# Research Report

**Advanced Transaction Models in Workflow Contexts**

G. Alonso D. Agrawal A. El Abbadi †
M. Kamath R. Günthör C. Mohan
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA
{*gustavoa,rgunther,mohan*}*@almaden.ibm.com*

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

**Research Division**
**Yorktown Heights, New York • San Jose, California • Zurich, Switzerland**

**Advanced Transaction Models in Workflow Contexts** *

G. Alonso D. Agrawal † A. El Abbadi †
M. Kamath ‡ R. Günthör C. Mohan
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA
{*gustavoa,rgunther,mohan*} *@almaden.ibm.com*

**ABSTRACT:** In recent years, numerous transaction models have been proposed to address the problems posed by advanced database applications. A few of these models have been implemented as prototypes but almost none are being used in a commercial product. In this paper, we make the case that such models are too centered around databases to be useful in real environments. Many of the new applications are heterogeneous, both in the supporting platforms and tools involved, and distributed over a wide geographic area. They raise a variety of issues that are not addressed at all by transaction models, which may explain the lack of success of the latter. These same issues, however, are the basis for many existing workflow systems, which are having considerable success as commercial products in spite of not having a solid theoretical foundation. We explore some of these issues and show that, in many aspects, workflow models are a superset of transaction models and have the added advantage of incorporating many ideas that to this date have remained outside the scope of traditional transaction processing.

---

*This work is partially supported by funds from IBM Hursley (Networking Software Division) and IBM Vienna (Software Solutions Division). Even though we refer to specific IBM products in this paper, no conclusions should be drawn about future IBM product plans based on this paper's contents. The opinions expressed here are our own.

†Part of this work took place while the authors were on sabbatical at IBM Almaden. Current address: Computer Science Department, University of California at Santa Barbara, CA 93106, USA. E-mail: {agrawal, amr}@cs.ucsb.edu.

‡Part of this work was done while the author was visiting IBM Almaden. Current address: Computer Science Department at the University of Massachusetts, Amherst, MA 01003, USA. E-mail: kamath@cs.umass.edu.

## 1. Introduction

It is a widely accepted fact that conventional databases are unsuitable for many applications. To address this problem, numerous advanced transaction models have been proposed [Elm92] but, to this date, most remain theoretical constructs and have not been implemented. There seems to be no interest in incorporating such models into products or even into prototypes. We believe, and this is the point we want to make in this paper, that the main reason for this state of affairs is the inadequacy of advanced transaction models to operate in real working environments. Advanced transaction models are too database-centric, which provides a nice theoretical framework but limits the possibilities and flexibility of the models. Furthermore, since they tend to remain theoretical models, they generally ignore a large number of important design issues [Moh94].

Paradoxically, there is a growing interest in tools to support distributed, heterogeneous applications very similar in nature to those envisioned by the designers of advanced transaction models. As a result of this interest there has been considerable effort to deliver *workflow* products intended for the management of *business processes*, to the point where nowadays there are more than 70 vendors who claim to have such systems [Fry94]. The goals of *Workflow Management Systems*, WFMSs, bear a strong resemblance to those of advanced transaction models, although addressing a much richer set of requirements. In this paper, we discuss the characteristics of workflow models and the notion of business processes by comparing them with existing transaction models. We show how workflow models have, in general, richer semantics and are more apt to be used in commercial products. In many aspects workflow models are a superset of advanced transaction models. We show this by implementing several advanced transaction models using a single workflow system. The main goal of the paper is to provide a better perspective of the relationship between advanced transaction models and workflow models. By analyzing and comparing the characteristics of both, we have developed a better understanding of the inherent limitations of the former and identified many points for improving on the latter. Our approach is not another attempt to merge different transaction models in one or to provide a general framework to program advanced transactions [BDG+94]. We believe those approaches are not suitable for advanced applications. Transactions are the accepted currency in the database community but they lack the expressibility and power of workflow models, even when compared with what is available in the research literature. We see workflow models as the natural step after advanced transaction models. They complement transaction models in many aspects and address an entirely new range of issues that make them more suitable for building applications.

The paper is organized as follows: Section 2 presents related work and motivation. Section 3 briefly describes the characteristics of workflow management systems. Section 4 discusses the implementation of a variety of transaction models in a workflow system. Section 5 describes a prototype tool to translate specifications of a transaction model into a workflow process. Section 6 concludes the paper. For reasons of space, the examples of execution of the different transaction models as workflow processes have been included in an appendix at the end of the paper.

## 2. Motivation and Related Work

In the last few years, several transactions models have been proposed to address non-traditional applications such as long running activities [DHL90, DHL91, GMGK+91b, ST94], extended transaction models [WR92, GHKM93, Elm92, Sal93, BDG+94], or multidatabases [ELLR90, MRSK92, ZNBB94] to name a few. In most cases, the models are developed from a database point of view, where preserving the consistency of the shared database by using transactions is the main concern. These models provide well-defined failure semantics in the sense of concurrency control and sophisticated recovery features. Although some of the ACID properties of transactions may be relaxed, the basic idea is always to use traditional transactions as building blocks. Along these lines, and taking advantage of the formalism inherent in database transactions, there have been many studies on the theoretical aspects of combining transactions into larger execution units. This has been done under several disguises: *intertask dependencies* [ASSR93, Kle91, G9̈3], modified *concurrency control* techniques [CR91], *inheritance* [Mos81], or *check/revalidate* mechanisms [WR92].

More recently, this trend has changed its focus towards *transactional workflows* [SR93] in an attempt to expand the goals of advanced transaction models. While the applications have been changed to reflect more realistic environments, many of the background assumptions on the nature of the system still remain. Much of the work done along these lines is still transaction based [TAC+93, MS93, GHS95, Hsu93, KS94, GHS95]. The motivation is to merge advanced transaction technology and workflow management systems to support business processes with well-defined failure semantics and recovery features. This work is based on an interpretation of the world from the database point of view. In doing so, these proposals limit their applicability and, in some cases, result in very restrictive models.

Parallel to this work, a wide range of *workflow management systems*, WFMSs, have become commercial products: *OPEN/Workflow* of Wang Laboratories, *ProcessIT* of AT&T, Fujitsu's *Regatta*, Sietec's *Staffware*, *Action Workflow* of Action Technology, Xerox's *InConcert*, IBM's *FlowMark*, among many others [Rei94, Fry94, GHS95]. Very few, if any at all, of these commercial systems incorporate the notion of transaction or other transactional concepts such as concurrency control, recovery, compensation, or formal dependencies. In their conception and design most of these systems are orthogonal to advanced transaction models and transactional workflows. The roots of these systems can be traced back to job control languages in the early 70's and the work done in office automation in the mid and late 80's [BP85]. These early systems concentrated on automation of office procedures and document management in local and centralized environments. Modern workflow management systems, however, provide support for complex long-running business processes executing in distributed, heterogeneous environments. As has been pointed out [GHS95], WFMSs lack the ability to ensure the correctness and reliability of the workflow execution in the presence of concurrency and failures. However, these are database concepts that cannot be interpreted in the same way in a workflow domain. While it is true that existing systems need to be enhanced to cope with more complex scenarios [Ley95, LA94, SJKB94, AAE+95, AGK+95, EL95, Bus95], they do provide a great deal of support for organizational aspects, user interface, monitoring, accounting, simulation, distribution, and heterogeneity. The success of existing systems is based on these features, and not on transactional aspects, which they obviously lack today.

This does not imply that transactional workflow, meaning workflow systems based on traditional transaction concepts and database oriented, should not play any role in future systems. However, the ideas and solutions derived from a transactional approach are only a fraction of the overall picture, much in the same way transaction management is only one of many components within current database systems. The similarity with databases can be taken even further. In spite of the very detailed models built for concurrency control [BHG87], the fact remains that most databases today use *Strict 2 Phase Locking* for write operations and inconsistent results are tolerated in the form of *dirty reads* [GR93]. Workflow systems are orders of magnitude more heterogeneous and distributed than databases, databases becoming just one more component of the workflow system, and the problems they pose in terms of performance are very complex. Successful systems will be required to be flexible and able to cope with environments where most activities are not of transactional nature. To tie a workflow system to a particular transaction model, or even to a combination of these models such as the one proposed in ASSET [BDG+94], will result in major restrictions that will limit its applicability and usefulness as a workflow tool.

## 3. Workflow Management Systems

Workflow is, in general, an ill-defined concept. Instead of trying to describe it precisely, we follow the *Workflow Management Coalition*, *WfMC*, [WfM95, Mem94] in providing a high level description of the model and functionality that a WFMS must support to be considered as such. When discussing particular implementation details, we use FlowMark [IBM95a, IBM95b, LA94, LR94], IBM's workflow product, which will also be briefly discussed. FlowMark follows very closely the reference model provided by the WfMC and the features used to implement different transaction models are found in many other workflow systems. Similar results can be accomplished with other settings. The important point is not so much which system is used but the fact that advanced transaction models can be implemented using current workflow management systems.

### 3.1. Business Processes

At the core of most workflow systems is the notion of a *business process* [HC93]. A business process, in general, is a set of activities with a common goal. The business process is built by linking together diverse activities, specifying the flow of data and control among them. Business processes tend to be of long duration, involve many users and tools over heterogeneous and distributed environments. Individual activities range from computer programs and applications to human activities such as meetings, phone calls or decision making. This is an important point, since the granularity at which workflow systems work is very coarse. For computerized activities, the granularity is that of the application. The workflow system has no way of controlling an application between successive invocations. This is in sharp contrast with the assumptions made in most transaction based systems.

### 3.2. Workflow Model

A workflow model is an acyclic directed graph in which nodes represent steps of execution and edges represent the flow of control and data among the different steps. The components described below follow the meta-model proposed by the Workflow Management Coalition [WfM95]. This model is only an abstraction and does not provide implementation details. These are described based on FlowMark's model, depicted in Figure 1:

- **Process**, a description of the sequence of steps to be completed to accomplish some goal. It should have a name, version number, start and termination conditions and additional data for security, audit and control. A process consists of *activities* and *relevant data.*
- **Activity**, or each step within a process. Activities have a name, a type, pre- and post-conditions and scheduling constraints. They can be *program activities* or *process activities.* A program activity has a program assigned to it that is executed when the activity is executed. A process activity has another process associated to it, so an entire process is executed when the activity is executed. Process activities are used for nesting and modular design. Each activity has an input data container and an output data container.
- **Flow of Control**: specified by *control connectors* between activities, is the order in which activities are executed. This corresponds to the *transition conditions* of the reference model.
- **Input Container**: a sequence of typed variables and structures which are used as input to the invoked application.
- **Output Container**: a sequence of typed variables and structures in which the output of the invoked application is stored.
- **Flow of Data**: specified through *data connectors* between activities, is a series of mappings between output data containers and input data containers to allow activities exchange information.
- **Conditions**, which specify the circumstances under which certain events will happen. There are three basic types of conditions. *Transition conditions* are associated with control connectors and specify whether the connector evaluates to true or false. A control connector that evaluates to false will not trigger the execution of the activity at its end. *Start conditions* specify when an activity will be started: either when all incoming control connectors evaluate to true - *and* condition - or when one of them evaluates to true - *or* condition. *Exit conditions* specify when an activity is considered to have terminated. After the execution of an activity the exit condition is checked. If true the activity has terminated, if false, the activity is rescheduled for execution.

An activity can be in one of the following states: *ready*, *running*, *finished* when the execution has completed, and *terminated* when execution has completed and the exit condition is satisfied. Activities can be started from the ready state either manually or automatically. Within a process,

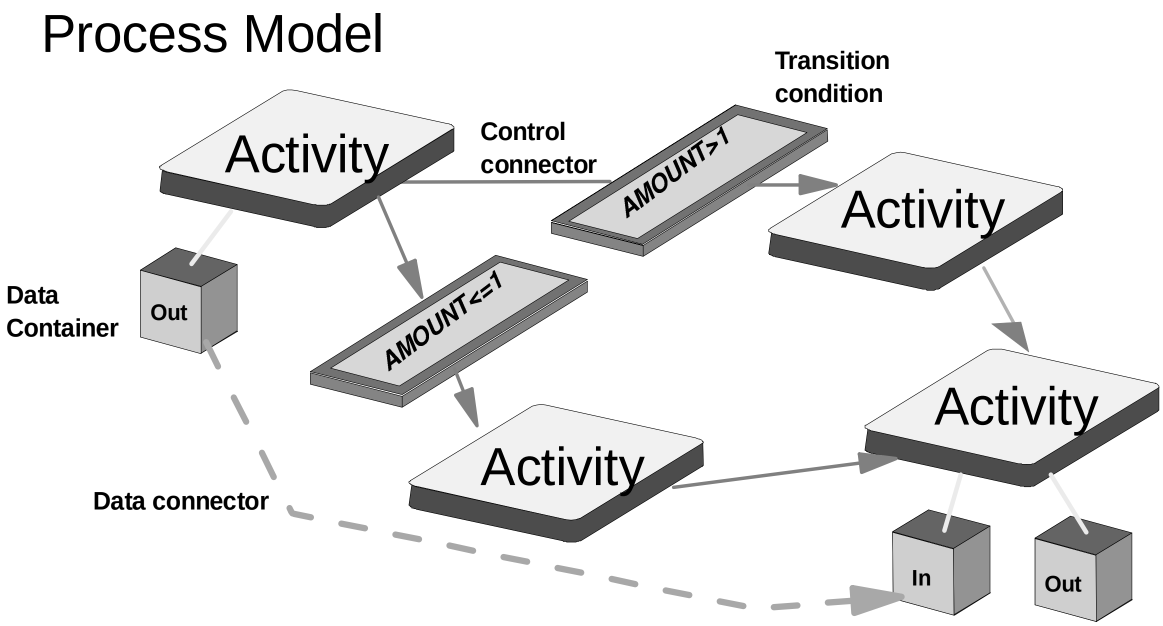


Figure 1: *Main components of FlowMark's model for control and data flow*

those activities without incoming control connectors are considered to be the *starting* activities of the process, and are set to the ready state when the process is started. Once an activity finishes, its exit condition is evaluated. If it is false, then the activity is reset to the ready state. Otherwise the activity is set to terminated and all the outgoing control connectors from that activity are evaluated. When the start condition for an activity is met, the activity is set to ready. If an activity will never be executed because its start condition evaluates to false, the activity is marked as terminated and all the outgoing control connectors from that activity are evaluated to false. This procedure is called *dead path elimination*. The process is considered finished when all its activities are in the terminated state.

In general, conditions increase the power and expressibility of the model. They provide the means for discarding some branches of the control flow and for implementing structures similar to *if-then-else*. Such features are not found in any transaction model, except in the ConTract model [WR92] which is more of a programming environment for reliable execution of sets of activities. Exit conditions can be used to implement loops, by embedding subprocesses within another process. For the purposes of this paper, we will refer to subprocesses as *blocks*. These embedded blocks or processes appear, at the higher level process, as an activity.

### 3.3. Workflow Features Not Found in Transaction Models

A WFMS considers four different sets of entities: *users*, *activities*, *programs*, and *data*. It controls and automates the interactions between elements of each set. It is the ability to integrate these four groups that sets WFMSs apart from transaction models. As outlined above, a WFMS automates the flow of control and data between activities, and maps activities to users and programs. Existing advanced transaction models limit themselves to only part of the problem. For instance, Sagas [GMS87] provide a limited form of flow of control and data between activities, but lack any reference to users or programs. Flexible transactions [ZNBB94], provide a more sophisticated flow of control, but no data, program, or user support. The ConTract model [WR92] provides flow of control and data similar to that of WFMSs but does not include users into the system.

Of the additional features provided by WFMSs, the most relevant is their ability to describe an organization and adapt the definition and execution of workflow processes to the particular characteristics of that organization. In a WFMS, the organization is described in terms of the roles, hierarchical levels and persons associated with it. A person can have several roles – manager, programmer, assistant – and a role can be assigned to several persons. When activities are defined, the workflow designer must specify who is responsible for the execution of the activity. This can be specified using a role, in which case all the persons that fit in that role are eligible to execute the activity. This provides a great deal of flexibility when executing a process. It is also possible to specify who must be notified if the activity is not executed within a certain period of time. Thus, activities do not happen automatically, as is assumed in advanced transaction models, but with direct user intervention. Even activities corresponding to programs that do not require human input for execution are associated with users who can monitor their progress and are responsible for their execution. The user can stop an activity, restart it, force it to finish, and so forth, independently of the rest of the process. This mapping between users and activities is possible in WFMS because of the granularity of the activities, which is that of applications, and not that of traditional transactions.

Moreover, activities in a WFMS can be of any type, not just computer programs, as long as there is a way to report their progress to the WFMS. WFMSs are not designed for transactions but for generic activities. In particular, in FlowMark, once a program is registered it can be invoked from any activity. An API interface is provided so the programs can access the data containers. When an activity is set to ready, it is determined who are the users eligible to execute the activity, and a notification is sent to each of them.

Regular users interact with the system using *worklists*. A worklist contains the activities that correspond to the user. Note that the same activity may appear in several worklists simultaneously, however, as soon as a user selects that activity for execution, it disappears from all other worklists. This can be effectively used to perform load balancing in the execution of a process. None of these ideas can be found in advanced transaction models.

Finally, a major difference between WFMSs and transaction models is in the area of correctness and reliability. Current WFMSs do not offer significant support for recovery and failure handling [GHS95]. In most cases, user intervention is required, either to solve consistency problems or to specify which activities are needed to recover from an exception. Transaction models, on the other hand, are in many cases motivated by these issues and many solutions have been proposed. However, it must be noted that since most advanced transaction models have not been implemented, the feasibility of these solutions is, in many cases, unclear. It must also be noted that in most WFMSs the execution of a process is persistent in the sense that forward recovery is always guaranteed, a feature not found in many advanced transaction models. In case of failures, the process execution will stop. Once the failures have been repaired, the process execution is resumed from the point where the failure occurred. There are some minor caveats to this behavior, especially considering most WFMSs treat the applications that actually run the activities as completely autonomous entities and the activities are not necessarily failure atomic. When a failure occurs it is possible that the activity was half-way executed, or even totally executed, but the WFMS had not been notified. In these cases the activity will be rescheduled to be executed from the beginning. It is the designer's tasks to do the appropriate checking and book-keeping to handle problems. Again, this is related to the granularity at which WFMSs operate.

## 4. Implementing Transaction Models using Workflow Tools

In this section we show how several transaction models can be implemented using a WFMS. An important point to note is that workflow models do not deal with the actual application semantics, i.e., the contents of the activities is orthogonal to the workflow process. As a result, workflow models cannot be used to implement transaction models based on semantics or internal operations of the transactions such as Split-Transactions [PKH88], for instance. However, we believe that such models will not be very useful in most real applications where activities are not transactional in nature.

### 4.1. Linear Sagas

Linear Sagas were originally proposed by García-Molina and Salem as a way to solve the problems related to long lived transactions [GMS87]. The model was later extended to parallel sagas and generalized sagas [GMGK+90, GMGK+91a, GMGK+91b]. For reasons of space the discussion will be limited to the linear sagas, but the same ideas apply to the more general case. The basic idea of the saga model is to allow a transaction to release resources before committing. A long lived transaction, or saga, is seen as a sequence of subtransactions that can be interleaved in any way with other transactions. Each of subtransaction is an ACID transaction that preserves database consistency. Partial executions of the saga are undesirable, if the saga aborts then subtransactions that have committed must be compensated. Thus, each subtransaction has a compensating transaction associated with it, which undoes any changes introduced by the subtransaction but does not necessarily return the database to the state it was before the subtransaction was executed.

More formally, let $T_1, T_2, ..., T_n$ be the subtransactions of a saga $T$. Let $C_1, C_2, ..., C_n$ be the corresponding compensating transactions. The system provides the following guarantee: either the sequence $T_1, T_2, ..., T_n$ is executed, or the sequence $T_1, T_2, ..., T_j, C_j, ...C_2, C_1$, for some $0 \leq j < n$, will be executed. In the original proposal only one level of nesting was allowed, i.e., only the top level saga and the subtransactions were considered

The translation of a linear saga into a workflow process is straightforward. Note that there are two phases of execution in a linear saga. In the first, the subtransactions are being executed. If they terminate successfully, the saga commits. However, if the saga aborts for any reason, then the

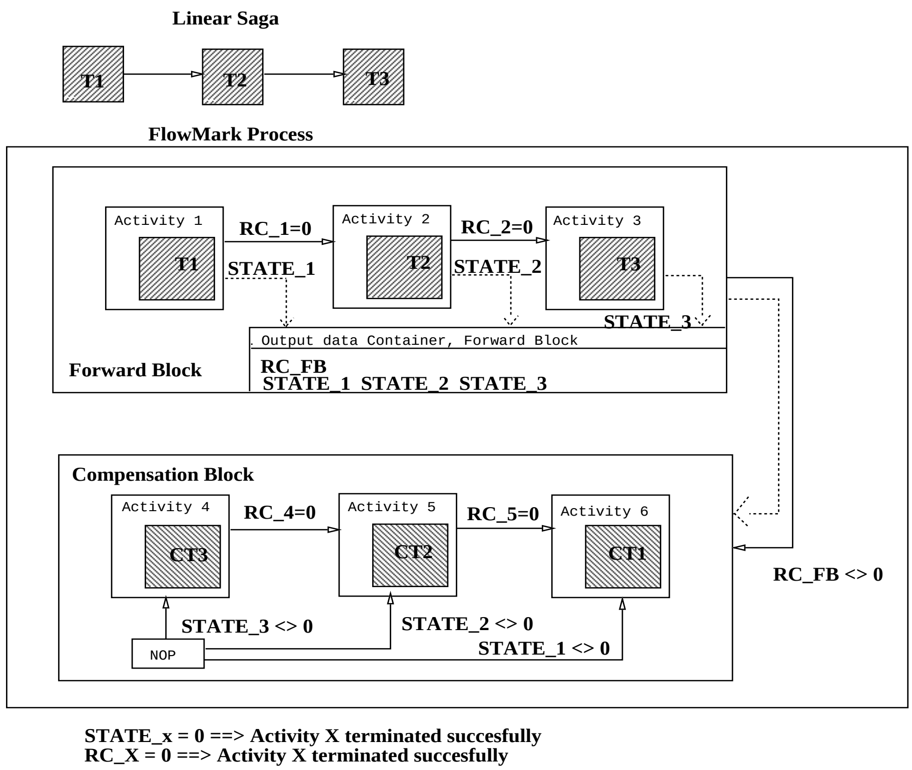


Figure 2: Translation of a linear Saga into a FlowMark process

second phase of execution takes place, compensating all the committed subtransactions. We use this idea in the translation. There are many other ways to perform this translation but we prefer the one presented here for its simplicity.

All the subtransactions of the saga are grouped into a block. The flow of control within the block reflects that of the saga, with each subtransaction represented as an activity. The control connectors have a condition associated with them, which is that the previous activity must have terminated successfully, i.e., the corresponding transaction has committed. When this is the case, the control connector evaluates to true and execution proceeds forward. If a transaction aborts, the corresponding control connector will evaluate to false, and by dead path elimination, no other activity in the block will be executed, and the block terminates. The result of the execution of the transaction, whether it committed or aborted, can be captured through the return code of the program. Each activity must also register its status, i.e., whether it has executed or not. This is done by mapping the return code of the output data container of each activity to the appropriate variable in the output data container of the block. When the block terminates, its output data container will contain a list of the status of each activity.

The second phase is implemented in another block containing the compensating activities in reverse order. There is also a null activity whose purpose is to trigger the execution of the compensation at the correct point. This activity is a no-operation, however it has control connectors to all the compensating activities. The condition on those control connectors is whether the corresponding forward activity was executed or not. This information is obtained by mapping the output data container of the forward block to the input data container of the compensating block. Thus, when the compensating block is executed – right after the forward block terminates – the starting null activity is executed, and the control connectors are evaluated. All those that correspond to activities that have executed will be activated, and compensation will proceed in the reverse order of execution starting from the last activity executed. This is shown in Figure 2. Note that strictly speaking, the last activity should not be compensated. In the original model, when the last activity commits, the entire saga commits. However, it is possible that users may require to compensate an already completed saga. In these cases all activities must be compensated.

Sagas introduced several valuable ideas: breaking a long running transaction into several units and establishing an explicit flow of control and data between them. These same ideas are also found in workflow systems, although in a more sophisticated form. As we have shown, Sagas can be implemented using a WFMS. Hence, applications for which Sagas is the appropriate model can also be modeled using a WFMS. This indicates that WFMSs can be used as "programming languages" to construct the particular execution model demanded by an application. Such an approach has the added advantage of being more versatile and provide features that do not exist in transaction models such as forward recovery, optional execution paths, and a clear separation between the flow of data and control from the transactions themselves.

### 4.2. Flexible Transactions

Flexible transactions work in the context of heterogeneous multidatabase environments [ELLR90]. In such environments, each local database acts independently from the others. Since a local database can unilaterally abort a transaction, it is not possible to enforce the commit semantics of global transactions [ZNBB94]. Flexible transaction were designed to address this problem.

A flexible transaction provides alternative execution paths, if a subtransaction is aborted, then a different subtransaction can be submitted in the hope that it will be successful. A flexible transaction commits if either the main subtransactions or their alternatives commit. For reasons of space we will not present here the formal model of flexible transactions. The interested reader can find the details elsewhere [MRSK92, ZNBB94]. However, it is interesting to discuss the main characteristics of the model. The one described here is similar to that of [ZNBB94].

A flexible transaction is a partial order of subtransactions. A subtransaction can be *compensatable*, *retriable*, or *pivot* [MRSK92]. A compensatable subtransaction is one whose effects can be undone after it commits by executing a compensation transaction. A retriable transaction is a subtransaction that will eventually commit if retried a sufficient number of times. A pivot subtransaction is one that is neither retriable nor compensatable. Note that it is possible for a subtransaction to be both compensatable and retriable. A flexible transaction is *well-formed* when the possible orders of execution do not violate the data dependencies between subtransactions and the flexible transaction is "atomic" (its effects can be undone or by retrying subtransactions it will

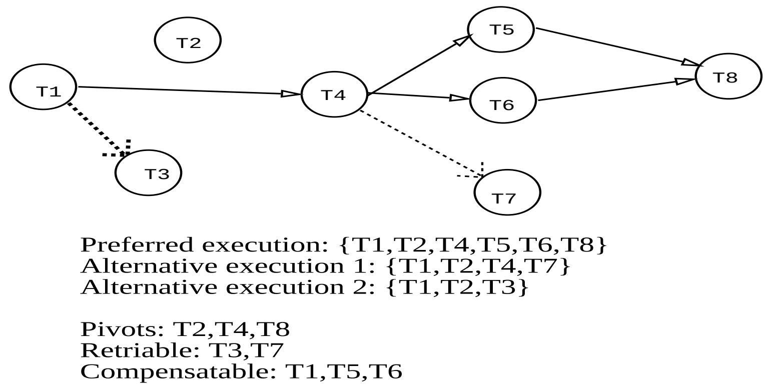


Figure 3: *An example of flexible transaction*

eventually commit). As has been shown [MRSK92], a well-formed flexible transaction contains at most one pivot subtransaction. Furthermore, all subtransactions that are non-retriable must be executed before the pivot, and all non-compensatable subtransactions must be executed after the pivot. Thus, before the pivot commits the transaction can always be rolled back, after the pivot commits the transaction has effectively committed. In [MRSK92] it is further assumed that there are no data dependencies among subtransactions. In [ZNBB94], it was noted that such restrictions apply only to the subtransactions that actually commits. As long as there is an alternative in case that a transaction aborts, there can be several pivots, and retriable and compensatable transactions can be interleaved. Correctness is guaranteeed by enforcing certain rules in the order of execution of the subtransactions and the overall structure of the flexible transaction. These rules, however, are beyond the scope of this paper, and in what follows we will assume well-formed flexible transactions.

As an example, consider the flexible transaction of Figure 3 [ZNBB94]. In this transaction, $T_1$, $T_5$ and $T_6$ are compensatable, $T_2$, $T_4$ and $T_8$ are pivot, and $T_3$ and $T_7$ are retriable. There are three possible execution paths: $p_1$=$\{T_1, T_2, T_4, T_5, T_6, T_8\}$, $p_2$=$\{T_1, T_2, T_4, T_7\}$, or $p_3$=$\{T_1, T_2, T_3\}$. The order of preference is $p_1$, $p_2$, $p_3$. First $T_1$ is executed, if it aborts, then the entire transaction is considered to be aborted. If it commits, then $T_2$ is executed. $T_2$ is a pivot subtransaction, therefore, once it commits, the entire transaction must commit. This is guaranteed by $T_3$, which is retriable, so if nothing else works, $T_3$ can be retried until it commits. When $T_4$ is executed, if it commits, then the transaction must follow either $p_1$ or $p_2$. $p_1$ is attempted first. If either $T_5$, $T_6$ or $T_8$ aborts, then $T_7$ is executed. Since it is retriable, the completion of the global transaction is guaranteed. In the case that $T_8$ is the one that aborts, $T_5$ and $T_6$ will be compensated before $T_7$ is executed.

Flexible transactions can be easily implemented using a WFMS. The only difficulty is to "mask" the roll back involved in a compensation as some form of forward progress. However, the characteristics of flexible transactions can be used to simplify the design. For instance, a pivot subtransaction must always be associated with a "way out". This is because if it aborts, there must be a way to either commit the transaction or compensate everything that has been executed so far. Thus, a pivot subtransaction becomes a branching point, depending on whether it committed or aborted.

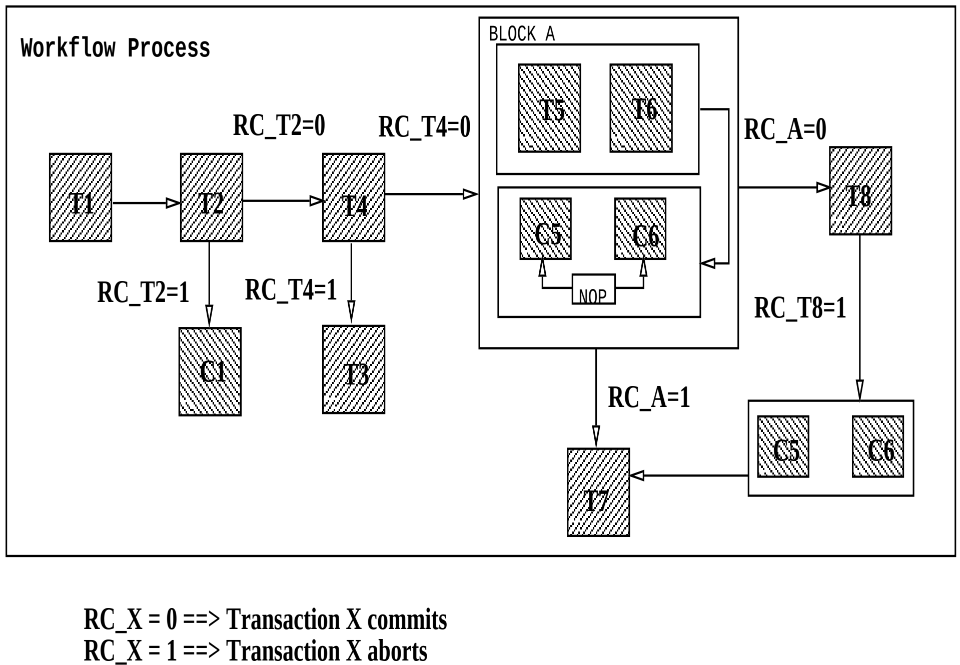


Figure 4: *A flexible transaction implemented as a workflow process*

Note also that the path between any two pivot subtransactions must contain only compensatable transactions. When a pivot subtransaction aborts all the subtransactions in the path must be compensated for until a point is reached in which there is an optional path. This compensation may be quite complex since many subtransactions may be involved and it is necessary to account for all possible executions. For simplicity, all compensatable subtransactions in the path between two pivot subtransactions that are not a bifurcation point of two optional paths will be grouped together into a single block. The status of the subtransactions, i.e., whether they committed or aborted, is passed as input data to the block.

We assume that the subtransactions, or the programs in which they are embedded, return a code indicating whether the transaction committed or aborted. Furthermore, we will assume the return code is 0 if the transaction aborted and 1 if it committed. For simplicity, we also assume that there is a specification of a flexible transaction in some notation - we will use graphs to better illustrate the process. The translation process is as follows:

1. Each subtransaction and compensating subtransaction of the flexible transaction corresponds to an activity. This is a one to one mapping, thus we will refer to pivot, compensatable, compensating and retriable activities.

2. The ordering among activities follows the ordering of the corresponding transactions. This is enforced by introducing control connectors between the activities.

3. Pivot activities have, at least, two outgoing control connectors. The transition condition for each of these connectors is the pivot transaction aborted and the pivot transaction committed, respectively.

4. Retriable activities have an exit condition that evaluates to false when the subtransaction aborts. In this way the activity is repeated until the subtransaction commits.

5. Compensatable activities that are not a bifurcation point for two optional paths, and that lay in the path between two pivot activities – or between the beginning of the transaction and a pivot activity – are grouped together in a block. They will have control connectors capturing the execution order among them, if any. When an activity terminates, the status of the corresponding subtransaction, committed or aborted, is recorded in the output data container of the block. This data container will be mapped into the input data container of the corresponding compensating block.

6. A block of compensatable activities has a corresponding block of compensating activities. The input to this block is the result of the execution of the activities. We introduce a no-operation activity for each of the compensating activities, connecting them with a control connector in which the transition condition is that the activity has committed. The connectors between the compensating activities are the same as those for the corresponding compensatable activities but reversed. Information about which activities were executed and which were aborted can be found in the input data container of the block. This is similar to the case of Sagas.

7. Changing from an execution path to another is done by compensating all the activities committed along the old path and starting another. Note that there is always a point, not necessarily unique, where it is possible to state that a path cannot be followed any longer. At these points the flow will be redirected to the corresponding compensating activities, if any, and the execution of the new path will be started. This can be represented as a linear succession of events by taking advantage of the dead path elimination features of FlowMark.

An example of this translation process is shown in Figure 4, which corresponds to the flexible transaction depicted in Figure 3. Flexible transactions introduce a very important aspect of advanced applications: how to "program" what to do if something goes wrong. It is unclear whether this is the most adequate paradigm, for designers cannot be expected to predict all possible errors. Moreover the requirements to build correct flexible transactions are very restrictive. As we have seen, flexible transactions can be built using a WFMS which will provide, in addition to the properties of flexible transactions, forward recovery and data flow mechanisms. It must be pointed out, however, that once outside the realm of transactions, the meaning of the model becomes unclear. For this reason WFMSs provide a much wider range of capabilities to deal with errors and exceptions.

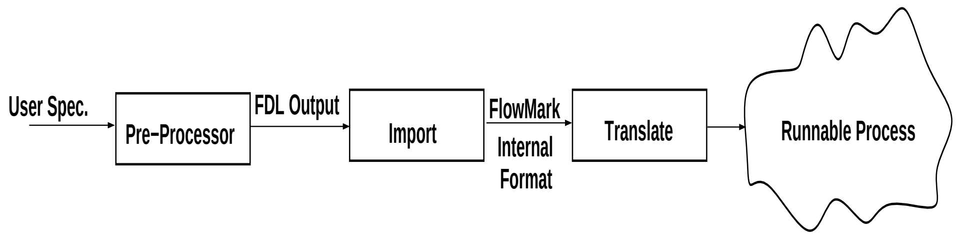


Figure 5: *Translation of User Specification of a Model into a FlowMark Process*

## 5. Exotica/FMTM as a Translation Tool

The translations described above are too complex to be performed by the user every time a process is built. To hide this complexity, we have designed a middleware module, *Exotica/FMTM*, which acts as a pre-processor that converts high level specifications of advanced transaction models into workflow processes. The block diagram of how the pre-processor fits into the overall FlowMark system is shown in Figure 5. The user creates a specification that contains the advanced transaction model to be used and the set of transactions to be executed. The pre-processor checks that the user specification meets the format of the advanced transaction model specified. It then takes the user specification and converts it into a FlowMark process in FDL (FlowMark Definition Language) format. Essentially this output is a program that defines all the activities within the FlowMark process with all the required control connectors. This FDL output is then imported into FlowMark and an internal representation of the process is created. During this conversion the import module checks for inconsistencies in the syntax of the process definition. Finally this internal format is translated into an executable FlowMark process. Here the translator checks the semantics of the FlowMark process to see if the specified user transactions are valid, i.e., *a suitable program definition exists*, if the control connectors are legal, etc. This executable FlowMark process is essentially a template that will be utilized to create run-time instances of the process. Based on our expertise in building this pre-processor, we can extend the pre-processor to convert any advanced transaction model specification into a correct FlowMark process implementation.

## 6. Discussion

WFMSs provide much broader functionality than that needed to implement ACID transactions, since their goal is to coordinate activities, users, programs and data, instead of just activities and data. This paper is a critique of the advanced transaction models proposed in the literature based on our belief that workflow models are a more appropriate framework to address advanced applications. In fact, since WFMSs are more flexible and in some aspects more general, they can be viewed as providing a ubiquitous programming environment for implementing a variety of such advanced models. In this paper we show this to be the case for both Sagas and Flexible transactions. WFMSs are still in their first generation and there are, undoubtedly, many areas where they need to be improved. In particular, it has been noted that they lack the functionality to cope with failures [GHS95]. This point deserves special attention. In conventional environments, coping with failures usually means to provide failure atomicity, i.e., a transaction is executed in its entirety or not at all [BHG87]. While more sophisticated failure handling capabilities have been widely discussed in the advanced transaction models literature, none of these techniques are currently being used in commercial, industrial strength systems. WFMSs provide forward recovery, but not atomicity, which is certainly not required in many cases. Moreover, existing WFMSs do not provide satisfactory solutions to the problem of exception handling. However, this is also true of transaction processing systems. It is important to make a distinction between these two characteristics. Recovery, in the database sense, is a well understood problem. Certainly it is an area in which existing WFMSs need improvement, but this is only a matter of time and the products reaching a more mature state. Exception handling is an entirely different matter. A workflow designer cannot predict every single possible case that may occur when a process is being executed. This is one of the problems that advanced transaction models try to address. For instance, through alternative execution paths, like in flexible transaction, or through compensation, like in Sagas. As we have shown in this paper, existing advanced transaction models can be implemented using a WFMS. Such transaction models provide a partial and limited solution to the problem of exception handling, and all of them can be used in the context of a WFMS. Still, they do not solve the problem. In this paper, we have demonstrated that transactional approaches to workflow management are not adequate since they do not address many of the issues that have made workflow systems so popular. Transactional properties will be used in some workflow activities and processes, but not in the majority of them. Schemas such as Sagas or Flexible transactions can be easily implemented in a WFMS, which provides the first realistic opportunity for these models to be used in a real environment. However, the solutions they provide to exception handling are very limited and certainly inappropriate for workflow environments where the main problem is not so much recovery but semantic exception handling. In this area, as in the other issues pointed out in the paper, workflow systems offer a much more comprehensive solution than advanced transaction models.

## References.


[AAE+95] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *IFIP WG8.1 Working Conference on Information System Development for Decentralised Organizations*, Trondheim, Norway, August 1995. See also http://www.almaden.ibm.com/cs/reports/workflow/exotica_home_page.html.

[AGK+95] G. Alonso, R. Günthör, K. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan. Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. In *Proceedings 3rd International Conference on Cooperative Information Systems*, Vienna, Austria, May 1995. See also http://www.almaden.ibm.com/cs/reports/workflow/exotica_home_page.html.

[ASSR93] Paul C. Attie, Munindar P. Singh, Amit Sheth, and Marek Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings 19th Conference on Very Large Databases (VLDB)*, Dublin, Ireland, August 1993.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proc. 1994 SIGMOD International Conference on Management of Data*, pages 44–54, May 1994.

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

[BP85] Giampio Bracchi and Barbara Pernici. The Design Requirements of Office Systems. *ACM Transactions on Office Information Systems*, 2(2):151–170, April 1985.

[Bus95] Christoph Bussler. User Mobility in Workflow-Management-Systems. In *Proceedings of the Telecommunications Information Networking Conference (TINA '95)*, Melbourne, Australia, February 1995.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transaction models. In *Proceedings 17th Conference on Very Large Databases (VLDB)*, pages 103–112, Barcelona, Spain, September 1991.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 204–214, June 1990.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transaction Model for Long-running Activities. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 113–122, August 1991.

[EL95] J. Eder and W. Liebhart. The Workflow Activity Model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems*, pages 87–98, Vienna, Austria, May 1995.

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, and M.E. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proc. of the 16th VLDB Conference*, August 1990.

[Elm92] A.K. Elmagarmid, editor. *Transaction Models for Advanced Database Applications.* Morgan-Kaufmann, 1992.

[Fry94] C. Frye. Move to Workflow Provokes Business Process Scrutiny. *Software Magazine*, pages 77–89, April 1994.

[G9̈3] R. Günthör. Extended Transaction Processing based on Dependency Rules. In *RIDE-IMS'93*, pages 207–214, Vienna, Austria, April 1993.

[GHKM93] Dimitrios Georgakopoulos, Mark Hornick, Piotr Krychniak, and F. Manola. Specification and Management of Extended Transactions in DOMS. In *RIDE-IMS'93*, pages 253–257, Vienna, Austria, April 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):119–153, April 1995.

[GMGK+90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-transaction Activities. Technical Report CS-TR-247-90, Department of Computer Science, Princeton University, 1990.

[GMGK+91a] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-transaction Activities. In *Proceedings IEEE Spring Compcon*, 1991.

[GMGK+91b] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling Long-Running Activities as Nested Sagas. *Bulletin of the Technical Committe on Data Engineering, IEEE*, 14(1):18–22, March 1991.

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

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

[HC93] M. Hammer and J. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution.* HarperBusiness, New York, 1993.

[Hsu93] M. Hsu. Special Issue on Workflow and Extended Transaction Systems. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[IBM95a] IBM. *FlowMark - Managing Your Workflow, Version 2.1.* IBM, March 1995. Document No. SH19-8243-00.

[IBM95b] IBM. *FlowMark - Modeling Workflow, Version 2.1.* IBM, March 1995. Document No. SH19-8241-00.

[Kle91] J. Klein. Advanced Rule Driven Transaction Management. In *36th IEEE Computer Society International Conference CompCon Spring 1991*, pages 562–567, San Francisco, California, March 1991.

[KS94] N. Krishnakumar and A. Sheth. Specifying Multi-system Workflow Applications in METEOR. Technical Report TM-24198, Bellcore, May 1994.

[LA94] F. Leymann and W. Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[Ley95] Frank Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *GI-Fachtagung Datenbanken in Büro Technik und Wissenschaft - BTW'95*, Dresden, Germany, March 1995. Springer Verlag.

[LR94] F. Leymann and D. Roller. Business Processes Management with FlowMark. In *Proc. 39th IEEE Computer Society Int'l Conference (CompCon), Digest of Papers*, pages 230–233, San Francisco, California, February 28 – March 4 1994. IEEE.

[Mem94] Workflow Management Coalition Members. Glossary, A Workflow Management Coalition Specification. Technical report, The Workflow Management Coalition, November 1994. Accessible via: http://www.aiai.ed.ac.uk/WfMC/.

[Moh94] C. Mohan. Advanced Transaction Models - Survey and Critique, 1994. Tutorial presented at ACM SIGMOD International Conference on Management of Data. Available at http://www.almaden.ibm.com/cs/reports/workflow/tran_models_tutorial_sigmod94.ps.Z.

[Mos81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, MIT Press, 1981.

[MRSK92] S. Mehrotra, R. Rastogi, A. Silberschatz, and H.F. Korth. A Transaction Model for Multidatabase Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 56–63, June 1992.

[MS93] D.R. McCarthy and S.K. Sarin. Workflow and Transactions in InConcert. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[PKH88] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings 14th Conference on Very Large Data Bases (VLDB)*, pages 26–37, Los Angeles, California, 1988.

[Rei94] B. Reinwald. Workflow-management. Tutorial handout, 13th IFIP World Computer Congress, Aug. 1994.

[Sal93] K. Salem. Implementing Extended Transaction Models Using Transaction Groups. Technical Report CS-TR-3051, Univ of Maryland, April 1993.

[SJKB94] H. Schuster, S. Jablonski, T. Kirsche, and C. Bussler. A Client/Server Architecture for Distributed Workflow Management Systems. In *Proc. of Third Int'l. Conf. on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[SR93] A. Sheth and Rusinkiewicz. On Transactional Workflows. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[ST94] B. Salzberg and D. Tombroff. A Programming Tool to Support Long-Running Activities. Technical report, College of Computer Science, Northeastern University, Boston, 1994. Technical Report NU-CCS-94-10.

[TAC+93] C. Tomlison, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, and D. Woelk. Workflow Support in Carnot. *Bulletin of the Technical Committee on Data Engineering*, 16(2), June 1993. IEEE Computer Society.

[WfM95] The Workflow Management Coalition, Workgroup 1 A. *The Workflow Reference Model*, February 1995. Document No. WFMC-WG01-1000, See also http://www.aiai.ed.ac.uk/WfMC/.

[WR92] H. Waechter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo, 1992.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proc. 1994 SIGMOD International Conference on Management of Data*, pages 67–78, 1994.

## Appendix: Execution Examples.

This appendix describes the execution examples of the transaction models outlined in the paper.

### Sagas.

Using Figure 2 as a reference, the execution of a saga is as follows. The entire saga is treated as a process. Thus, when the process gets started, the first activity is selected for execution. In this case the first activity happens to be a process activity corresponding to the forward block. When this activity is executed, the block is executed. Hence, the starting activities within the block are selected for execution. In this case it is activity 1. Activity 1 executes by invoking transaction $T1$. When it terminates it provides a return code $RC_1$. If this return code is 0, the activity terminated successfully, so the flow of control proceeds to activity 2. Activity 1 also produces some output data, variable $State_1$, which indicates whether activity 1 executed successfully or not. This can be the same as the return code, however we make it explicit here to simplify the exposition. This variable is mapped from the output data container of activity 1 to the output data container of the forward block. A similar process takes place with activities 2 and 3. If both of them execute successfully, the block terminates with a return code $RC_{FB}$ equal to 0. This return code is used as the transition condition between the forward and the compensation block. In the case that it is 0, the compensation block is not executed. By dead path elimination it is marked as finished and the entire process terminates. If $RC_{FB}$ is not 0, this indicates that some of the activities could not execute and, therefore, compensation should take effect. In order to do so, the output data container of the forward block is mapped to the input data container of the compensation block. This allows the compensation block to access the data about the status of each activity. Such data, stored in the variables $State_i$, is used as the transition conditions of the control connectors between the nul $NOP$, activity and all others in the compensation block. If an activity did not execute, its compensation will not take place since its start condition will never become true. The overall effect of this mechanism is that compensation will start at the activities that were executed last. Using return codes similar to those mentioned before, the execution of compensations takes place in the reverse order in which the activities were executed. Finally, note that compensations are in general considered retriable, in the sense that the compensation must be executed. If it fails, it should be retried until it succeeds. This can be done by using the exit condition of the activities. Compensation activities will not finish until the return code from the transaction indicates that it has committed.

**Flexible Transactions.**

The textual description provided in the paper is rather complex, however the process can be illustrated with an example that will help to understand the intuition behind these concepts. Note that, once there is a fixed notation for specifying flexible transactions, this translation process can be easily automated. The flexible transaction depicted in Figure 3 is implemented as the workflow model shown in Figure 4.

FlowMark executes the process of Figure 4 as follows: first $T_1$ is executed. If it aborts, the return code is 0 and therefore the outgoing control connector from $T_1$ is deactivated. As a consequence, and since $T_2$ does not have any other incoming control connector, $T_2$ is marked as terminated by dead path elimination. All other activities will be marked as terminated following a similar mechanism and the overall process eventually terminates, which corresponds to the desired behavior. If $T_1$ commits, then the return code is 1 and the control connector evaluates to true. Then $T_2$ is executed. If $T_2$ aborts, return code is 0, the compensation for $T_1$ is executed and, by dead path elimination, all other activities will be marked as terminated. When $T_2$ commits, $T_4$ is executed. If $T_4$ aborts, $T_3$ is executed until it successfully commits. All other activities are marked as terminated by dead path elimination. Upon successful completion of $T_4$, the block that contains $T_5$ and $T_6$ is started. If both transactions commit, $T_8$ is executed. If either one of $T_5$, $T_6$ or $T_8$ aborts, control is given to the compensation block containing $T_5^{-1}$ and $T_6^{-1}$. Using the data connector, the return code for both $T_5$ and $T_6$ is available in the compensating block. $T_5^{-1}$ and $T_6^{-1}$ are executed depending on whether their corresponding transaction committed or not. Once the compensating block commits, $T_7$ is executed until it commits.