookingWatch*.

Based on this notion of a constraint argument we can give another, yet equivalent, definition of database consistency: a database state S is consistent with respect to CC if each constraint argument of CC in S is consistent with respect to CC.

In analyzing the effects of an option we can now restrict ourselves on the constraint arguments instead of considering the whole database and all its consistency constraints.

4.7.2 Explicit and implicit consistency constraints

We make a distinction between explicit consistency constraints and implicit consistency constraints depending on whether the constraint is expressed by data items or by metadata items.

An *explicit consistency constraint* is one that is stated in the metadata using a specific language, e.g., by a key constraint, a referential integrity constraint, a CHECK statement or an assertion in SQL.

We say that a data item is an *implicit consistency constraint* if

- it is a constraint argument of an explicit consistency constraint, and

- it models the state of certain other data items of the database.

For example each tuple of the relation *ReservationStates* (Figure 4.10) is an implicit consistency constraint: each such tuple is a constraint argument of the explicit consistency constraint *BookingWatch*, and each tuple specifies the state of certain tuples in the relation *Reservations* (Figure 4.9). For example, the first tuple states that there are three reservations on the flight A-123 and that there may be at most 60 reservations on that flight.

One can replace implicit consistency constraints by explicit consistency constraints. If an implicit constraint is set upon more than one relation (e.g., the tuples of the relation in Figure 4.12 are set upon the relations *Reservations* and *PreliminaryReservations*) then it must be expressed by an assertion, while a constraint upon one relation can be expressed by a CHECK-statement. For example, the previously stated implicit consistency constraint (i.e., the tuple (A-123, 3, 60)) could be replaced by inserting the following statement into the specification of the relation *Reservations*:

```
CONSTRAINT BookingWatchOnFlightA-123
CHECK (60 ≥
      (SELECT SUM(resSeats)
      FROM Reservations
      WHERE flight = 'A-123'));
```

Explicit and implicit consistency constraints have their strenghts and weaknesses. Implicit constraints ease the modifications of constraints as instead of modifying metadata only tuples have to be accessed. On the other hand, explicit constraints ease database updates. For example, making a reservation only involves inserting a tuple to the relation *Reservations*, and the relation *ReservationStates* would not be needed at all.

4.7.3 Managing options by modifying the allowable states of constraint arguments

Each option, be it public or internal, modifies the state of the database in a way that temporarily restricts the allowable states of the database, and thereby ensures that the option can be realized if needed. The restriction of allowable states is done by modifying either explicit or implicit consistency constraints.

To illustrate this we denote by CC a consistency constraint and by A the set of its constraint arguments, i.e., a subset of a database. Further, we denote by $S(A)$ the set of all possible states of A, by $C(A)$ the set of all consistent states in $S(A)$, and by $T(A)$ the set of temporarily allowable states of A. Thus $T(A)$ varies according to the active options. In order to illustrate this dynamic property, *T(A)* is denoted by dotted lines in Figure 4.17. If there are no active options, then *C(A)* and *T(A)* equal. Otherwise, since setting an option always tightens a constraint, *T(A)* is a proper subset of *C(A)*.

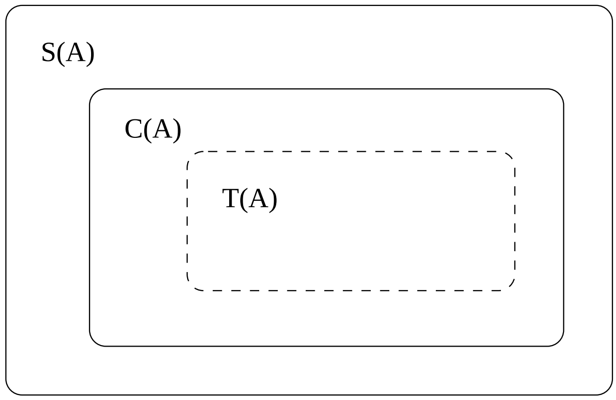

Figure 4.17: Relationships of the states of a constraint argument A.

Assume that the workflow system receives an option request concerning a set of constraint arguments A. The system is allowed to accept the request if the current state of A is included in *T'(A)* which is obtained from *T(A)* by reducing it by the requested option. If the request is accepted, then *T'(A)* is assigned to *T(A)*. So, accepting an option reduces *T(A)*, and cancelling an option enlarges *T(A)*. In realizing the option, first *T(A)* is enlarged ensuring that the following update will not move the state of A outside *T(A)*.

4.8 Discussion

Failure atomicity and high performance are contradicting goals. In conventional (syntactic) transaction models failure atomicity is guaranteed by forcing transactions to hold their data locked until the whole activity either commits or aborts, thereby decreasing the performance of the system. With semantic atomicity models (e.g., with sagas and flexible transactions) transactions may release their data resources before the whole activity (workflow or advanced transaction) is committed or aborted, thus leading to better performance but in the same time exposing dirty data to other concurrent transactions.

Within this context, options represent an intermediate form: transactions (or tasks) hold their data, but only in a liberal way allowing other transactions to access the data as long as it does not jeopardize failure atomicity.

Logically, internal options and transaction coupling have similarities with *escrow locking* [O'N86]. The idea behind escrow locking is to preserve the truth of a predicate on a data item between the time the transaction first reads the data item and the time the predicate is evaluated at committing the transaction. Escrow locking was originally designed to deal with arithmetic operations on hotspot data. Later on with the contract model [Wäc91, WR92, Sch93, RS95a, Sch94, SR96] escrow locking was proposed to be used in ensuring execution atomicity and the executability of the compensating transactions. Our approach differs from escrow locking in three aspects. First, instead of predicates we evaluate only consistency constraints (predicates) that can be expressed using SQL. Second, with options the truth of the consistency constraint is enforced during the time between two separate transactions — not a time interval within the execution of a single transaction. Third, options are stored in the database (stable storage) while with escrow locking predicates are stored in volatile storage.

Chapter 5

Failure Atomicity of Two-Phase Tasks

In Chapter 4 we attached a communicating task to each workflow, subworkflow and task. The function of a communicating task is to terminate a workflow instance in a coordinated way. In terms of options this means that either all the accepted options will be realized or all are canceled.

In this chapter we present the algorithms of the communicating tasks. These algorithms comprise the failure atomicity protocol. Before presenting the protocol we will consider the communication topology of the protocol.

5.1 Communication topology

A communication topology [BHG87] specifies who sends messages to whom. The communication topology of our protocol is derived from the structure of a workflow instance tree. For example, the topology might be the one presented in Figure 3.9.

Each communicating task receives as input parameters the names of its parent (if any) and children (if any). So the *Coordinator* task in the tree knows the names of its children, but does not know whether they are *Inter-coordinator* tasks or *Participant* tasks. Respectively each *Inter-coordinator* task knows its children and parent but does not know whether its parent is an other *Inter-coordinator* task or the *Coordinator* task. Consequently no task knows the entire structure of the tree.

5.2 The Failure Atomicity Protocol

Analogous to the two-phase commit protocol (2PC-protocol) [BHG87] the communication in the protocol is comprised of two phases: a voting phase and a decision phase.

1. First, in the voting phase the coordinator sends a request message to its children. Those children that are inter-coordinators forward the message to their children. Eventually all the participants will receive the request message. The participants vote by sending either the *failure* or *success* message to their parents. Each inter-coordinator votes *success* if the corresponding subworkflow has reached its objective, i.e., all its vital children (vital tasks and vital subworkflows) have voted *success*.

2. The decision phase starts when the coordinator has received the responses from all its vital children. If all the vital children have voted *success*, the coordinator decides to commit, and otherwise to abort. Then the coordinator sends the decision to its children. Further, those children that are inter-coordinators forward the decision to their children. So each participant task will receive the decision, and then, in the case of a commit message, the participants start the positive complementary task. In the case of an abort message they start the negative complementary task.

In practice the protocol should also tolerate system and communication failures. Therefore we will require that, if all existing failures are repaired and no new failures occur for a sufficiently long time period, then the protocol will eventually be executed.

There are two reasons why the simple protocol described above does not tolerate communication and system failures:

- First, at various points of the protocol either the coordinator, inter-coordinators or the participants must wait for messages before proceeding, but messages may not arrive due to a failure giving rise that those may be waiting forever. For this reason we have to add timeout actions to the protocol.

- Second, when the coordinator, an inter-coordinator or a participant recovers from a failure it has to reach the decision. This requires that the coordinator, inter-coordinators and the participants have to keep some information in the log (stable storage).

The coordinator's, inter-coordinator's and participant's algorithms are presented below.

Coordinator task:
send a request message to all the children;
write the *start-record* in the log;
wait for response messages from the children;
if all the messages from the vital children have been received **or** a timeout has expired **then**
begin
    if all the vital children voted *success* **then**
    begin
        write the "commit" decision to the log;
        send the commit message to all the children
    end else
    begin
        write the "abort" decision to the log;
        send the abort message to all the children
    end;
end
else write the decision message "no response received" to the log;
return

Inter-coordinator task:
wait for a request message from the parent;
send a request message to all the children;
wait for response messages from the children;
if all the messages from the vital children have been received **or** a timeout has expired **then**
begin
    if all the vital children voted *success* **then**
    begin
        write the "commit" decision to the log;
        send the commit message to the parent
    end else
    if some vital child voted *failure* **then**
    begin
        write the "abort" decision to the log;
        send the abort message to the parent
    end;

```
    wait for a decision message from the parent
        on timeout initiate the termination protocol;
    on receiving the decision message
    begin
        write the decision record to the log;
        send the decision message to all the children
    end;
end;
return
```

```
Participant:
wait for a request message from the parent;
if the preliminary task did not fail then
begin
    send a response message "success" to the parent;
    wait for a decision message from the parent
        on timeout initiate the termination protocol;
    on receiving a decision message
    begin
        if the decision is "commit"
        then execute the positive complementary task
        else execute the negative complementary task;
        write "complementary executed" to the log
    end;
end
else send a response message "failure" to the parent;
return
```

The period between the moment a participant or inter-coordinator sent a positive response message and the moment it received the decision message is the *uncertainty period* for that participant or inter-coordinator. During that period the participant and the inter-coordinator do not know how to proceed. If a participant or an inter-coordinator in its uncertainty period is interrupted by a timeout, then it must consult its parent to find out the decision. This consultation is carried out by the following *termination protocol*.

Initiator's algorithm:
start: **send** a decision request to the parent;

wait for a decision message
on timeout goto start;
on receiving a decision message **return**;

Parent's algorithm:
wait for a decision request from any child;
if the decision is not known
then initiate the termination protocol (initiator's algorithm);
send the decision message to the requesting child;
return

5.3 Extending the Failure Atomicity Protocol

So far we have assumed that each task is implemented as a 2P-task. Now we extend the failure atomicity protocol such that it also allows other approaches for implementing tasks. Particularly a task may be:

- an atomic task (ACID-transaction),
- a semantically atomic task, or
- a manual or partially automated task.

By an atomic task we mean a task that possesses the ACID-properties, i.e., a traditional transaction. In a semantically atomic task the atomicity is enforced by compensation. We will also incorporate manual and partially automated tasks as there may be tasks whose complete automation is neither possible nor meaningful.

Through these approaches we do not gain the advantages of 2P-tasks (options) but rather we make the protocol more flexible as there are many ways to implement each task, and thus increase its usability. For example, if each 2P-task would be implemented as an atomic transaction and there are no subworkflows, then the protocol would work similar to the 2PC-protocol.

A feature of the failure atomicity protocol is that the way a task is implemented is hidden from the user process that communicates with the participant task. As illustrated in Figure 5.1, the coordinator (or inter-coordinator) sees only the interface provided by the participant.

So, the other approaches for implementing tasks require only modifications on the participant's algorithm. The modifications required by atomic transactions are considered in Subsection 5.3.1, modifications required by

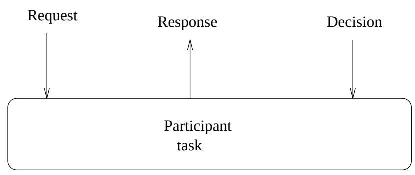

Figure 5.1: The interface of the participant.

semantically atomic tasks are considered in Subsection 5.3.2, and then, in Subsection 5.3.3, the connection of manual and partially automated tasks are discussed.

5.3.1 Executing a task as an atomic transaction

The states of transactions are essential in executing a task as an atomic transaction. From a user's point of view a transaction is either in an active, aborted or committed state (Figure 5.2). An active transaction is one that

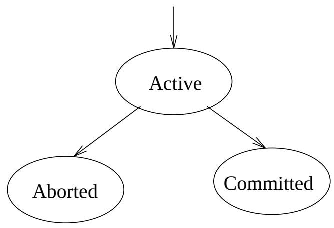

Figure 5.2: The states of a transaction from the user's point of view.

has not yet committed or aborted. From the system's point of view, an active transaction is either in a running, terminated or prepared state (Figure 5.3). A terminated transaction is one whose execution is terminated. The correctness of the execution is not necessarily checked and the updates made by a terminated transaction may be lost in the event of a system failure.

A prepared transaction is one that can be committed

- from the atomicity point of view (i.e., all of its updates are logged in the stable storage), and
- from the recovery point of view (i.e., it has not read dirty data).

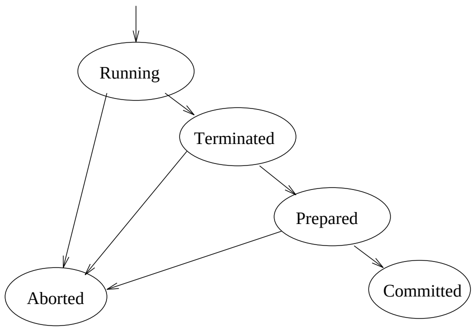

Figure 5.3: The states of a transaction from the system's point of view.

In distributed systems which provide atomic distributed transactions each local system has to support the prepared state as an external state. According to the 2PC-protocol [BHG87] the coordinator decides to commit the subtransactions if it knows each subtransaction of the distributed transaction to be in the prepared state, and otherwise it decides to abort the subtransactions.

Through the prepared state we will also tie atomic tasks (transactions) into the protocol. Semantically the prepared state corresponds to the successful preliminary task as it ensures that the subtransactions can be aborted or committed according to the coordinator's decision. Further, semantically aborting a subtransaction corresponds to the execution of the negative complementary task, and committing a subtransaction corresponds to the execution of the positive complementary task.

We can divide the execution of an atomic subtransaction into a prepare task, a commit task and an abort task and connect these to the participant task as presented in Figure 5.4. Now if the prepare task does not fail, then the participant sends the success message to its parent, otherwise it sends the failure message. When it receives the decision message, it commits or aborts the transaction according to the message. The modified participant algorithm is presented below.

Participant:
wait for a request message from the parent;
if the prepare task did not fail **then**
begin
    send a response message (success) to the parent;
    wait for a decision message from the parent

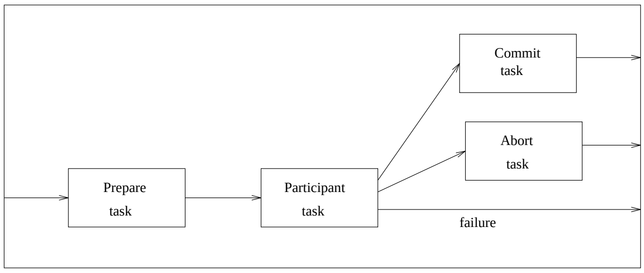

Figure 5.4: Task execution graph.

on timeout initiate the termination protocol;
on receiving a decision message
begin
if the decision is "commit" **then**
execute the commit task **else**
execute the abort task;
write "complementary executed" to the log
end;
end
else **send** a response message (failure) to the parent;
return

A problem with this solution is that the transaction has to hold data resources (e.g., locks) from the running state until it reaches either the committed state or the aborted state. Hence this kind of implementation is not suitable as a general method for long lasting activities. However, the 2PC-protocol has been found necessary in workflow environments, and has been implemented in a CORBA-based workflow architecture [MSKW96]. In FlowMark, the 2PC-protocol is used for replication, which in turn increases system availability [AAEAM97].

5.3.2 Executing a task as a semantically atomic transaction

Replacing a 2P-task by a semantically atomic transaction and its compensating transaction would require the following modification to the participant algorithm. First, instead of executing a preliminary task, the atomic transaction is executed. Then, if the execution is successful the participant sends the success message to its parent; otherwise it sends the failure message to its parent.

Finally, if the participant receives the commit decision, then no transaction has to be executed, but if the message was an abort decision then the compensating transaction is executed. By connecting these actions to the participant task we get the execution graph presented in Figure 5.5.

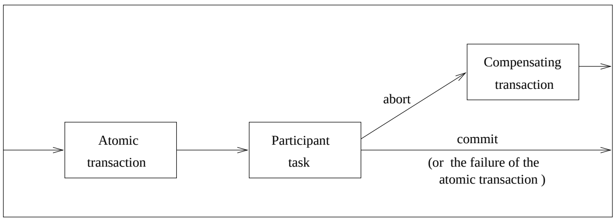

Figure 5.5: Task execution graph.

The modified participant algorithm is the following.

Participant:
wait for a request message from the parent;
if the atomic transaction did not fail **then**
begin
    send a response message (success) to the parent
        wait for a decision message from the parent
        on timeout initiate the termination protocol;
    on receiving a decision message
    begin
        write the decision to the log;
        if the message was abort **then**
        execute the compensating transaction
    end;
end
else **send** a response message (failure) to the parent;
return

As we have already stated, the main problem with using compensating transactions is that the function of the preliminary transaction is carried out by a transaction having different semantics. Note that making a reservation for a flight and making a preliminary reservation are not semantically equivalent. In practice, the seriousness of this problem depends on the semantics of the other transactions in the system.

5.3.3 Executing partially automated and manual 2P-tasks

The notion of a workflow was introduced to facilitate business process reengineering and automation. However, it is neither meaningful nor possible to automate all tasks. Therefore, the wide usability of a workflow system requires that it should also provide support for partially automated and manual tasks.

We now consider how the interaction between the failure atomicity protocol and human users performing partially automated or manual tasks can be achieved. Our idea is to replace the preliminary task by a request to a human user. The user is asked whether she or he commits to execute the positive or negative complementary task in response to a forthcoming decision. If the user responses yes, then the preliminary task is considered as successful, and otherwise as failed. Then the participant task passes this information to its parent. Finally, after receiving the decision the participant task requests the user to proceed according to the decision, i.e., to execute the complementary task. If the user leaves the complementary task undone, the workflow may commit or abort, though it is not finished correctly. However, the positive response is stored in the log, and thereby the failure can be traced.

The participant protocol incorporating the modifications required by partially automated or manual tasks is presented below.

```
Participant:
wait for a request message from the parent;
if  the prepare task did not fail then
begin
      send a response message (success) to the parent;
      wait for a decision message from the parent
            on timeout initiate the termination protocol;
      on receiving a decision message
      begin
            pass the message (commit or abort) to the human;
            write "complementary requested" to the log
      end;
end
else  send a response message (failure) to the parent;
return
```

As an example of a manual or partially automated task, let us reconsider the *Loan request* workflow presented in Chapter 3. Its task *Enter loan decision* (Figure 3.6) typically requires human intervention, and thus should be executed as a manual or partially automated task. There may also be variations of this task with respect to the automation level, e.g., with minor loans the totally automated variation is executed and in other cases partially automated variation is executed.

5.4 Discussion

Failure atomicity of long lasting activities can be enforced by 2P-tasks, atomic transactions, or by semantically atomic transactions. Each of these approaches has its strengths and weaknesses. Which approach is appropriate for a given task is task-specific. Moreover, some tasks may require manual execution.

The failure atomicity protocol allows each task to be implemented as a 2P-task, an atomic task, or a semantically atomic task. Further a task may be manual or partially automated. In this sense, the range of applicability of the protocol is wide.

In addition to failure atomicity the workflow system should provide support for execution atomicity. These two requirements are however partially intertwined: for example with options the preliminary transaction prevents interleavings that would cause the complementary transaction to fail. Particularly, in Chapter 7 we will extend the failure atomicity protocol to provide support for execution atomicity.

Chapter 6

Negotiation workflows

Advanced transaction models [ELLR90, GMGK+91, RELL90] as well as transactional workflow models (e.g., [WR92, BDS+93, GH94]) have been introduced in order to achieve the correctness and reliability of workflows. Also cooperative transaction models (e.g., [PH88, KP92, NRZ92, HOS90, NG92]) have been developed for supporting cooperative work in a centralized environment. These models allow users to cooperate only by successive tasks, i.e., a workflow defines the order of task execution or conditions under which tasks must be executed, but negotiation-like cooperation cannot be modelled. On the other hand, with groupware systems [EGR91] users (humans) can cooperate spontaneously without predefined protocols. Such cooperation is necessary for many groupware applications, and integrating such features also for workflow systems would increase their usability.

6.1 Cooperative workflows

In many distributed applications the users of autonomous local systems have to cooperate. Typical cooperation is based on a predefined practice, but such cooperation cannot be automated if there are neither a model nor methods to carry out the cooperation. For example, the business processes where dealing is made by first requesting the offers and then making the deal are very common in practice. Our goal then is to develop appropriate concepts and methods to automate such cooperative processes.

The corner-stone for our analysis is a protocol, called the *2-phase negotiation protocol*, or the *2PN-protocol* for short. The 2PN-protocol is similar to the failure atomicity protocol of the previous chapter in that it consists of a vote phase and a decision phase, but the protocols differ in that the spectrum of the negotiation in the 2PN-protocol is not restricted to atomic commitment. In addition the 2PN-protocol is always two-level, i.e., there are no inter-coordinators.

Essentially, the 2PN-protocol synchronizes the execution of the tasks of a distributed workflow. Further, in order to avoid the problems related to long-lasting transactions, we have extended the notion of an option to a *partially realizable option*, that in turn can be implemented by a variation of a 2P-task, called *multivalued 2P-task*.

6.2 Enforcing execution dependencies

Our view of cooperative workflows is twofold: we concentrate on the modular workflow structure as well as the structure of negotiation workflows. Further, the intertwinement of these two structures is our main topic. Ideally,

- the semantics of negotiation is included in workflow modules (tasks),
- while negotiations are carried out by the 2PN-protocol, and
- the updates resulting from negotiation are executed by the multivalued 2P-tasks.

In negotiation workflows, execution dependencies and data flow are carried out by the 2PN-protocol. A negotiation workflow may also be a subworkflow of another workflow (as will be illustrated in the following example), and so the entire negotiation workflow may be activated as any task or subworkflow of the workflow. For example, a task or a subworkflow may not begin before a particular previously started task commits. Such constraints are usually specified by triggers [DHL90, DHL91], Petri nets [Jen91] or by finite state automata [ASSR93].

6.2.1 A negotiation example

As we have stated, the travel-planning example [Gra81] was originally presented to show that the subtransactions of a transaction may have different kinds of transactional requirements, e.g., some subtransactions may be undone or redone while other subtransactions may be deferred but not undone. Later on, alternative subtransactions [ELLR90] and contingency transactions [BÖH+92] have been motivated by the same example. Again, we motivate the need of negotiation workflows by the example.

Here we are not interested in the precedence graph of the tasks in a workflow but rather of the subworkflows, which could be implemented as negotiation workflows. So we are interested in the following tasks:

Flight reservation: An agent negotiates with airlines to order a ticket for a flight.

Hotel reservation: An agent negotiates with hotels to reserve a room.

Car reservation: An agent negotiates with car rental companies for car reservation.

For simplicity we assume that there are only two airlines, say Finnair and SAS, and two hotels, say Sheraton and Hilton, to be consulted. The travel agent can then order a ticket either from Finnair or from SAS, and respectively reserve a room either from Sheraton or from Hilton. Hence, using alternative subtransactions one can specify that if the subtransaction reserving a ticket from Finnair fails, then the subtransaction which tries to make the reservation from SAS is executed, or vice versa.

Now assume that a customer may not prefer any of the airlines or hotels, but rather wants to make the decision (reservation) based on an explicit choice criterion (e.g., choose the cheapest one). Such a functionality can only be transactionally processed by a negotiation workflow.

6.3 The negotiation model

This section presents the structure of a negotiation workflow and the 2PN-protocol, and particularly, how these are linked together.

6.3.1 The structure of a negotiation workflow

A *negotiation workflow* is comprised of one *coordination task* and one or more *participant tasks* (Figure 6.1). These tasks are located at the *task*

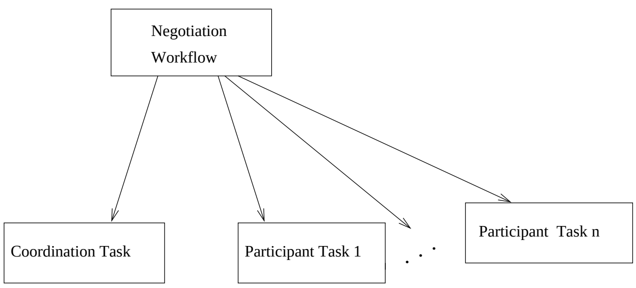

Figure 6.1: The tasks of a negotiation workflow.

pools of different sites.

The coordination task negotiates with the participant tasks to make a deal. For example, the subworkflows *Flight reservation*, *Hotel reservation* and *Car reservation* can be modelled as negotiation workflows, and are therefore denoted by double outlines in Figure 6.2. The home site of these negotiation workflows is the site of their coordination task, i.e., the site of the travel agent.

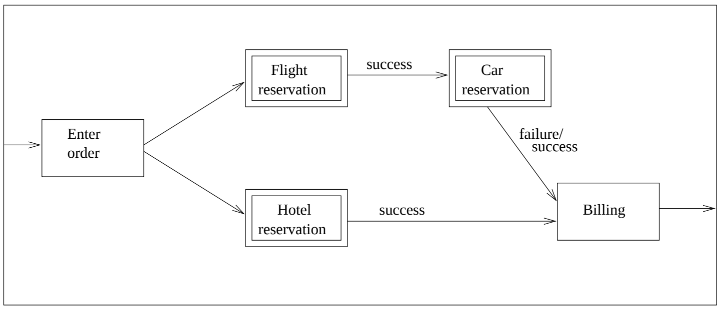

Figure 6.2: Negotiation workflows as subworkflows in the workflow *Travel-planning*.

From a transactional point of view a negotiation workflow is comprised of one coordination transaction and one or more multivalued 2P-tasks (Figure 6.3). The *coordination transaction* is activated by the coordinator in the

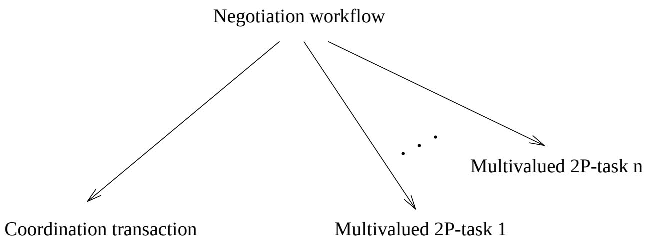

Figure 6.3: The components of a negotiation workflow.

2PN-protocol after the decision of the negotiation is done. It is a short-lasting ACID transaction that stores the result of the negotiation in the database. The components of the multivalued 2P-tasks (i.e., a preliminary task and one or more complementary tasks) are activated by the participant in the 2PN-protocol. The coordinator and the participants communicate through the messages of the 2PN-protocol. Such communication is denoted by the broken lines in Figure 6.4.

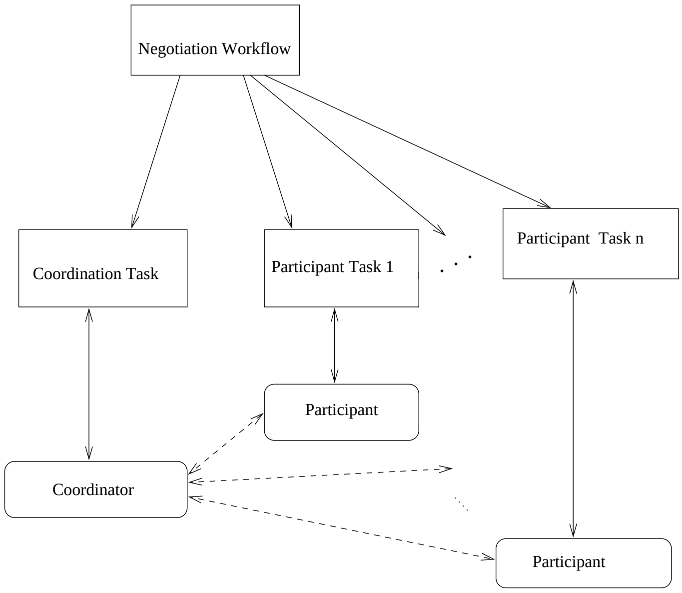

Figure 6.4: The interactions of the negotiation workflow and the negotiation transaction.

6.3.2 Multivalued 2P-tasks

A *multivalued 2P-task* differs from a 2P-task in that the number of complementary tasks is not restricted (Figure 6.5). As a 2P-task, a multivalued 2P-task consists of two phases: in the first phase the preliminary task is executed, and in the second phase one of the complementary tasks is executed. However, in many cases, only two complementary tasks are needed in the specification of a multivalued 2P-task. In such a case:

- one complementary task compensates the effects of the preliminary task (unsuccessful negotiation), while
- the other completes the preliminary task (successful negotiation).

For example, assuming that the travel agent negotiates with Finnair and SAS to order a ticket, and the agent accepts the offer from SAS and refuses

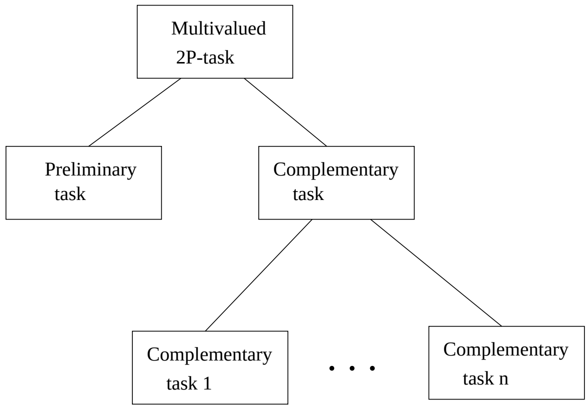

Figure 6.5: The components of a multivalued 2P-task.

that from Finnair. As a result, at the site of SAS a complementary task which completes the preliminary task is executed, and at the site of Finnair a complementary task which compensates the effects of the preliminary task is executed. As a result, the execution structure of Figure 6.6 will be generated. The root of the tree presents the execution of a negotiation workflow. The leaves of the tree are executed as local transactions, and in particular, their execution dependencies are enforced by the 2PN-protocol to be described in the next section.

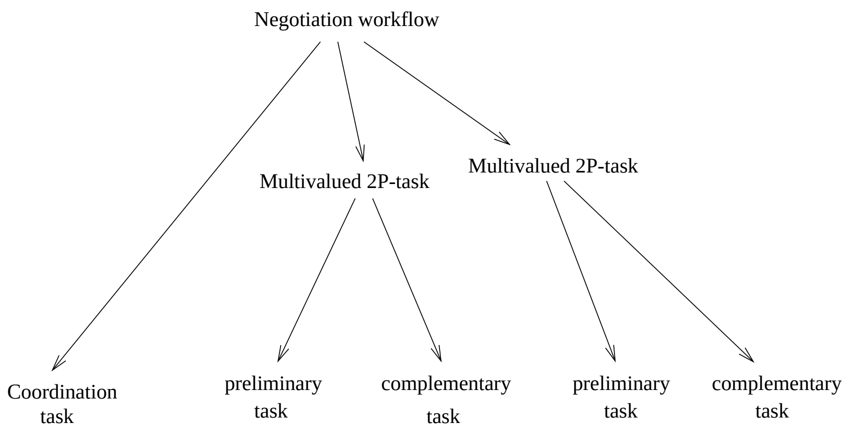

Figure 6.6: The execution structure of a negotiation transaction.

If the participant task is not willing to take part in the negotiation requested by the coordination task, then none of the multivalued 2P-tasks is executed, e.g., if an airline is requested for a seat, but all seats are already booked. In our framework this means that the participant task does not need the services of the multivalued 2P-task.

6.3.3 The 2PN-protocol

The 2PN-protocol has two functions. First, it coordinates the negotiation, and second, it ensures reliability in the presence of failures. For clarity we first describe the protocol assuming that there are no failures and then we present the exact protocol, which is tolerant for system and communication failures.

The coordinator knows the names of the participants, and so it can send them messages. The participants know the name of the coordinator, but they do not know each other. Assuming that there are no failures, the 2PN-protocol works as follows:

1. The coordination task activates the coordinator, and thus starts the 2PN-protocol, by passing a request message to it. Then the coordinator sends a request message to all the participants.

2. When a participant receives a request message, it first activates the participant task by passing the message to it. After processing the message the task passes a response message to the participant. If the participant task accepted the request, then the participant starts the preliminary transaction, and if it commits then the participant sends a response message to the coordinator. Otherwise it informs the participant task about the abort of the preliminary task.

3. The coordinator collects the response messages and passes the messages to the coordination task. After their processing, the coordination task passes a decision message to the coordinator, which sends participant-specific decision messages to the participants. (For illustrative purposes we say that a decision message is negative, if the coordinator did not accept the response, and otherwise it is positive.)

4. Each participant that accepted the request waits for a decision message from the coordinator. After receiving the decision message the participant passes it to the participant task, which after processing it returns the identifier of the complementary task to be executed, and then the participant starts the complementary task.

In Figure 6.7 we have illustrated by a state diagram the synchronization of the protocol messages and the execution of the preliminary and the complementary tasks. The diagram represents the states of the participant in the case of two complementary tasks (a positive and a negative complementary task). With respect to the negotiation of a ticket for a flight we can give the following interpretation for the diagram.

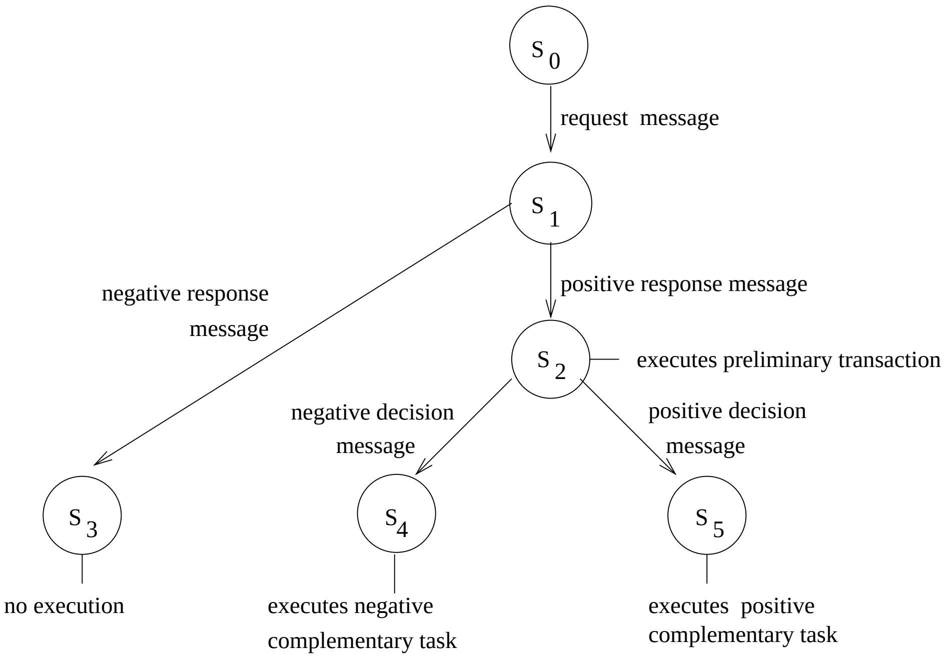

Figure 6.7: The states of a participant.

First, the coordinator sends a *flight-id/seats*-request, which identifies the flight and the number of requested seats. Then the participant executes the preliminary reservation (preliminary task) and sends the *flight-id-price/seats*-response (positive response) or *flight-id/not-available*-response (negative response) to the coordinator. Finally, the coordinator sends participant-specific *flight-id/passengers*-message(s) (positive decision message) to the participant(s) offering the flights with the lowest price. To the other participants it sends *flight-id/nodeal* messages (negative decision message).

The protocol described above does not tolerate communication and system failures. The following 2PN-protocol tolerates both failures.

```
Coordinator:
send a request message to all the participants;
write the start-2PN record to the log;
wait for response messages from the participants;
if all the messages from the participants have been received or
a timeout has expired then
begin
      pass the response messages to the coordination task;
      wait for a decision messages from the coordination task;
      write the decision-record to the log;
      execute the coordination transaction;
      send decision messages to participants
end
else write the decision message “no response message received”
to the log;
return
```

```
Participant:
wait for a request message from the coordinator;
on receiving a request message activate the participant task by
passing the request message to it;
wait for a decision messages from the participant task;
write the response-record to the log;
if  the response message is positive then
begin
          execute the preliminary task;
          if the preliminary transaction committed then
          begin
             send a response message (positive message) to the coordinator;
             wait for a decision message from the coordinator
                        on timeout initiate the termination protocol;
             write the decision-record to the log;
             pass the decision messages to the participant task;
             wait for a decision messages from the participant task;
             execute a complementary task
          end
          else activate the participant task by passing the abort message to it
end
else  send a response message (negative message) to the coordinator;
return
```

The *termination protocol* of the 2PN-protocol is presented below

Initiator's algorithm:
start: **send** a decision request to the coordinator;
  wait for a decision message
    on timeout goto start;
  on receiving a decision message **return**

Coordinator's algorithm:
wait for a decision request from any participant;
send a decision message to the participant;
return

As we have assumed that the coordination task, preliminary tasks and complementary tasks are executed as local transactions in the local systems providing the ACID-properties [BHG87], we do not have to take care of their reliability.

6.3.4 Implementing multivalued 2P-tasks

A multivalued option differs from other options in that it may be partially realized. This feature can be implemented in such a way that, instead of executing only one complementary task (the positive or the negative one), several complementary tasks (positive, negative, or both) can be executed. Partial realization has not to be taken into account in a preliminary task, i.e., the preliminary task of a 2P-task and that of a multivalued 2P-task have an equal functionality.

As an example of a multivalued 2P-task assume that a preliminary task does five preliminary reservations; its complementary task confirms two reservations and cancels three preliminary reservations. To illustrate how this can be implemented using options lets us consider the *FlightReservation* option discussed in Chapter 4:

PUBLIC (INTERNAL) OPTION FlightReservation
  Preliminary task: PreRes(customerID, flightId, resSeats)
  Positive complementary task: Confirm(customerID, flightId, resSeats)
  Negative complementary task: Cancel(customerID, flightId, resSeats)

Now the participant in the 2PN-protocol activates the positive complementary task by giving the value 2 to the parameter resSeats, and activates the negative complementary task by giving the value 3 to the parameter resSeats.

6.4 Discussion

Our perspective on negotiation workflows is focused on their functionality rather than on their formal specification, which could be given for example by the ACTA-framework [CR90, CR92, CR93].

The main aim was to automate negotiation-like tasks which so far have required human intervention. Further, the 2PN-protocol ensures that negotiation transactions are atomic. However we did not considered their atomicity from the whole workflow's point of view. For example, what happens if the negotiation workflow *Flight reservation* (a subworkflow) were executed successfully and then the whole workflow *Travel-planning* fails. This problem, however, has a simple solution, namely executing negotiation workflows as two-phase subworkflows as discussed in Section 4.3. That is, the complementary task of a negotiation workflow is not executed until it is known whether the whole workflow will commit or abort.

And, finally, as the starting motivation for workflow systems was a need to automate office paper processing, we believe that this progress will continue by automating still more and complex tasks.

Chapter 7

Self-isolating workflows

Serializability as the correctness criterion means that for any set of consistency constraints imposed on the database the execution preserves database consistency. Syntactic and semantic information about an application makes it possible to weaken the correctness criterion and allows nonserializable executions which are correct from the point of view of an application.

Using semantic information we can allow workflow instances to access data without completely isolating the accessed data (e.g., by a write-lock) from other concurrent transactions and workflows. In workflow concurrency control sufficient isolation requirements include conditions stating that:

- a certain attribute stays within a specified range,
- a certain tuple is not deleted, and
- the number of tuples of certain type is restricted.

Traditionally these kinds of requirements are expressed by consistency constraints in creating the database schema.

Our idea is to use consistency constraints in the same way as locks are used in traditional concurrency control. We call consistency constraints as *data restrictions* when they are used in concurrency control. So data restrictions differ from traditional consistency constraints in that they are short-lasting. As a data restriction is created and dropped by a workflow instance, its duration is at most the duration of the workflow instance. In addition, a data restriction differs from a consistency constraint in that it may be specific for a workflow instance that is generated from the input parameters of a workflow or task instance. Thus, different instances of a given workflow may set different kinds of data restrictions. This feature makes it possible to use as liberal data restrictions as possible.

By using data restrictions we can achieve the following:

- The expression power of data restrictions matches the workflow isolation requirements, which in turn makes it possible to achieve a high degree of concurrency.

- Workflow concurrency control can be implemented by utilizing the features of the underlying database system.

- Transactions executed outside the control of the workflow system can be controlled through the consistency constraints supported by the database system.

Although workflow isolation requirements can be enforced by data restrictions, in some cases it seems to be more appropriate to control the execution of workflows and tasks directly, i.e., restricting the execution of the programs behind the workflows, subworkflows and tasks rather than indirectly through the data they access. We call these restrictions *task restrictions*, independently on the object of the restriction (i.e., whether it is a workflow, a subworkflow or a task). They differ from traditional semaphores [BHG87] in the same way as data restrictions differ from traditional locks, i.e., the locking expression is based on semantic information.

Both data and task restrictions are given in a workflow specification. A workflow instance then sets and unsets restrictions according to the specification. Due to this property we call such workflows *self-isolating workflows*.

The rest of this chapter is organized as follows. First, in Section 7.1, we consider *assertions* [UW97], a type of consistency constraints in SQL. Then in Section 7.2, we extend the notion of a 2P-task to incorporate specific tasks for setting and unsetting restrictions (e.g., assertions). The specification and management of data restrictions are considered in Section 7.3, and the specification and management of task restrictions are considered in Section 7.4. Finally, the scheduling policy of self-isolating workflows is discussed in Section 7.5.

7.1 Assertions

SQL-systems provide a variety of techniques for expressing constraints as part of the database schema. These include key constraints, domain constraints, CHECK-constraints and assertions (also called general constraints). Using assertions it is possible to define any condition, while the other types are associated with other schema elements (e.g., with relations or domains). Using assertions it is possible to define constraints

- on the values of the attributes,

- on the values of relations as a whole, and
- on the relationships between several relations.

Like other schema elements, assertions are declared by a CREATE statement:

CREATE ASSERTION <name> CHECK <condition>

Each assertion has a name. The name makes it possible to add, modify and delete assertions at any time. For example, an assertion can be dropped by the statement DROP ASSERTION <name>.

Assertions allow us to enforce any conditional expression that can appear in a WHERE clause in SQL. The condition must have a boolean value, and therefore it is necessary to aggregate the results of a non-boolean expression in some way to make a single true/false choice, so that for example the operator NOT EXISTS is applied to an expression producing a relation.

The condition in an assertion must always be true, and any database modification whatsoever that causes it to become false will be rejected by the SQL-system, i.e., the transaction will be aborted.

SQL3 [UW97] extends SQL2 [UW97] assertions so that assertions can be specified to be triggered by programmer-specified events. As it is explicitly stated after which modifications the condition must be checked, the burden of the SQL-system can be alleviated.

In our approach, the isolation requirements of a workflow are expressed by assertions. A workflow instance passes an assertion request to the SQL-system, which either accepts or rejects the request. In principle, the SQL-system accepts the request, if it does not contradict with the state of the database, i.e., if the assertion evaluates to true. The SQL-system enforces that each accepted assertion is valid until the workflow instance requests to drop the assertion.

7.2 Isolated tasks

Self-isolating workflows set and unset data and task restrictions. Each restriction is attached to a workflow unit, i.e., to a workflow, a subworkflow, or a task. All the restrictions attached to a workflow unit are set in a specific task, called *Set isolation*, and, respectively, all the restrictions are unset in a task called *Unset isolation*. A workflow unit having such additional (sub)tasks is called an *isolated workflow unit*. To illustrate this a task T and the corresponding *isolated task T* are presented in Figure 7.1.

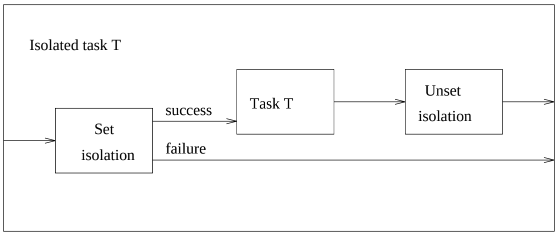

Figure 7.1: The structure of an isolated task T.

The task *Set isolation* may fail as any other task. Its failure means that the corresponding task instance has not succeeded in acquiring the required isolation properties. Therefore the task *Set isolation* is always a vital task. In contrast the task *Unset isolation* cannot fail semantically. That is, dropping a constraint may never cause an inconsistent state of the database.

By incorporating the notion of an isolated task into a 2P-task we get the structure presented in Figure 7.2 (cf. Figure 4.3). In order to associate

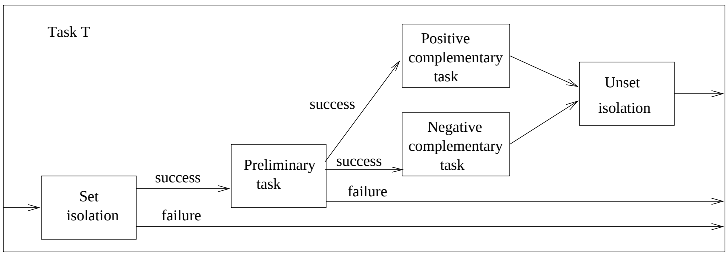

Figure 7.2: The structure of an isolated 2P-task.

an isolated 2P-task with the failure atomicity protocol we must add a participant task. This leads to the task structure presented in Figure 7.3 (cf. Figure 4.4). So the failure atomicity protocol synchronizes the execution of the Unset isolation tasks: they are not executed until the workflow is committed or aborted.

By inserting isolation tasks to 2P-workflows and 2P-subworkflows we can construct isolated 2P-workflows and isolated 2P-subworkflows.

The notion of isolated workflow units nicely matches the modular workflow structure: a task, subworkflow or a workflow may have variations that

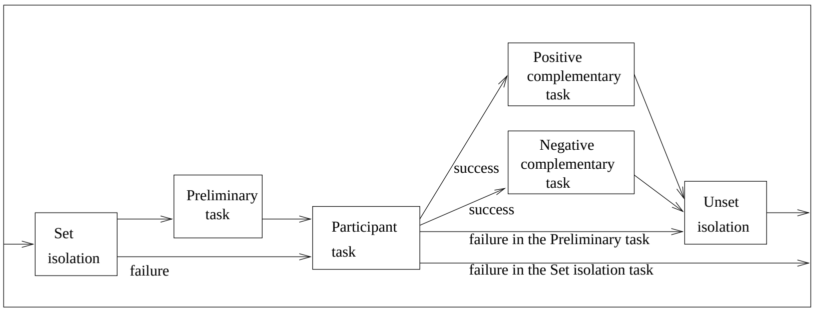

Figure 7.3: The execution graph of an isolated 2P-task.

differ only in the isolation task. For example, a workflow may not need isolation at all when executed alone (i.e., as an independent workflow) but it may have different isolation settings when appearing as a subworkflow in other workflows.

7.3 Data restrictions

7.3.1 Expressing data restrictions

The task *Set isolation* is a transaction that sets one or more restrictions (assertions). Setting an assertion fails if the assertion is not valid in the database. In general, each assertion is formulated in a way that it produces a relation, and the assertion is valid only if the relation is empty. If an assertion is not valid then *Set isolation* fails.

In order to illustrate the use of data restrictions we now return to the isolation problem of the *Loan request* workflow. Particularly we show how it can be solved by data restrictions. Later on, we will show how the same isolation problem could also be solved by using a task restriction.

The isolation problem of the *Loan request* workflow was that the decision of granting a loan is based among other things on the client's existing loans which were retrieved in the task *Credit check*. A necessary isolation requirement is that the loans of a client are granted in a serial way, i.e., all the previously granted loans are known when a decision to grant or reject a new loan request is done. To enforce this requirement we extend the workflow *Loan request* by the tasks *Set isolation* and *Unset isolation* as presented in Figure 7.4. The isolation tasks use the input parameters of the workflow *Loan request*. We assume that the workflow *Loan request* has three input

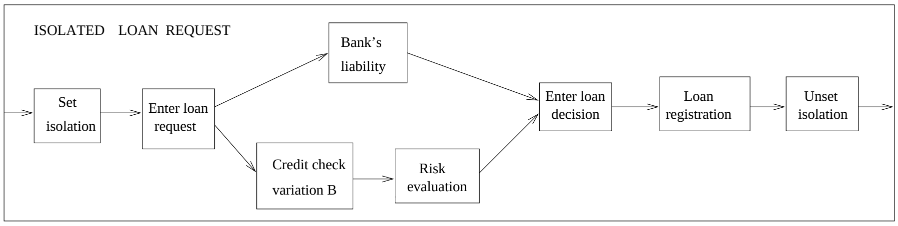

Figure 7.4: Isolated version of the workflow *Loan request processing.*

parameters: W-ClientNumber, W-ClientName, W-LoanAmount. So, an instance of the *Loan request* workflow is activated, e.g., by the call

Loan-request(12345, Smith, 10 000).

Now the task *Set isolation* is comprised of the following transaction named *LoanIsolation*:

```
Program LoanIsolation(W-clientNumber)
Begin Transaction
   SELECT COUNT(*) FROM Loans INTO :counter
         WHERE clientNumber = :W-clientNumber;
   CREATE ASSERTION <system generated name>
         (CHECK :counter =
         (SELECT COUNT(*) FROM Loans
         WHERE clientNumber = :W-clientNumber));
End Transaction
```

The transaction first counts from the relation *Loans* (Figure 7.5) the number of the loans the client has. Then the transaction creates an assertion, which is named the client number. If the transaction commits, then the restriction prevents adding new loans for the customer.

For example, the call LoanIsolation(12345, Smith, 10000) would ensure that no new loans for the customer 12345 (Smith) can be inserted until the data restriction is released, i.e., the task *Unset isolation* is executed. The transaction *Unset isolation* would be the following:

```
Program UnSetLoanIsolation(W-clientNumber)
Begin Transaction
   DROP ASSERTION <system generated name>
End Transaction
```

Loans	clientNumber	clientName	loanNumber	balance
	33	Smith	123	1000
	22	Ford	234	2000
	11	Taylor	345	5000
	33	Smith	456	7000

Figure 7.5: Relation *Loans*.

In the previous example we enforced an assertion that restricts the number of tuples of certain type. The two other common types of workflows isolation requirements are:

- a certain attribute stays within a specified range, and
- a certain tuple is not deleted.

To illustrate these types of restrictions we specify the restrictions *IncrementWatch* and *DeleteWatch* on the relation *Loans*.

The following version of the restriction *IncrementWatch* ensures that the loan balances of a client (specified by the input parameter) are not allowed to be increased at all.

```
Program IncrementWatch(W-clientNumber)
    Begin Transaction
    CREATE ASSERTION <system generated name>
    AFTER
        UPDATE OF balance ON Loans
    REFERENCING
        OLD AS OldTuple,
        NEW AS NewTuple
    CHECK (NOT EXISTS
        (SELECT balance FROM Loans
        WHERE clientNumber = :W-clientNumber AND
        OldTuple.balance < NewTuple.balance))
    FOR EACH ROW;
    End Transaction
```

The clauses OLD AS and NEW AS refer to the old and new values of the tuple to be updated.

By replacing the inequality in the WHERE clause by the inequality

NewTuple.balance $<$ 1.1$*$ OldTuple.balance AND
NewTuple.balance $<$ 10 000

the restriction would allow decrements and the increments which are less than 10 per cent as long as the new balance is less than 10 000.

The restriction *DeleteWatch* (assertion) ensures that the tuple corresponding a certain loan cannot be deleted.

```
Program DeleteWatch(W-loanNumber)
   Begin Transaction
   CREATE ASSERTION <system generated name>
   CHECK (EXISTS
         (SELECT * FROM Loans
         WHERE loanNumber = :W-loanNumber));
   End Transaction
```

7.3.2 Managing data restrictions

With traditional locking the conflicts of locks are represented by a conflict matrix, and there may be more than one lock on a data item as long as there are no conflicting locks. For example, a write-lock conflicts with another write-lock or read-lock, while there may exist many read-locks on a data item as far as there exists no write-lock on the data item. Further, if additional lock types, such as increment-locks and decrement-locks, are introduced, then the conflict matrix contains more than the basic four entries. If the scheduler produces conflict serializable histories, then the lock manager has to ensure that different transactions do not have conflicting locks on a data item.

Although we use restrictions as locks they have essential differences as compared with traditional locks. First, restrictions are stored in the database while traditional locks are stored in a volatile storage. Second, in scheduling self-isolating workflows the conflicts of data restrictions need not to be considered as each workflow instance specifies the isolation requirements it needs. That is, each workflow has a correctness criterion of its own, while with conflict schedulers (i.e., schedulers which produce serializable histories) each workflow has the same correctness criterion, namely the requirement of serializability.

In managing data restrictions it is necessary to check whether a requested data restriction (assertion) conflicts with the state of the database. If the request (assertion) conflicts with the current state of the database (i.e., if the SQL-system rejects the assertion as it evaluates to false) then the request is rejected; otherwise it is accepted. Hence, accepting a request for a data restriction means that the restriction is valid at the time it is accepted, and then the database system will enforce its validity until the data restriction is dropped.

To illustrate the management of data restrictions we will consider an instance of the relation *Accounts* (Figure 7.6), and the following three data restrictions which could easily be expressed as SQL-assertions.

Accounts	accountNum	clientName	balance
	55	Ford	2000
	22	Smith	800
	33	Smith	600

Figure 7.6: Relation *Accounts*.

Restriction *R1*: The balance of account 55 must be more than 1500.

Restriction *R2*: The balance of account 55 must be less than 1200.

Restriction *R3*: The balance of account 55 must be more than 1600.

Restrictions *R1* and *R2* are in a conflict in the sense that they cannot be valid simultaneously. However, such a conflict does not have to be detected as the acceptance of a data restriction is only based on the state of the database: if *R1* is accepted, then account 55 must have a value that prevents the acceptance of *R2*. Also note that the validity of *R3* also implies the validity of *R1*. However, such implications are meaningless as we do not know in which order the restrictions are dropped.

Further assume the existence of the following (traditional) consistency constraints:

Constraint *C1*: The sum of all the balances must be more than 10 000.

Constraint *C2*: The sum of the balances of each client must be positive.

Now assume that a workflow instance issues the following restriction:

Restriction *R4*: The sum of accounts 22 and 33 must be negative.

Clearly this restriction conflicts with constraint *C2*, and therefore it cannot be accepted. Indeed the SQL-system would always reject *R4* which implies that *R4* cannot be true in the database.

The fact that the system needs not find whether restrictions or constraints are in conflict eases the implementation and alleviates the burden of the workflow system.

7.4 Task restrictions

Workflow systems store information concerning the proceeding of workflow instances, e.g., which tasks are active, terminated or failed. Such *state information* is needed when the users are querying the state of certain workflow instances. Also the scheduling of workflow instances (i.e., enforcing the correct execution structure of workflow instances) requires the use of state information.

Our idea is that by setting restrictions on the state information (relations) it is possible to control the isolation (interleaving) of workflow instances. Especially, by setting a restriction on the state information means that certain execution states are not allowed. Such restrictions we call *task restrictions*.

We make a distinction between static and dynamic task restrictions.

- *Static task restrictions* are created by the workflow designer.
- *Dynamic task restrictions* are created and dropped by the workflow instances.

Analogous to data restrictions both kinds of task restrictions are expressed as assertions.

7.4.1 Managing task restrictions

To be more concrete we make some assumptions of the form of the state information. Particularly, we assume that for each workflow unit there is a relation called the *task relation*. Thus, there is a task relation corresponding with each task, subworkflow and entire workflow. The name of the task relation is the name of the workflow unit. Its attributes may include all or some of the input parameters of the workflow unit. There may also be additional attributes when needed. For example, if a workflow instance cannot be identified by its parameters, then the identification number of a workflow instance (generated by the system) must be used as an attribute.

An important point in using task restrictions is that before a workflow instance, say W, is allowed to execute a task T, it must succeed in inserting a tuple into the task relation T. However, the insertion fails if there is a task restriction (static or dynamic) on the task relation T that would evaluate to false after the insertion.

The creation and dropping of task restrictions are performed in the *Set isolation* and *Unset isolation* tasks. Analogous to creating a data restriction also the creation of a task restriction may fail. It fails if the task restriction is in conflict with the state of the task relation. So the management of task restrictions is analogous to the management of data restrictions, i.e., accepting a task restriction is independent of other active task restrictions.

When data restrictions and dynamic task restrictions are used we can divide the functions of the task *Set isolation* (concerning the task T) into three steps:

1. Set data restrictions (if any).

2. Set dynamic task restrictions (if any).

3. Insert a tuple in the task relation T.

If one of the above actions fails, then the task *Set isolation* fails. In the task *Unset isolation* all the tuples that have been inserted in the task *Set isolation* are deleted and all the task restrictions and data restrictions set are dropped.

7.4.2 An example of using task restrictions

We now show how the isolation problem of the workflow *Loan request* can be solved by task restrictions.

The parameters of the workflow *Loan request* are the client's name and number and the amount of the requested loan. We therefore assume that there is a task relation *LoanRequest*. Its attributes include the input parameters of the workflow and an additional attribute, *requestNum*, which is used to identify the different instances (Figure 7.7). The relation instance indicates that Smith has two active loan requests, and Robinson and Taylor each have one active loan request.

Now it is possible to set static and dynamic task restrictions on the relation *LoanRequests*. For example, setting a static task restriction which states that there may not be two tuples having equal values on the attribute *clientNumber* would solve the problem of concurrent loan requests. This can be done by the following SQL clause:

LoanRequests	requestNum	clientNumber	clientName	amount
	115	44	Robinson	2300
	143	33	Smith	500
	105	11	Taylor	70000
	123	33	Smith	4500

Figure 7.7: Relation *LoanRequests*.

```
ALTER TABLE LoanRequests ADD CONSTRAINT SingleWatch
    UNIQUE clientNumber
```

By using dynamic task restrictions we can specify still finer isolation requirements. In particular, different instances may set appropriate restrictions on the relation *LoanRequests* or on other task relations. For example, in the beginning of the processing of Taylor's loan request it is possible to set a task restriction on the task relation corresponding to the subworkflow *Credit check* which states that there may be at most one tuple where the attribute clientNumber equals to 11. Such a restriction can be expressed as follows:

```
Program RequestWatch(W-clientNumber)
    Begin Transaction
    CREATE ASSERTION <system generated name>
    (CHECK
            (SELECT COUNT(∗) FROM CreditCheck
            WHERE clientNumber = :W-clientNumber) ≤ 1);
    End Transaction
```

So, if the request RequestWatch(11) is accepted then during the processing of Taylor's loan request other workflows are not allowed to evaluate Taylor's credit worthiness.

Task restrictions can also be used for other purposes. For example, as in many workflows some tasks require human interaction, there may be reasons for imposing a limit on the number of simultaneous workflow instances according to the human resources available. A constraint to this effect can be simply enforced by assertions stating the maximal number of tuples in appropriate task relations, or by specifying more complex assertions. For example, with the workflow *Loan request* the maximal number of concurrent instances might be a function of the requested amounts.

Also with partially automated tasks and workflows there may be a need to control the time when they are started. For example, a short lasting and partially automated task should be started only on office hours, otherwise they may hold on some resources (e.g., on data) for an unnecessarily long time.

7.5 The scheduling policy of self-isolating workflows

In Section 3.4 we presented a workflow system architecture where concurrent workflow instances were scheduled by the isolation manager. We now analyse its scheduling policy.

In general, a *scheduler* is a program that controls the execution of concurrent activities. When it receives an operation it may immediately schedule it, delay it, or reject it. Each scheduler usually favors one or two of these choices. Depending on what the scheduler favors the distinction between aggressive and conservative schedulers can be done.

An aggressive scheduler tends to avoid delaying operations, and so it loses the opportunity to reorder operations it receives later on. A conservative scheduler tends to delay operations, and so it can reorder operations it receives later on. Each type of scheduler works well for certain types of application. We next consider which kind of scheduling policy is appropriate for the isolation manager. We first consider the scheduling of data restrictions and then the scheduling of task restrictions.

7.5.1 Scheduling data restrictions

The isolation manager (Figure 3.10) schedules the tasks that we have called "set isolation tasks", which in the subsequent analysis will be shortly called *SI-tasks*.

The isolation manager accepts an SI-task if it succeeds in setting the requested restriction on the database. Analogously a traditional locking scheduler accepts a lock request if it succeeds in inserting the requested lock in a lock table.

If the isolation manager fails in setting a restriction (i.e., the requested restriction is not true in the database), it rejects the SI-task. Hence, the failure of an SI-task causes the failure of the entire task as we have already illustrated in Figure 7.1.

The isolation manager is an aggressive or a conservative scheduler depending on whether it rejects the failed SI-task, or whether it places it in a queue internal to the isolation manager for waiting a subsequent execution. If the isolation manager places the failed SI-task in a queue it cannot remove it from the queue until the database is in a state where the requested data restriction is true. However, due to the static nature of data restrictions, it is not likely that after a delay the database will be in such a state. Therefore, we conclude that the isolation manager should not delay the failed requests. That is, only an aggressive scheduling policy is appropriate for scheduling data restrictions.

7.5.2 Scheduling task restrictions

An essential difference between a data restriction and a task restriction is that the former is a constraint on the database while the latter is a constraint on the information modelling the state of workflow instances (i.e., task relations). So, when task restrictions are used the isolation manager may also receive requests to set a restriction on a task relation and requests to insert a tuple in a task relation.

The isolation manager accepts a restriction on a task relation if it does not conflict with the current state of the task relation, and it accepts the insertion into a task relation if the insertion does not violate the consistency of the task relation.

As task relations model the state of workflow instances they are very dynamic. Workflow instances may also set and unset assertions (i.e., dynamic task restrictions) on them and thereby task relations are also dynamic with respect to their consistency constraints. Note that task relations are similar to traditional lock tables in that their content is also determined according to the current activities (transactions).

If the setting of a task restriction fails there is a tuple or a temporal constraint that caused the failure. However, unlike with data restrictions, it is likely that after a delay the tuple or the constraint that caused the failure will be removed, i.e., when the workflow instance(s) that caused the failure have been finished. This suggests that the isolation manager should use a conservative scheduling policy with task restrictions, i.e., place the failed request in a queue, and later on remove the request from the queue and execute it.

In general, scheduling policies which use delaying require a solution for the problem of deadlocks [BHG87]. A solution with workflows is to execute the tasks of a workflow in a predefined order even though the workflow determines only a partial order of the tasks. Also if two workflows have more than one common task then in both workflows the common tasks should be executed in a predefined order.

7.6 Discussion

We have considered the use of data and task restrictions in ensuring execution atomicity. The proposed method is flexible in the sense that many isolating requirements can be enforced by a data restriction or by a task restriction. Which type of restriction should be used depends on the isolation requirements. On the other hand, with some workflows it might be more appropriate to use syntactic information in specifying transactional properties. For example, in [PTV96b, PTV97] we considered *tasks' transactional integration* where it is possible to define which tasks comprise a unit of atomicity and which tasks comprise a unit of serializability. To illustrate this, assume that transaction Tw (withdrawing from an account) and Td (depositing to an account) are ACID-transactions, and we then want to form task Tt (transfer of money) by combining Tw and Td. Now it is is possible to define that Tt is a unit of atomicity while Tw and Td are the units of serializability. In this choice a user may see the money to be transferred on neither of the accounts, on one account, or on both accounts. This suggests that extending our semantic approach by the features of *transactional task integration* might increase the usability of the approach.

Chapter 8

Conclusions

We have studied the transactional properties of workflows, and their relationships to aspects such as workflow modularity and reusability. The notion of task variations was introduced to increase tasks reusability. The variations of a task may differ either in their functionality or in their transactional properties.

An important property of a workflow is its atomicity. Most transactional workflow models use some form of compensation to ensure the atomicity of a workflow. We introduced options to ensure failure atomicity. A workflow instance acquires options, and if the workflow instance can be committed then the options are eventually realized by installing the updates to the database. Through options we avoid the problems of dirty data.

We considered the implementation of options by specific two-phase tasks. A two-phase task divides task execution into two short-lasting ACID transactions, and thereby decreases the time the tasks hold their data resources. The coordination of two-phase tasks was carried out by the failure atomicity protocol. The protocol is similar to the two-phase commit (2PC) protocol in that it consists of a vote phase and a decision phase. It differs from the 2PC protocol in that its communication topology is not restricted to two levels. Nor does the commitment require yes-votes from all the participants, but only from the vital tasks.

The range of the applicability of options is wide. By allowing partially realizable options negotiation-like cooperation between tasks can be supported. The cooperation can be carried out by a protocol which only deviates from the 2PC protocol in that the spectrum of the negotiation is not restricted to atomic commitment.

A self-isolating workflow sets and unsets restrictions in a way similar to the way in which traditional transactions obtain and release locks. A restriction can be imposed on the data items on the database or on the state information of tasks. Both kinds of restrictions can be expressed as assertions in the SQL database language.

At this point we summarize the feasibility of the ideas we have presented. We evaluate the different approaches from four points of view.

- Failure atomicity: each workflow instance should reach an accepted termination state.

- Isolation: concurrent workflow instances should not interfere with each other.

- Performance: the workflow instances should hold data resources only when it is necessary to ensure the required isolation or atomicity properties.

- Usability: the specification of workflow's failure and isolation requirements should be intuitive and clear.

These aspects are highly related: each can be easily achieved at the expense of others. For example, the traditional correctness criteria of an atomic transaction and a serializable history are ideal in view of all but the performance aspect, and the criteria provide failure and execution atomicity. These criteria are also intuitive and clear: all or any of the actions are performed and the effect is equivalent to that of a serial execution. The application programmer does not have to take care of failures and concurrency. Unfortunately, this is not the case with semantic models, including our approach. Instead, the workflow designer must have a thorough insight into the application semantics to be able to write compensating transactions, or options and restrictions using our approach. However, there is probably no other way than the use of semantic models to preserve sufficient performance.

We assume that the correctness aspect is a strength of our approach. Through the expression power of SQL-assertions we can specify the isolation requirements of workflows. Further, through options we can avoid the correctness problems related to dirty data, and by using data and task restrictions it is possible to prevent those interleavings where concurrent activities would interfere with each other. On the other hand, we argue that the predefined isolation and atomicity properties provided by advanced transaction models (ATMs) are neither always necessary nor meaningful for all workflows. In fact, only in a very limited class of applications they can achieve the ideal correctness and performance.

We also argue that the performance aspect is a major strength of our approach. The more accurately the isolation requirements can be expressed the higher degree of concurrency can be achieved. Using the full expression power of SQL-assertions a workflow designer can specify very accurately the isolation requirements.

The usability is a critical issue with our approach. The problem (for us as well as with any model using semantic information) is that the workflow designer is burdened with writing options and restrictions. However, the threshold of this additional function is lowered as it can be done using SQL features familiar to database designers. On the other hand, workflow designers have to know the application well enough to be able to write options and restrictions. Writing options is similar to writing compensating actions in ATMs.

Probably, the only way to evaluate definitively our proposed transactional workflow would be to implement a workflow system that uses the ideas presented and to test the system in the case of some real applications.

References

[AAEA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. of the 12th International Conference on Data Engineering*, pages 574–583, 1996.

[AAEAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionalities and limitations of current workflow management systems. *IEEE Expert: Special Issue on Cooperative Information Systems*, 1997.

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure handling in large scale workflow management systems. Technical Report RJ9913, IBM Almaden Research Center, November 1994.

[AMG+94] G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. Technical Report RJ9912, IBM Almaden Research Center, November 1994.

[ANRS92] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proc. of the 18th International Conference on VLDB*, pages 65–76, 1992.

[ASSR93] M. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proc. of the 19th International Conference on VLDB*, pages 134–145, 1993.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. *SIGMOD Record*, 23(2):44–54, June 1994.

[BDS+93] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *SIGMOD Record*, 22(3):23–30, September 1993.

[BEK93] O. Bukhres, A. Elmagarmid, and E. Kuhn. Implementation of the flex transaction model. *IEEE Data Engineering Bulletin*, 16(2):28–32, June 1993.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BÖH+92] A. Buchmann, M. Özsu, M. Hornick, D. Georgakopoulos, and F. Manola. A transaction model for active distributed object systems. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 5, pages 123–158. Morgan Kaufmann Publishers, 1992.

[CC82] J.-M. Chang and S.-K. Chang. Database alerting technique for office activities management. *IEEE Transactions on Communications*, 30(1):74–81, January 1982.

[CGM88] W. Cellary, E. Gelembe, and T. Morzy. *Concurrency Control in Distributed Database Systems.* North-Holland, 1988.

[CR90] P. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transactions structure and behavior. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[CR92] P. Chrysanthis and K. Ramamrithan. ACTA: The saga continues. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 10, pages 349–397. Morgan Kaufmann Publishers, 1992.

[CR93] P. Chrysanthis and K. Ramamrithan. Delegation in ACTA as a means to control sharing in extended transactions. *IEEE Data Engineering Bulletin*, 16(2):16–19, June 1993.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the 16th International Conference on VLDB*, pages 567–577, 1990.

[CW92a] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. Technical Report 92-078, Politecnico di Milano, October 1992.

[CW92b] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proc. of the 18th International Conference on VLDB*, pages 339–351, 1992.

[Dat86] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1986.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, 1990.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A transaction model for long-running activities. In *Proc. of the 17th International Conference on VLDB*, pages 113–122, 1991.

[EGL98] J. Eder, H. Groiss, and W. Liebhart. The workflow management system Panta Rhei. In A. Dogac, L. Kalinichenko, T. Öszu, and A. Sheth, editors, *Workflow Management Systems and Interoperability.* Springer-Verlag, 1998.

[EGR91] C. Ellis, S. Gibbis, and G. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[EL95] J. Eder and W. Liebhart. The workflow activity model Wamo. In *Proc. of the Third International Conference on Cooperative Information Systems (CoopIS-95)*, 1995.

[EL96] J. Eder and W. Liebhart. Workflow recovery. In *Proc. of the International Conference on Cooperative Information Systems (CoopIS-96)*, 1996.

[Ell92] C. Ellis. A model and algorithm for concurrent access within groupware. Technical Report CU-CS-616-92, University of Colorado at Boulder, October 1992.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multibase transaction model for interbase. In *Proc. of the 16th International Conference on VLDB*, pages 507–518, 1990.

[FKB95] A. Forst, E. Kuhn, and O. Bukhres. General purpose workflow languages. *Distributed and Parallel Databases*, 3(2):187–218, 1995.

[GH94] D. Georgakopoulos and M. Hornick. A framework for enforceable specification of extended transaction models and transactional workflows. *Journal of Intelligent and Cooperative Information Systems*, pages 599–617, September 1994.

[GHM+93] D. Georgakopoulos, M. Hornick, F. Manola, M. Brodie, S. Heiler, F. Nayeri, and B. Hurwitz. An extended transaction environment for workflows in distributed object computing. *IEEE Data Engineering Bulletin*, 16(2):24–27, June 1993.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):592–620, 1995.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMGK+90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleisner, and S. Salem. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, University of Maryland, February 1990.

[GMGK+91] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested sagas. *IEEE Data Engineering Bulletin*, 14(1):14–18, March 1991.

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proc. of the 7th International Conference on VLDB*, pages 144–154, 1981.

[HOS90] W. Harrison, H. Ossher, and P. Sweeney. Coordinating concurrent development. In *Proc. of the Conference on Computer-Supported Cooperative Work*, pages 157–167, 1990.

[Jen91] K. Jensen. *Colored Petri Nets*. Springer-Verlag, 1991.

[KAGM95] A. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing high availability in very large workflow management systems. Technical Report RJ9967, IBM Almaden Research Center, July 1995.

[Kai90] G. Kaiser. Flexible transaction model for software engineering. In *Proc. of the Sixth International Conference on Data Engineering*, pages 560–567, 1990.

[Kam98] M. Kamath. *Improving correctness and failure handling in workflow management systems*. PhD thesis, University of Massachusetts, 1998.

[Kle91] J. Klein. Advanced rule driven transaction management. In *IEEE COMPCON*, pages 562–567, 1991.

[KLLO96] D. Kuo, M. Lawley, C. Liu, and M. Orlowska. A general model for nested transactional workflows. In *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA)*, 1996.

[KP92] G. Kaiser and C. Pu. Dynamic restructuring of transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 8, pages 265–295. Morgan Kaufmann Publishers, 1992.

[KPE92] E. Kuhn, F. Puntigam, and A. Elmagarmid. Multidatabase transaction and query processing in logic. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 9, pages 297–348. Morgan Kaufmann Publishers, 1992.

[KRSRS96a] G. Kappel, S. Rausch-Schott, M. Retschitzegger, and M. Sakkinen. From rules to rule patterns. In *Proc. of the 8th International Conference on Advanced Information Systems Engineering (CAiSE'96)*, pages 99–115, 1996.

[KRSRS96b] G. Kappel, S. Rausch-Schott, M. Retschitzegger, and M. Sakkinen. Multi-parent subtransactions covering the transactional needs of composite events. In *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA)*, 1996.

[KRSRS96c] G. Kappel, S. Rausch-Schott, M. Retschitzegger, and M. Sakkinen. A transaction model for handling composite events. In *Proc. of the Third International Workshop on Advances in Databases and Information Systems*, 1996.

[KS95] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[KU93] J. Klein and F. Upton. Constraint based transaction management. *IEEE Data Engineering Bulletin*, 16(2):20–23, June 1993.

[LMWF94] N. Lynch, M. Merritt, M. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[Lyn83] N. Lynch. Multilevel atomicity — a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):65–76, December 1983.

[MAA+95] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Günthör, and M. Kamath. Exotica: A project on advanced transaction management and workflow systems. *ACM SIGOIS Bulletin*, 16(1), August 1995.

[MAGK95] C. Mohan, G. Alonso, R. Günthör, and M. Kamath. Exotica: A research perspective on workflow management systems. *IEEE Data Engineering Bulletin*, 18(1):19–26, March 1995.

[MD89] D. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, 1989.

[Mos85] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1985.

[MSK+95] J. Miller, A. Sheth, J. Krys, K. Kochut, X. Wang, and A. Murugan. Simulation modeling within workflow technology. In *Proc. of the 1995 Winter Simulation Conference*, 1995.

[MSKW96] J. Miller, A. Sheth, K. Kochut, and X. Wang. CORBA-based run-time architectures for workflow management systems. *Journal of Database Management*, 7(1):16–27, 1996.

[NG92] K. Narayanaswamy and K. Goldman. "Lazy" consistency: A basis for cooperative software development. In *Proc. of the Conference on Computer-Supported Cooperative Work*, pages 257–264, 1992.

[NRZ92] M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 3, pages 53–85. Morgan Kaufmann Publishers, 1992.

[O'N86] P. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[PH88] C. Pu and N. Hutchinson. Split transactions for open ended activities. In *Proc. of the 14th International Conference on VLDB*, pages 26–37, 1988.

[PTV96a] J. Puustjärvi, H. Tirri, and J. Veijalainen. Concurrency control for overlapping and cooperative workflows. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(1):24–30, Spring 1996.

[PTV96b] J. Puustjärvi, H. Tirri, and J. Veijalainen. Managing overlapping transactional workflows. In *Proc. of the 8th International Conference on Advanced Information Systems Engineering (CAiSE'96)*, pages 345–361, 1996.

[PTV97] J. Puustjärvi, H. Tirri, and J. Veijalainen. Reusability and modularity in transactional workflows. *Information Systems*, 22(2&3):101–120, April 1997.

[Puu91] J. Puustjärvi. Distributed management of transactions in heterogeneous distributed database systems. *BIT*, 31(3):406–420, 1991.

[Puu92] J. Puustjärvi. Management of transactions in heterogeneous distributed database systems. Technical Report C-1992-31, University of Helsinki, 1992.

[Puu97] J. Puustjärvi. Negotiation transactions: An approach to increase the automation of workflows. In *Proc. of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97)*, pages 89–102, 1997.

[RELL90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin. Extending the transaction model to capture more meaning. *SIGMOD Record*, 19(1):592–620, 1990.

[RS95a] A. Reuter and F. Schwenkreis. ConTracts — a low level mechanism for building general-purpose workflow management-systems. *IEEE Data Engineering Bulletin*, 18(1):4–10, March 1995.

[RS95b] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems,The Object Model, Interoperability, and Beyond.* Addison-Wesley, 1995.

[Sar96] S. Sarin. Workflow and data management in Inconcert. In *Proc. of the 12th International Conference on Data Engineering*, pages 497–499, 1996.

[Sch93] F. Schwenkreis. APRICOTS — a prototype implementation of a Contract system — management of the control flow and the communication system. In *Proc. of the 12th Symposium on Reliable Distributed Systems*, 1993.

[Sch94] F. Schwenkreis. A formal approach to syncronize long-lived computations. In *Proc. of the 5th Australian Conference on Information Systems*, 1994.

[SR93] A. Sheth and M. Rusinkiewicz. On transactional workflows. *IEEE Data Engineering Bulletin*, 16(2):37–40, June 1993.

[SR96] A. Schwenkreis and A. Reuter. The impact of concurrency control on the programming model of ConTracts. In *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA)*, 1996.

[SRK92] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using polytransactions to manage interdependent data. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 14, pages 555–581. Morgan Kaufmann Publishers, 1992.

[TV95] J. Tang and J. Veijalainen. Enforcing inter-task dependencies in transactional workflows. In *Proc. of the Third International Conference on Cooperative Information Systems (CoopIS-95)*, 1995.

[UW97] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 1997.

[Wäc91] W. Wächter. ConTracts: A means for improving reliability in distributed computing. In *IEEE COMPCON*, pages 574–578, 1991.

[WDSS93] G. Weikum, A. Deacon, W. Schaad, and H. Schek. Open nested transactions in federated database systems. *IEEE Data Engineering Bulletin*, 16(2):4–7, June 1993.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[Wei93] G. Weikum. Extending transaction management to capture more consistency with better performance. In *Proc. of the 9th French Database Conference*, 1993.

[WR92] H. Wächter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 7, pages 219–263. Morgan Kaufmann Publishers, 1992.

[WS91] G. Weikum and H. Schek. Multi-level transactions and open nested transactions. *IEEE Data Engineering Bulletin*, pages 60–64, March 1991.

[WS92] G. Weikum and H. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Chapter 13, pages 515–554. Morgan Kaufmann Publishers, 1992.

[WS96] D. Worah and A. Sheth. What do advanced transaction models have to offer for workflows? In *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA)*, 1996.

[WWW+97] D. Wodtke, J. Weissenfels, G. Weikum, A. K. Dittrich, and P. Muth. The MENTOR workbench for enterprise-wide workflow management. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 576–579, 1997.

[WWWD96] D. Wodtke, J. Weissenfels, G. Weikum, and A. Dittrich. The Mentor project: Steps towards enterprise-wide workflow management. In *Proc. of the 12th International Conference on Data Engineering*, pages 556–565, 1996.

ISSN 1238-8645
ISBN 951-45-8642-5
Helsinki 1999
Helsinki University Printing House

Juha Puustjärvi: Transactional Workflows **A-1999-2**