

## Swinburne Research Bank

<http://researchbank.swinburne.edu.au>



Kuo, D., Lawley, M., Liu, C., & Orłowska, M. (1996). A general model for nested transactional workflows.

Originally published in *Proceedings of the International Workshop on Advanced Transaction Models and Architectures (ATMA 96), held in conjunction with the 22nd International Conference on Very Large Data Bases (VLDB 96), Goa, India, 31 August–02 September 1996.*

Goa: Department of Computer Science and Technology, Goa University.

Copyright © 1996.

This work is reproduced in good faith. Every reasonable effort has been made to trace the copyright owner. For more information please contact [researchbank@swin.edu.au](mailto:researchbank@swin.edu.au).

# A General Model for Nested Transactional Workflows\*

Dean Kuo<sup>†</sup>  
Div. of Information Technology  
CSIRO  
GPO Box 664  
Canberra ACT 2601 Australia  
dean.kuo@cbr.dit.csiro.au

Michael Lawley Chengfei Liu Maria Orłowska  
CRC for Distributed Systems Technology  
Department of Computer Science  
The University of Queensland  
Qld 4072 Australia  
{m.lawley,c.liu,m.orłowska}@dstc.edu.au

## Abstract

This paper applies concepts from transaction processing to workflows, thus enabling workflows to exhibit relaxed transactional behaviour. A general model for transactional workflows is presented. We define correctness of transactional workflows in terms of the model, and use the model to find schedules such that the execution of a transactional workflow is guaranteed to terminate in one of its acceptable termination (commit or abort) states.

We allow a transactional workflow to consist of a number of tasks composed by the constructs ordering, contingency, alternative, conditional and iteration. In addition, we show how the model can be used to support proper nesting of transactional workflows to reduce the difficulty of scheduling algorithms and to provide for structured and modular TWF specification.

**Keywords:** Transactional workflows, tasks, failure atomicity, model, correctness, schedules.

## 1 Introduction

With current advances in inter-operability between different hardware and operating systems [Dou92, SHM94], many (distributed) applications are now taking advantage of this technology by accessing data and executing processes in a heterogeneous distributed environment. These data sources may be database management systems (DBMSs), files systems or spreadsheets, etc. The processes may be legacy applications or services executed on some machine in the distributed environment. A process may even be a human oriented activity. Many organisations, especially in telecommunications, take advantage of inter-operability for the integration of their existing (legacy) systems. Two major problems associated with inter-operability are: the management of failures (node and communication failures), and management of concurrent executions.

The concept of transactions [BHG87, GR93, LMWF93], which originated from research in centralised databases, is designed to deal precisely with these problems but for a restricted environment. Transactions provide the well-known ACID (atomicity, consistency, isolation, durability) properties. That is, a transaction either executes in its entirety or not at all, it takes the state of the system/database from one consistent state to another, the execution is not affected by other concurrently executing transactions, and if the transaction commits, its effects are permanent (durable). These properties are managed by a transaction processing system.

---

\* The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

<sup>†</sup>Work done while the author was at the CRC for Distributed Systems Technology

In a distributed environment, the role of the transaction processing monitor is to manage the execution of distributed transactions. There are a number of protocols for distributed transactions. The most well known protocol is the two-phase commit protocol (2PC) [BHG87, GR93]. The transaction processing monitor co-ordinates the *commit* of a distributed transaction by *preparing* each sub-transaction of the distributed transaction. If any of the *prepares* fail, the sub-transactions are aborted; otherwise, they are committed. If one of the databases does not support the 2PC protocol, distributed transactions can still be supported by *preparing* those sub-transaction that are executing on databases that support 2PC. The transaction processing monitor then requests that the sub-transaction that is executing on the database that does not support 2PC to commit. If the commit fails, the other transactions are aborted; otherwise they are committed. However, if there is more than one database that does not support 2PC, then it is impossible to support distributed transactions.

Another protocol is the optimistic commit protocol [LKS91], however it requires that each sub-transaction be compensatable – the sub-transaction can be semantically/logically undone after the sub-transaction has committed. The transaction manager simply commits each sub-transaction and if any of them aborts, the transaction manager then aborts those sub-transactions that have already committed by executing their compensation.

Another property that a transaction manager exploits to support distributed transactions is forcibility. Transactions may be forcible. That is, the underlying system can guarantee that the transactions will eventually commit even if failures occur. For example, a deposit into a bank account that does not have an upper limit is forcible. It is straightforward to support (distributed) transactions if all of the sub-transactions are forcible.

Each of the approaches described above requires that the sub-transactions conform to their specific set of criteria when in fact distributed transactions that are composed of a mixture of sub-transactions that are compensatable, forcible, and support 2PC, can be supported. The transaction processing manager needs to execute those sub-transactions that are compensatable first, then co-ordinate the execution of those sub-transactions that support 2PC and then execute the forcible sub-transactions. Our research in transactional workflow generalises these methods of supporting transactions in a distributed environment. It also relaxes failure atomicity (all commit or all abort) of transactions. In this paper, we do not deal with the issues of execution atomicity (management of concurrent executions). This is beyond the scope of this paper but is part of current research.

Transactional workflows provide a mechanism for an application programmer to write applications without the need to deal explicitly with failures. Unlike transactions, transactional workflows allow the application to define the level of failure atomicity required by the application and there is no requirement that each resource supports the 2PC protocol. A transactional workflow (TWF), which may be long-lived, consists of a number of tasks composed by ordering, contingency, alternative, conditional and iteration constructs. The tasks may be a traditional transaction, a service, human oriented activity or another TWF. A TWF relaxes failure atomicity of transactions by allowing the specification of which tasks are non-critical and which tasks do not require undo.

A TWF commits if and only if all the critical tasks commit, and all the non-critical tasks either commit or abort. Similarly, a TWF aborts if and only if all of the tasks that require undo are in their initial or abort states and tasks that do not require undo are in their initial, abort or commit states. A TWF system manages failures such that a workflow application programmer does not need to deal with them in an ad-hoc manner. In summary, a TWF system would be a more general and flexible approach to the issues addressed by transaction processing monitors [GR93].

To provide a TWF system with the ability to reason and correctly execute a TWF, we need a model. This gives us two levels; the specification level which an application programmer uses to specify the TWF and a model level to which the specification can be mapped and which the TWF system can use to reason about the TWF and to derive a schedule such that the TWF is guaranteed to either commit or abort. A model should be as simple as possible, but able to represent all the constructs that a TWF application programmer would expect of a TWF system.

One of the contributions of this paper is a mechanism to support the nesting of TWFs – a TWF may be a task in another TWF. For large TWFs there will be many tasks, so reasoning about them to determine correctness and generate schedules may be computationally expensive. Using nesting, determining correctness of, and schedules for, the larger TWF only requires examining the (derived) task properties of any nested TWFs instead of examining the whole TWF.

Recently, transaction processing monitors such as Tuxedo [Lab91] and Encina [Tra92] have enjoyed a rise in popularity and are becoming more widely used. However, most applications only use them as middleware – to communicate with the resources in a homogeneous environment by providing gateways to the resources, for load balancing and for location independence – but not for transaction processing. This is due to the requirement that each resource in the distributed environment needs to support the 2PC protocol and due to the poor performance of the 2PC protocol. There are also a number of commercial multidatabase management systems (MDBMSs) such as Digital’s DBI [Dig94]. Their approach to supporting (distributed) transactions requires that all but one of the DBMSs support 2PC. If this condition is not satisfied, then transactions are not supported and users are advised not to perform any updates since inconsistencies may arise. At the other end of the spectrum, there are commercial workflow systems such as LinkWorks [Lin93] and FlowMark [LR95]. They, however, do not provide any transactional behaviour – that is, they do not manage failures or concurrent executions – and are biased towards workflows of human oriented tasks – creating, updating and passing documents around. These products often use email to pass the objects around an organisation.

Other related work is on advanced transaction models (ATMs). A number of ATMs are presented in [Elm92]. These models require that each resource supports 2PC or each transaction is compensatable. The ATM are designed for specific application domains, and are consequently inflexible. There has been some work on transactional workflows [RKT<sup>+</sup>95, RS94]. They support only a subset of the constructs in our transactional workflow specification model. Furthermore, they do not support nesting and they concentrate more on supporting a workflow of transactions instead of providing relaxed transactional behaviour to workflows. The aims in [YW93] and [AAE<sup>+</sup>95] are similar to our aims for workflows. However, we present a formal model for transactional workflows, define correctness for failure atomicity, provide a richer set of constructs and allow nesting. Furthermore, our model models the commit protocol of tasks and do not assume that all tasks are compensatable.

In summary, our research focuses on using the properties of the tasks to generate a schedule such that we can guarantee that the TWF either commits or aborts. This approach is more general than the approach taken in [RKT<sup>+</sup>95], ATMs, and transactions since we do not require that each task is compensatable or supports 2PC. In addition, we allow the relaxation of failure atomicity.

The rest of this paper is organised as follows. In section 2, we give a motivating example which highlights the uses of a TWF system. In section 3, we give a detailed description of TWFs and tasks, then in section 4 we present the model. In section 5, we show how the model represents the constructs, such as contingency, that are in the specification level. A conclusion and suggestions for future work are given in section 6.

## 2 Example

In this section we introduce a motivating example to illustrate the various features of TWFs (and its associated model). The example is that of a software installation/upgrade process across a distributed network.

### Problem Description

The following is a brief outline of the problem parameters:

- A distributed set of client/server machines running V1 (version 1) client and V1 server software.

- Both client and server software needs to be upgraded to V2 (version 2).
- V2 servers can support both V1 and V2 clients.
- V2 clients can only talk to V2 servers.
- Both V1 and V2 servers can be running simultaneously in the network, but only on different nodes.
- The V1 server cannot be running during the installation of the V2 server.

The TWF specification involves performing some system-wide configuration, then upgrading each of the nodes in the network. This is illustrated in Figure 1. Note that the two task nodes in this transactional workflow are in fact transactional workflows themselves. The double box indicates that this task does not require undoing should the transactional workflow abort. The single box indicates a task that is both critical and must be undone on failure, while an oval node represents a decision node.

Figure 2 shows the sub-transactional workflow *Perform Configuration* and provides an example of a contingency plan. If we fail to get the old configuration information (i.e., that pertaining to the installation of the V1 server), then we must attempt to get a new set of configuration information (presumably by querying the user).

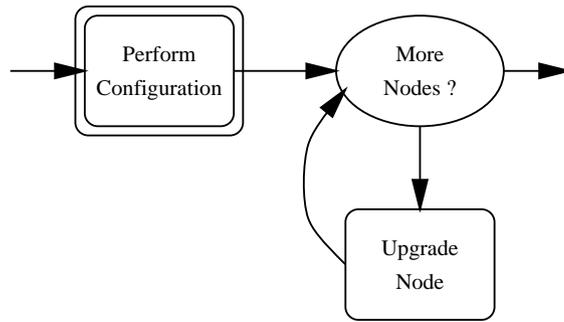


Figure 1: Perform entire upgrade

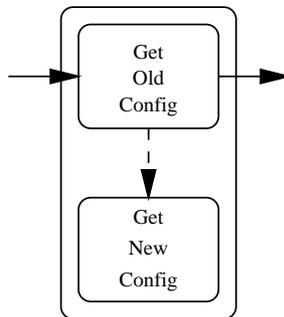


Figure 2: Get configuration information

Figure 3 shows the specification for *Upgrade Node*. The dotted box indicates that the task is non-critical. That is, the transactional workflow should not abort if the task fails. Also, if the server is upgraded but the client upgrade fails, we do not need to undo the upgrade to the server.

Finally, Figure 4 shows how to upgrade the server on a node. The split boxes indicate a task (top) and its compensation (bottom). If the last task (Test Server) fails, then the transactional workflow must be

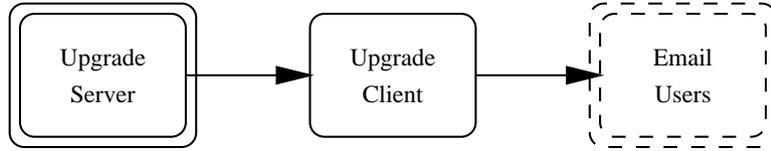


Figure 3: Upgrade a node

aborted (backward recovery). This involves compensating or rolling back each task that requires undoing. In this case, the server will be stopped, the new version (V2) de-installed (putting the V1 server back in place), then the server (now V1) started again.

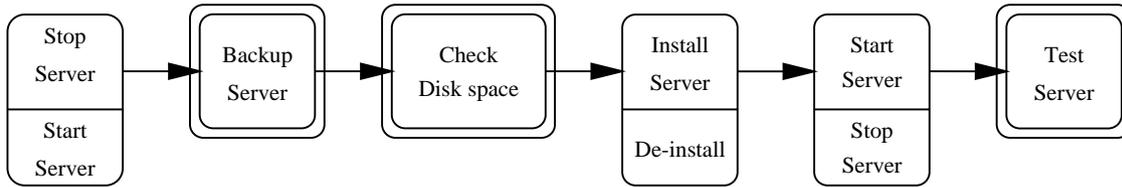


Figure 4: Upgrade the server

### 3 Transactional Workflows

In this section, we describe transactional workflows, their properties and how they are composed from tasks.

#### 3.1 Tasks

A task can be a function, an application, a service or an activity carried out by some process. The process may even be an human oriented activity. The TWF system does not deal with the actual semantics of the tasks but manages the order of execution of the various stages of the task in a TWF. The system assumes that the application programmer has implemented the TWF according to a meaningful conceptual specification.

Associated with each task is a finite state machine consisting of a set of visible states and a set of labelled transitions between these states. All tasks have an *initial* state, a set of *commit* states, and a set of *abort* states. The initial state represents the state of a task before it is invoked. If a task has been successfully executed then the task will be in one of its commit states while if the task failed in its execution, then the task will be in one of its abort states. Most tasks will have one commit state and one abort state. The reason why we have multiple commit and abort states is to enable us to model contingency and alternatives easily (see Section 5.2).

Each task essentially has two properties which affect the schedules of a TWF. They are *compensatability* and *forcibility*. A task is forcible if the underlying system can guarantee that the task will succeed. Also, a transition in a task may be forcible. For example, if a DBMS supports 2PC, then the transition from the *prepared* state to the *commit* state is forcible. However, the transition to the transaction's *prepare* state is not forcible since the DBMS may abort the transaction. In summary, a task or a transition (between states of a task) is forcible if the TWF system or the underlying system can guarantee that, in the presence of failures, the task or a transition of the task will eventually succeed.

The other property is compensatability. If a task can be *logically* undone after it has successfully executed (committed) then we say that the task is compensatable. The TWF system does not deal with the actual semantics of the compensation. It is up to the application programmer to define a task's compensation such that the task is undone meaningfully according to a specification. The TWF system views a task which has been undone via its compensation as an aborted task even though there may be some side effects. The example in section 2 shows an example of a compensation. The task *de-install* is the compensation of the task *install*.

A task's compensation can be defined as part of the task so the task is undone through the same compensation action independent of which TWF executed the task. However, for some applications (TWFs), this approach is too restrictive since TWFs may have different requirements and therefore need to define a task's compensation specifically in the context of the TWF. We, therefore, also allow a task's compensations to be defined within a TWF. This compensation is then executed if the task requires compensation even if a compensation is also defined as part of the task. In [RKT<sup>+</sup>95], each activity (task) must have a compensation defined and it must be defined within the activity. Thus, the tasks described in this paper are more flexible and general.

### 3.2 Composing Tasks to form Transactional Workflows

A transactional workflow is composed of an arbitrary number of tasks using a number of constructs. We have identified a collection of constructs which one would expect from such a TWF system. These are ordering, contingency, alternative, conditional and iteration. In the future, if we discover other useful constructs, then we shall extend our work to include them.

**Ordering** A basic requirement of a TWF specification is the specification of the order in which the tasks may be executed. A TWF can be represented as an acyclic graph in which each node corresponds to a task, and an edge represents an ordering constraint. Each task has a set of visible states. Every edge between a pair of nodes (tasks)  $tk$  and  $tk'$  in a TWF graph is labelled with two states  $vs$  and  $vs'$ . The state  $vs$  (respectively,  $vs'$ ) must be one of the visible states of the task  $tk$  (respectively,  $tk'$ ). The edge specifies that it is illegal, in this TWF, for the task  $tk'$  to be in any state after<sup>1</sup>  $vs'$  until the task  $tk$  is in state  $vs$  or some state after  $vs$ . Typically, an edge specifies that a task may not start until some other task has committed.

**Contingency** The execution of a task in a TWF may fail. A contingency plan is composed of two tasks. It allows the specification of a contingency task for the main task. The TWF system will execute the contingency task if the main task aborts. During the execution of a TWF, if a task fails in its execution, the TWF can still make forward progress (commit) if the contingency task can be successfully executed. An example usage of a contingency is shown in Figure 2 where the task fails to get the old configuration information and the contingency is to get the information from the user. Another example is making airline reservations. If we fail to reserve a seat on QANTAS then the contingency task may be to reserve a seat on British Airways.

**Alternative** An alternative is composed of an arbitrary number of tasks. It specifies a number of alternatives to achieving a goal. The TWF system has the freedom to choose the task to execute in the alternative such that the TWF is guaranteed to terminate in one of its acceptable termination states. The choice may depend on the properties of the task. Unlike a contingency plan, there is no constraint that a task must abort before another task can commit.

---

<sup>1</sup> Each task is described by a set of visible states and transitions and they form a directed acyclic graph (a partial order). Thus, *after* is well defined.

**Conditional** Conditionals (*if* statements) are also supported by our TWF model. They are composed of a condition and two tasks. It is only during execution the TWF system will be able to determine which of the two tasks is executed. The condition is based on the some data in the distributed system – for example, if the balance of an account is greater than a certain amount then invoke a particular task; otherwise some other task.

Conditionals are not supported in advanced transaction models such as DOM [MHG<sup>+</sup>92] and related research in transactional workflows [RKT<sup>+</sup>95, RS94].

**Iteration** An iteration contains a condition and a subtask as shown in figure 5. While the condition is true, the subtask is repeatedly invoked. In section 5, we show how to encapsulate the iteration – the condition and the subtask – as a single task. In this manner we are able to support iterations while still requiring that the TWF graph is acyclic.

We say that this task commits if all the invocations of the subtask commit. This condition can be relaxed by encapsulating the subtask in a contingency plan. With the appropriate use of counters and conditions, we can effectively relax this constraint to produce an iteration where the subtask commits exactly  $m$  times after  $n$  invocations. A typical usage of this is when the application requires that a task commits once but the specification only allows it to try a fixed maximum number of times.

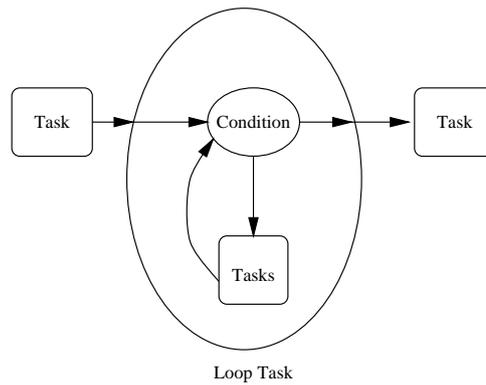


Figure 5: Iteration

### 3.3 Relaxing Failure Atomicity

By allowing an application programmer to specify the non-critical tasks and tasks that do not require undo in a TWF, failure atomicity is relaxed. We represent, in our model, non-critical tasks and tasks that do not require undoing using the constructs described earlier. A task,  $tk$ , that is *not* critical in a TWF is represented by a contingency plan with a null task<sup>2</sup> as its contingency task. Thus, if the task  $tk$  aborts then its contingency task is executed. The null task will always commit, therefore, the contingency plan will commit even if the task  $tk$  aborts. A task in a TWF that does not require to be undone even if the TWF aborts is represented as a task with a null compensation. Thus, we can define, in our model, that a TWF commits if all the tasks in the TWF are in one of their commit states, and aborts if the tasks are either in their initial state or in one of their abort states. This keeps our model simple without losing expressibility.

The example shown in section 2 for upgrading a node (Figure 3) gives an example of critical tasks and tasks that do not require undo. In the example, it is not critical to successfully send email to users

<sup>2</sup>The null task is a task which has no effect and will always commit when executed (is forcible).

for the TWF to commit, and it is not necessary to undo the server upgrade if the TWF for upgrading a node aborts.

## 4 The Model

In this section, we present the model.

### 4.1 Tasks

The model represents a task as a non-deterministic finite state machine with an arbitrary number of states. Each task has an initial state, a set of intermediate, commit and abort states, a set of labelled state transitions and a name. A formal description of a task is given in figure 6. The initial state represents the state of the task before it is executed. The commit (respectively, abort) states represent the task as successfully (respectively, unsuccessfully) executed. A transition is composed of an *old* state, an *action* (label) and a *new* state. The *old* and *new* states must be states of the task. Also, there must be at least one transition from the *initial* state, a transition to each *commit* state, and a transition to each *abort* state. Finally, the states and transitions of a task form a directed acyclic graph (a partial order).

---

```
task_name_t : string;

task_state_t : string;

action_t : string;

transition_t = record
    old_state : task_state_t;
    action : action_t;
    new_state : task_state_t;
end;

task_t = record
    name : task_name_t;
    initial : task_state_t;
    intermediate : set of task_state_t;
    commit : set of task_state_t;
    abort : set of task_state_t;
    transitions : set of transition_t;
end;
```

Figure 6: Tasks

---

In our model, a task may have multiple commit states. That is, there are a set of states of a task that represent that the task has been successfully executed (committed). A task may also have multiple intermediate and abort states. These are required to model contingencies and alternatives. This is more general than the model presented in [RS94] since they only allow a task to have one commit state, one abort state and at most two intermediate states.

Forcibility, between states of a task, is derived from the transitions. A task's behaviour is not always deterministic. For example, when the task shown in figure 7 is in its *initial* state and the action *execute*

is issued, the state after the action is either the *ready* state or the *abort* state. However, when the task is in its *prepared* state and the action *commit* is requested, then the transition is deterministic. We say that a task is forcible from a state  $s$  to a set of states  $S$  if it is possible to manage the execution of the task from state  $s$  to one of the states in  $S$  even if some of the transitions are non-deterministic. If a task is forcible from its initial state to one of its commit states, then we say the task is forcible.

A task is compensatable if a task has a transition from each of its commit states to one of its abort states. However, if there are only transitions to abort states from a subset of the commit states, then the task is only compensatable from this subset. It is a requirement that a task's compensation must be forcible – that is, if a system failure occurs during the execution of the compensation, then the underlying system will continuously re-execute the compensation until the compensation task successfully executed.

Definition 1 formally defines forcibility. In our model, we have a set of commit and abort states which is why we need to define forcibility to a set of states and not to a single state. From this definition, we can then identify the set of states that are forcible to the task's commit and abort states. In this definition,  $forcible(tk, s, S)$  means that the state  $s$  of the task  $tk$  is forcible to one of the states in the set  $S$  where  $S$  is a subset of the states of the task  $tk$ .

**Definition 1** *Let  $tk$  be a task,  $S'$  be the set of states of  $tk$ ,  $s \in S'$  and  $S \subseteq S'$ .*

*Then  $forcible(tk, s, S)$  is true if and only if*

- $s \in S$  or
- $\exists act \forall s, s' ((s, act, s') \in tk.transitions \text{ and } forcible(tk, s', S))$ .

From our definition, a state  $s$  of a task is forcible to a set of states  $S$  does not require that there exist a sequence of deterministic transitions from state  $s$  to one of the states in  $S$ . If one of the transitions is non-deterministic, we require each possible new state of the transition be forcible to  $S$ . Suppose task  $tk$  is a transaction that supports the 2PC protocol, then the finite state machine for the task is shown in figure 7. Notice that the task is forcible to the abort state in every state except the commit state<sup>3</sup> and the task is forcible to its commit state only when the task is in its prepared state. Managing the execution of a TWF relies on the knowledge of which states of a task are forcible to their commit and abort states. Definitions 2 and 3 define the states of a task that are forcible to commit and abort respectively.

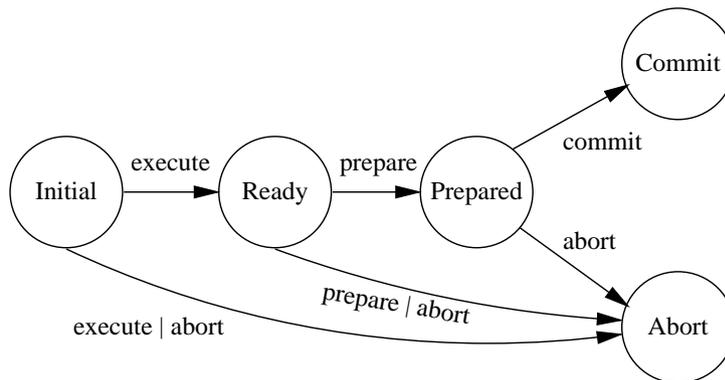


Figure 7: State Transition Diagram for a 2PC Tasks

**Definition 2** *Let  $tk$  be a task. Then*

$$forcible\_commit(tk) = \{s \mid forcible(tk, s, tk.commit)\}.$$

<sup>3</sup>if the task is compensatable, then the task is also forcible to the abort state from its commit state.

**Definition 3** Let  $tk$  be a task. Then

$$forcible\_abort(tk) = \{s \mid forcible(tk, s, tk.abort)\}.$$

## 4.2 Transactional Workflows

In this section, we model TWFs. The formal description of TWFs is given in figure 8. A TWF is represented by a name, a set of tasks and a set of illegal states. We do not model the constructs contingency, alternative, conditional, iteration and their associated tasks explicitly. They are modelled as tasks in a TWF and in sections 5, we describe how they are represented as a single task. Ordering is represented by a set of illegal states. We show in section 5 how to map an ordering constraint to a set of illegal states of a TWF.

A TWF is composed by a set of tasks. Thus, the state of a TWF is the set of states of the tasks in the TWF. This is represented as a set of task name and task state pairs. For example, a state of the TWF comprising tasks  $\langle tk_1, tk_2, \dots, tk_n \rangle$  is initially

$$\{\langle tk_1, Initial \rangle, \langle tk_2, Initial \rangle, \dots, \langle tk_n, Initial \rangle\}.$$

That is, each task in the TWF is in its initial state.

A TWF system manages the execution of a TWF from its initial state to one of its termination (commit or abort) states. The set of illegal states defines a set of constraints on the execution of the TWF. A TWF is not permitted to enter any of its illegal states during its execution. The illegal states of a TWF enables the model to represent the constructs such as ordering.

---

```
twf_state_t = set of <task_name_t, task_state_t>;
```

```
twf_t = record
  name : string;
  tasks : set of task_t;
  illegal : set of twf_state_t;
end;
```

---

Figure 8: Transactional Workflows

---

A TWF system needs to first find an execution of the TWF such that it never enters one of its illegal states and it always terminates in one of its commit or abort states. If no failures occur, then the TWF commits. We introduce the notion of *safe* states to define correct TWF executions. Informally, a TWF state is safe if it is guaranteed that the TWF can get to one of its acceptable termination states even if failures occur. During the execution of a TWF, each intermediate state must be a safe state.

If all the tasks in a TWF are in a state where they are forcible to one of their commit (respectively, abort) states, then the TWF is forcible to its commit (respectively, abort) state. Also, if one of the tasks is in a state which is neither forcible to commit or abort, but all the other tasks are both forcible to commit and abort, then the state is still safe since, if the non-forcible task commits or aborts, we can still force the other tasks to a state such that the TWF commits or aborts. These are the safe states of a TWF. In distributed transaction processing, all the sub-transactions of the distributed transaction are forcible to their abort states at all times up to and including their *prepared* state. When all the sub-transactions are in their *prepared* states, then the distributed transaction is forcible to commit and abort. However, when one of the sub-transaction commits, the state of the distributed transaction is no

longer forcible to abort (we assume here that the sub-transactions are not compensatable) but it is still forcible to commit and, therefore, a safe state.

Definitions 4, 5 and 6 define the states of a TWF that are forcible to commit, forcible to abort and safe states respectively.

**Definition 4** *Let  $twf$  be a transactional workflow and  $s$  a state of  $twf$ .*

*Then,  $s \in twf\_forcible\_commit(twf)$  iff*

$$\forall \langle name, state \rangle \in s, \exists tk \in twf.tasks, tk.name = name \text{ and } state \in forcible\_commit(tk)$$

**Definition 5** *Let  $twf$  be a transactional workflow and  $s$  a state of  $twf$ .*

*Then,  $s \in twf\_forcible\_abort(twf)$  iff*

$$\forall \langle name, state \rangle \in s, \exists tk \in twf.tasks, tk.name = name \text{ and } state \in forcible\_abort(tk)$$

**Definition 6** *Let  $twf$  be a transactional workflow and  $s$  a state of  $twf$ .*

*Then,  $s \in twf\_safe\_states(twf)$  iff*

- $s \in twf\_forcible\_commit(tk)$  or
- $s \in twf\_forcible\_abort(tk)$  or
- $\exists tk'$  such that  $tk' \in twf.tasks$  and  $\langle tk'.name, state' \rangle \in s$ 
  - $state' \notin forcible\_commit(tk') \cup forcible\_abort(tk')$  and
  - $\forall tk, tk \neq tk', \langle tk.name, state \rangle, state \in forcible\_commit(tk) \cap forcible\_abort(tk)$ .

A transition of a TWF corresponds to a transition of one of the tasks in the TWF. A schedule of a TWF is a sequence of transitions that takes the TWF from its initial state to one of its commit or abort states.

A TWF is in its initial state if each task in the TWF is in its initial state. This is defined in definition 7.

**Definition 7** *Let  $twf$  be a transactional workflow. Then,*

$$twf\_initial\_state(twf) = \{ \langle tk.name, tk.initial \rangle \mid tk \in twf.tasks \}$$

The TWF is in a commit state if each task in the TWF is in one of its commit states. Similarly for abort states of a TWF. If a TWF is composed of two tasks  $tk_1$  and  $tk_2$ , and the commit states of  $tk_1$  (respectively,  $tk_2$ ) are  $c_1, c_2$  (respectively,  $c_3, c_4$ ), then the commit states of the TWF are:

$$\left\{ \begin{array}{l} \{ \langle tk_1.name, c_1 \rangle, \langle tk_2.name, c_3 \rangle \}, \{ \langle tk_1.name, c_1 \rangle, \langle tk_2.name, c_4 \rangle \}, \\ \{ \langle tk_1.name, c_2 \rangle, \langle tk_2.name, c_3 \rangle \}, \{ \langle tk_1.name, c_2 \rangle, \langle tk_2.name, c_4 \rangle \} \end{array} \right\}$$

The abort state of a TWF is when at least one of the tasks is in one of its abort states and the other tasks are either in one of their abort states or their initial state.

Definitions 8 and 9 define the set of commit and abort states for a TWF respectively.

**Definition 8** *Let  $twf$  be a TWF. Then,*

$$twf\_commit\_states(twf) = \prod_{tk \in twf.tasks} \{ \langle tk.name, state \rangle \mid state \in tk.commit \}.$$

**Definition 9** *Let  $twf$  be a TWF. Then,*

$$twf\_abort\_states(twf) = \prod_{tk \in twf.tasks} \{ \langle tk.name, state \rangle \mid state \in tk.abort \cup \{ tk.initial \} \} - twf\_initial\_state(twf).$$

A TWF is in an intermediate state if it is not in the initial state, or one of its commit or abort states. This is defined formally in definition 10.

**Definition 10** *Let twf be a transactional workflow.*

*Then,  $s \in twf\_inter\_states(twf)$  iff*

- $s \in \Pi_{tk \in twf.tasks} \{ \langle tk.name, state \rangle \mid state \in States(tk) \}$   
*where  $States(tk) = \{tk.initial\} \cup tk.intermediate \cup tk.commit \cup tk.abort$  and*
- $s \notin \{twf\_initial\_state(twf)\} \cup twf\_commit\_states(twf) \cup twf\_abort\_states(twf)$ .

Using the definition of safe states, definition 12 defines safe schedules. Crucial to the definition are transitions between two states of a TWF which is given in definition 11.

**Definition 11** *Let twf be a transactional workflow.*

*$\langle s, \langle tk.name, action \rangle, s' \rangle \in twf\_transitions(twf)$  iff*

- $s$  and  $s'$  are states of twf, and
- $\langle state, action, state' \rangle \in tk.transitions$ , and
- $\langle tk.name, state \rangle \in s$ , and
- $\langle tk.name, state' \rangle \in s'$ , and
- $\forall tk' \text{ such that } tk' \neq tk, tk' \in twf.tasks, (\langle tk'.name, state \rangle \in s \Leftrightarrow \langle tk'.name, state \rangle \in s')$ .

That is, a transition is when one of the tasks in the TWF makes a transition.

**Definition 12** *Let twf be a transactional workflow.*

*Then,  $\langle s_0, act_1, s_1, \dots, act_n, s_n \rangle \in twf\_sched(twf)$  iff*

- $s_n \in twf\_commit\_states(twf) \cup twf\_abort\_states(twf)$ ,
- $s_0 = twf\_initial\_state(twf)$ ,
- $\forall i, 0 \leq i < n$ 
  - $\langle s_i, act_{i+1}, s_{i+1} \rangle \in twf\_transitions(twf)$ ,
  - $s_i \notin twf\_illegal$ ,
  - $s_i \in twf\_safe\_states(twf)$ ,
  - $\forall s, \langle s_i, act_{i+1}, s \rangle \in twf\_transitions(twf), s \in twf\_safe\_states(twf)$ .

The last condition says that all possible states of the TWF following an action must be safe. Thus the TWF will always be able to terminate in one of its commit or abort states.

A TWF is correct if the TWF system can execute the TWF and guarantee that it will terminate in one of its commit or abort states, and if no failures occur, that the TWF commits – that is, there exists a schedule from its initial state to one of its commit states. Correctness is defined in definition 13.

**Definition 13** *Let twf be a transactional workflow. If  $\exists \langle s_0, act_1, s_1, \dots, act_n, s_n \rangle \in twf\_sched(twf)$  such that  $s_n \in twf\_commit\_states(twf)$  then the twf is correct.*

### 4.3 Mapping a Transactional Workflow to a Task

To support nesting of TWFs, we need to find a mapping from a TWF to a task – that is, we need to find a single finite state machine to represent the TWF. In this section, we define this mapping.

The states of the task representing a TWF are the Cartesian product of the states of all the tasks in the TWF. There may be a large number of states but many of them are unimportant states which we can later remove. Thus, any implementation of a TWF system need not generate all possible states. The mapping is defined in definition 14. In section 4.4 we describe how to determine the visible (important) states of a task and how we can represent a set of states by a single state (merging task states). We use the notation *state\_to\_string* (respectively, *action\_to\_string*) to convert a state (respectively, action) of a TWF to its string representation. These functions are injective.

**Definition 14** *Let  $twf$  be a transactional workflow. We define the mapping  $M : twf\_t \rightarrow task\_t$ .*

- $M(twf).name = twf.name;$
- $M(twf).initial = state\_to\_string(twf\_initial\_state(twf));$
- $M(twf).intermediate = \{state\_to\_string(s) \mid s \in twf\_inter\_states(twf)\};$
- $M(twf).commit = \{state\_to\_string(s) \mid s \in twf\_commit\_states(twf)\};$
- $M(twf).abort = \{state\_to\_string(s) \mid s \in twf\_abort\_states(twf)\};$
- $M(twf).transitions = \left\{ \begin{array}{l} (state\_to\_string(s'), action\_to\_string(act), state\_to\_string(s)) \mid \\ \exists \langle s_0, act_1, \dots, s', act, s, \dots, act_n, s_n \rangle \in twf\_sched(twf) \end{array} \right\};$

It is elementary to show that a state in a TWF is forcible to one of its commit states if and only if the corresponding state in the task representing the TWF is forcible to its commit state. A similar result holds for forcibility to abort states.

### 4.4 Visible States and Merging States

We have shown how to represent a TWF as a task (finite state machine). However it is clear that there may be many states when a TWF is composed of a large number of tasks. What is required is a method to reduce the number of states of a task. This mapping (task simplification mapping) is applicable to *any* task and not just to tasks that represent TWFs.

Some states of a task are useful while others are not useful to a TWF system managing the execution of the task in some TWF. We say that the useful states are the visible states of a TWF.

A transition consists of an old state, an action, and a new state. The new state of a transition is a visible state of the task if the new state becomes forcible to commit or forcible to abort. That is, if the old state is not forcible to commit (respectively, abort), and the new state is forcible to commit (respectively, abort), then the new state is a visible state.

A TWF system finds these states useful in managing the task's execution in a TWF. In a distributed transaction, a sub-transaction becomes forcible to commit when it enters its *prepare* state and we know the importance of the state, since without it, it is impossible to support distributed transactions. The initial state, the commit and abort states are also visible states since they are vital to any TWF system.

Definition 15 formally defines visible states.

**Definition 15** *Let  $tk$  be a task and  $S$  be a subset of the states of  $tk$ .*

*$s \in visible\_states(tk, S)$  iff*

- $s \in \{tk.initial\} \cup tk.commit \cup tk.abort \cup S$  or

- $\exists(s', act, s) \in tk.transitions$  such that either
  - $s' \notin forcible\_commit(tk)$  and  $s \in forcible\_commit(tk)$  or
  - $s' \notin forcible\_abort(tk)$  and  $s \in forcible\_abort(tk)$ .

Notice that a task can explicitly specify a set of visible states via  $S$ . This enables the model to represent dependencies between arbitrary states of tasks in a TWF. This is required for the model to represent data flow between tasks. For example, the *ready* state in figure 7 is not a visible state. However, the task may make output available when it reaches this states. Thus, the *ready* should be a visible state since a TWF system needs to know when the task has entered the *ready* state.

A transition exists between the visible states  $vs$  and  $vs'$  of a task if there is a sequence of actions which can take the task from state  $vs$  to  $vs'$ .

We can further reduce the number of states by merging some of the states of a task into a single state – that is, using a single state to represent a set of states – but we need to first define equivalence classes for task states. Informally, they are:

1. the task's initial state;
2. the task's commit states;
3. the task's abort states;
4. intermediate states that are not forcible to commit and not forcible to abort;
5. intermediate states that are not forcible to commit but forcible to abort;
6. intermediate states that are forcible to commit but not forcible to abort;
7. intermediate states that are forcible to commit and forcible to abort.

Definition 16 formally defines equivalence.

**Definition 16** *Let  $tk$  be a task and  $s, s'$  be states of the task  $tk$  – that is*

$$\{s, s'\} \subseteq \{tk.initial\} \cup tk.intermediate \cup tk.commit \cup tk.abort.$$

*Then,  $s \equiv s'$  iff*

- $(s = s' = tk.initial)$  and
- $(s \in tk.commit) \Leftrightarrow (s' \in tk.commit)$  and
- $(s \in tk.abort) \Leftrightarrow (s' \in tk.abort)$  and
- $(s \in forcible\_commit(tk) \Leftrightarrow s' \in forcible\_commit(tk))$  and
- $(s \in forcible\_abort(tk) \Leftrightarrow s' \in forcible\_abort(tk))$ .

We can then simplify a task further by having a single state to represent the set of state from the same equivalence class. For example, we can represent all the commit states as a single state and all states that are forcible to commit and forcible to abort as a single state. A transition exists between a state  $s_{equiv}$  representing one equivalence class of states to another state  $s'_{equiv}$  representing another equivalence class of states if there is a transition from each state in  $s_{equiv}$  to a state in  $s'_{equiv}$ .

For nested TWF, a TWF may be a task in a larger TWF. We first use the mapping  $M$  to derive the task that represents the inner TWF. Then, we find the visible states of the task and then merge the states. Thus, the difficulty of finding schedules for the larger TWF is reduced.

In general, each task is forcible to abort initially and may become forcible to commit during the execution (intermediate forcible to commit state). Thus, in most cases, each task, even tasks formed from several levels of nesting, will only contain one or two visible intermediate states. Thus, the notion of visible states and merging states reduces the complexity of tasks that are formed from several levels of nesting.

## 5 Modelling TWF Constructs

In this section, we show how we model the constructs ordering, contingency, alternative, conditional and iteration.

### 5.1 Ordering

Ordering constraints are represented by the illegal states of a TWF. If there is an edge from task  $tk$ 's visible state  $vs$  to task  $tk'$ 's visible state  $vs'$  (see figure 9), then

$$\{s \mid s \in twf\_state\_t, \langle tk'.name, v' \rangle \in s, v' > vs', \langle tk.name, v \rangle \in s, v < vs\}$$

is the set of illegal states that represent the ordering constraint. Recall that the task's states and transitions form a directed acyclic graph and define a partial order. Thus,  $v < vs$  and  $v' > vs'$  are well defined.



Figure 9: Ordering Constraint

For example, an ordering which specifies a task  $tk_i$  must commit before  $tk_j$  starts in a TWF composed of tasks  $\langle tk_1, tk_2, \dots, tk_i, \dots, tk_j, \dots, tk_n \rangle$  is represented by the following set of illegal states:

$$\left\{ \begin{array}{l} s \mid s \in twf\_state\_t, \langle tk_j.name, v' \rangle \in s, v' > tk_j.initial, \\ \langle tk.name, v \rangle \in s, v < Com\_State \in tk_i.commit \end{array} \right\}$$

If a TWF, in its execution, does not enter one of these (illegal) states, then we know that the ordering constraints have been satisfied.

### 5.2 Contingency and Alternatives

A contingency plan is composed of two tasks while an alternative is composed of an arbitrary number of tasks. We take advantage of nesting by representing a contingency plan and an alternative as a single task. In fact, a contingency plan and alternative are special TWFs. The commit states of a TWF are where each task in the TWF is in one of its commit states. As introduced earlier, a contingency plan is composed of a main task and a contingency task. The contingency plan is in the commit state if either the main task or the contingency task is in one of its commit states. An illegal state of the contingency is when the main task is not in its abort state but the contingency is in its commit state – that is, the contingency can not be executed without trying to commit the main task.

An alternative is composed of an arbitrary number of tasks. Its commit state is the set of states where exactly one of the tasks is in one of its commit states. The other tasks are either in their initial state or in one of their abort states. Unlike a contingency, there are no further constraints.

The remaining steps to map a contingency and alternative to a task are similar to mapping a TWF to a task. Representing contingencies and alternative require that a task has multiple commit states.

Mapping contingencies and alternatives uses the features that tasks have multiple commit and abort states. If tasks were restricted to one commit state and one abort state, it would still be possible to map a contingency and a alternative to a task but the mapping would be more complex.

### 5.3 Conditionals

A conditional is composed of a condition and two tasks which we represent as a single task. Conditionals introduce non-determinism since the condition can only be evaluated at run-time to determine which task will be executed. Therefore, when a TWF system schedules a conditional, it needs to take a pessimistic view. For example, if one of the tasks contains a forcible to commit state (e.g. a *prepared* state) and the other does not, then the TWF system can not depend on the existence of such a state. However, if both tasks of the conditional contain a forcible to commit state then the task representing the conditional will contain a state that is forcible to its commit state.

If the two tasks of a conditional both contain an intermediate state that is in the same equivalence class, then the conditional task has an intermediate state which represents the two states and the state is in the same equivalence class. The commit (respectively, abort) state represents the states when one of the tasks is in the commit (respectively, abort) state while the other is in the initial state. Finally, a transition between two states of a conditional task exists if there is a sequence of action which takes one of the two tasks in the conditional from one state to the other.

### 5.4 Iteration

Iterations contain a condition and a task. They are a more general structure than conditionals since a conditional can be expressed using an iteration.

We model the iteration as a single task. An iteration specifies that while a condition is satisfied, the task be repeatedly executed. There are two cases. If the task in the iteration does not contain a *prepare* state (a forcible to commit state just before the commit state) then the task representing the iteration contains three states. They are an initial state, a commit state and a abort state. Otherwise, if the task in the iteration does support a *prepare* state, then there are four states in the task which represents the iteration. They are an initial state, a prepare state, a commit state and an abort state. If the task in the iteration is forcible to one of its commit states from its initial state, then the iteration task is forcible from its initial state to its commit state. Also, the iteration task is compensatable if the task within the iteration is compensatable.

## 6 Conclusion and Future Work

This paper presents a model for TWFs that are composed of a set of tasks. The model can represent ordering, contingencies, alternatives, iterations and conditionals. Using this model, a TWF system can extract the properties – forcibility and compensatability – of the task and reason about the TWF to determine its correctness and find a schedule such that the TWF is guaranteed to terminate in one of its commit or abort states even if failures occur during the execution. We are currently implementing a prototype based on the model presented in this paper. The prototype will be able to map a TWF to a task and simplify a task’s finite state machine by deriving the visible states and merging states of tasks. Furthermore, it will be able to map the TWF constructs – ordering, contingency, alternative, conditional and iteration – to the model. In the future, we will be using the model to derive necessary and sufficient conditions for correct schedules. The results will reduce the difficulty of determining the correctness of a TWF and finding schedules.

Also, the model currently only deals with the management of failures for TWFs. We are now extending this model so that a TWF system can manage concurrent executions of TWFs.

## References

- [AAE<sup>+</sup>95] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Guenthoer, and C. Mohan. Advanced transaction models in workflow contexts. Research Report RJ9970, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA, July 1995.
- [BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [Dig94] Digital. *DEC DB Integrator Handbook*. Digital, Maynard, Massachusetts, 1.0 edition, January 1994.
- [Dou92] Dale Dougherty, editor. *Power Programming with RPC*. O'Reilly & Associates, Inc., 1992.
- [Elm92] A. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.
- [GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [Lab91] UNIX System Laboratories. *TUXEDO ETP System Release 4.2 Product Overview*, 1991.
- [Lin93] *LinkWorks - User Guide*, 1993.
- [LKS91] E. Levy, H. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In J. Clifford and R. King, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 20, pages 88–97, 1991.
- [LMWF93] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1993.
- [LR95] F. Leymann and D. Roller. Business process management with flowmark, 1995. Accessible via: <http://www.software.ibm.com/workgroup/flowmark/exmn0b65.htm>.
- [MHG<sup>+</sup>92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornik, and M. Brodie. Distributed object management. *International Journal of Intelligent and Cooperative Information Systems*, 1(1), March 1992.
- [RKT<sup>+</sup>95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch, and P. Muth. Towards a cooperative transaction model - the cooperative activity model -. In *Proceedings of the 21th VLDB Conference*, pages 194–205. VLDB, September 1995.
- [RS94] 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, 1994.
- [SHM94] J. Shirley, W Hu, and D. Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc, 1994.
- [Tra92] Transarc. Distributed transaction processing with Encina and the OSF DCE, September 1992. White paper.

- [YW93] A. Sheth Y. Breibart, A. Deacon, H.-J. Schek and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *SIGMOD Records*, 22(3), 1993.