

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.
A Model for Transactional Workflows* 

Dean Kuo      Michael Lawley      Chengfei Liu      Maria Orlowska

CRC for Distributed Systems Technology
Level 7, Gehrmann Laboratories
The University of Queensland, Qld 4072 Australia
Email: {d.kuo, m.lawley, c.liu, m.orlowska}@dstc.edu.au

Abstract
In this paper we present a model for representing Transactional Workflows (TWFs) involving sequencing, parallel, alternative, conditionals, and iteration. It allows us to reason about the correctness of a TWF, and to generate and execute TWF schedules.

Keywords Transactions, Workflow, Model, Schedules, Failure Atomicity, Integration.

1 Introduction
Many organisations have systems running on different hardware and under different operating systems which are inter-operating via remote procedure calls (RPCs) [3, 15], or simply through manual business processes. This inter-operation gives rise to the complex problem of maintaining consistency of the whole system in the presence of failures and concurrent users.

The concept of Transactions [1, 6, 11] allows an application programmer to write applications without needing to deal with maintaining consistency in the presence of failures and concurrent users since transactions provide the well known ACID (Atomicity, Consistency, Isolation and Durability) properties. That is, a transaction either executes in its entirety or not at all, it takes the state of the resource (database) from one consistent state to another, the execution is not affected by other transactions that may be executing concurrently, and if the transaction commits, the effects of the transaction become permanent. These properties are managed by a transaction processing system. However, in a distributed environment, Transactions are often too restrictive since each system needs to support the standard distributed commit protocol, two-phase commit (2PC). It is only recently that commercial DBMSs have fully supported 2PC, so legacy systems and non-traditional databases are unable to take advantage of this technology. Transactions, however, do not come for free as transactions lock shared resources which blocks other transactions from accessing them. Also, the 2PC protocol is blocking if the co-ordinator of the transaction crashes. For many applications the level of consistency may be safely relaxed to provide better behaviour when failures occur. This is especially important when maintaining local autonomy [12] and dealing with long lived [8] transactions. Thus, transaction processing monitors such as Encina¹ [10] and Tuxedo² [9] are often too restrictive for systems in a real-world distributed heterogeneous environment.

Like Transactions, Transactional Workflows (TWFs) provide a mechanism for an application programmer to write applications without the need to deal explicitly with failures. Unlike Transactions, TWFs allow the application programmer to define the level of consistency required by the application and there is no assumption that each system component need to support the 2PC protocol. Instead they allow a controlled relaxation of the ACID properties of Transactions. Essentially, the TWFs dealt with in this paper allow a user-defined notion of failure atomicity and our future work will allow a user-defined notion of execution atomicity.

TWFs should be distinguished from workflows [7] and commercial workflow products such as LinkWorks [10] since the latter do not provide any Transactional behaviour. Also, the activities managed by commercial workflow products tend to be human oriented, such as creating and updating documents. The workflow product simply provides a mechanism to pass and track the movements of objects around an organisation.

One application for TWFs is in the integration of autonomous database management systems. A multibase database system (MDBMS) attempts to provide a solution by presenting a virtual database, which can be queried and updated as if it were

¹ Encina is a registered trademark
² Tuxedo is a registered trademark
a single real database. This virtual database, therefore, should support transactions. There are numerous commercial MDBMS already available such as DBI [2] from Digital but they, like Encina and Tuxedo, can only support transactions if each DBMS supports 2PC. Most legacy systems, however, do not support 2PC and as a result, the commercial MDBMS are unable to support transactions. It is well known that unless all but one of the DBMS support 2PC then it is impossible to support transactions in a MDBMS.

For financial reasons, replacing the legacy DBMS is impossible to support transactions in a MDBMS. One of the DBMS supports 2PC then it is well known that unless all but one of the DBMS support 2PC then it is impossible to support transactions in a MDBMS.

For financial reasons, replacing the legacy DBMS with DBMS that support 2PC is often not feasible.

TWFs provide a solution to maintaining the consistency of the MDBMS without the strong restriction that each DBMS has to support 2PC by allowing the application programmer to define the level of consistency required by the application. Also, many MDBMS applications need to access data which is only available through an existing application program’s interface (using screen scraping techniques) or data that is stored in non-traditional database systems such as spreadsheets. TWFs provide a mechanism which allows for the integration of legacy DBMSs and non-traditional database systems while providing a level of consistency based on a relatively high-level specification.

In this paper we present a model for representing Transactional Workflows involving sequencing, conditionals, and iteration. It allows us to reason about the correctness of a TWF, and to generate and execute TWF schedules. Our work is more general than the transaction models presented in [5, 13, 14] which do not support conditional and iteration constructs. In addition, in our work we allow a TWF to be a task in another TWF – that is, we allow nesting. The difference between our research and related research is that we are providing transactional behaviour to workflows while related works are supporting workflows consisting of tasks that are transactions. The model presented in [13] is designed for multirelational transactions (the integration of traditional databases) but not for non-traditional database systems such as spreadsheets and legacy applications. Our model is, again, more general as it can model both accesses to traditional and non-traditional databases.

There are numerous extended transaction models proposed to overcome the limitations of transactions by relaxing the ACID properties. Many of the these models are presented in [4]. However, these models still require that the components in the distributed environment support the two-phase commit protocol and do not allow human oriented tasks or access to non-traditional databases.

The rest of this paper is organised as follows. In Section 2, we define TWFs in terms of primitive and compound tasks. In Section 3, we investi-

![Figure 1: State Transition Diagram for Tasks](image)

Figure 1: State Transition Diagram for Tasks
gate the generation and execution of schedules for a TWF. Section 4 gives an example of a TWF and shows a schedule for the example TWF. Conclusions and future work are presented in Section 5.

2 Transactional Workflows

A TWF is either a primitive task, or a compound task consisting of TWFs composed by, among other things, sequencing, conditionals, iteration, alternatives and contingencies.

2.1 Primitive Tasks

A primitive task can be a function, application, service or an activity carried out by some process. Essentially, it can do as much or as little as you want. It is not the job of the TWF system to deal with the actual semantics of tasks. Rather, the TWF system simply manages the order of execution of the various stages of the tasks in a TWF and assumes the programmer has defined the TWF such that it does something meaningful. Hence, there is no need or reason to supply pre/post-conditions or any other semantic information describing what a task does.

A primitive task is described by the following properties:

- **Commit**: {0PC, 1PC, 2PC}
- **Compensatable**: Boolean
- **Forcible**: Boolean

plus a set of inputs and outputs.

One can associate a set of visible states with the commit property of a task. Figure 1 shows the full set of visible states a primitive task may have and the transitions which may occur between these states.

In transaction management, a typical transaction executes a sequence of actions and then requests a commit or abort. The commit may fail in which case the transaction is aborted. Some transaction managers support the distributed commit protocol 2PC such that the transaction can be part of a distributed transaction – that is, the transaction can enter a prepared state and then, if the coordinator of the distributed transaction decides to commit, the transaction is guaranteed to commit.
A transaction manager supports the 1PC protocol if one can explicitly request a commit or abort after the actions in the transaction have been executed. Similar to transactions, some tasks support 2PC while others 1PC. However, some tasks may not support 1PC or 2PC as there is no commit phase—that is, the task either executes or it does not. We say that these tasks support the 0PC (zero phase commit) protocol. An example of a 0PC task is airline reservation since a request to reserve a seat on a flight either commits (books a seat on the flight) or aborts (fails to book a seat) and there is no explicit command to commit or abort the task. If one of the tasks in the TWF accesses data via screen scraping of a legacy application, then the task would be a 0PC task.

Some tasks are compensatable. For example, an airline reservation is compensatable by cancelling the reservation. If a task is compensatable, then there is a legal transition from the commit state to the abort state through the compensation action.

Other tasks may be forcible—that is, the system will guarantee that the task will eventually succeed. The state transition diagram for forcible tasks is deterministic since the only transitions to the abort state is by the abort action. Withdrawals from accounts that do not allow over-draft are not forcible while accounts that do allow over-draft are forcible, for example.

A task’s interface is like a C function prototype. It contains a name for the task and describes the inputs and outputs variables. If a task’s output is the input to another task, then there is an order dependency. Thus, the data-flow in a TWF maybe determined by examining the inputs and outputs of the tasks in a TWF.

There are no restrictions on the actual implementation of the task as the interface does not make any assumptions and some of these tasks may even be human oriented tasks.

2.2 Visible States

Each task displays a number of visible states and the list of all possible visible states are listed below.

- Initial
- Ready
- Prepared
- Committed
- Aborted

The Initial state models an inactive task—that is, no request to execute the task has occurred while a Ready state reflects that all but the commit phase of the task has been executed. For tasks that support the 2PC protocol, the visible state prepare reflects the prepared state of the task. Finally, the state commit and abort respectively reflect the commit and abort states. A task’s visible state is therefore dependent on the task’s commit property. If a task supports the 2PC protocol, then it would display all visible states while a task which only support 1PC would not display the prepared state but would display all other states. Finally, a 0PC task would only have initial, commit and abort as its visible states. Figures 2 to 4 show the state transition diagrams for these three types of tasks and Figure 5 shows the additional transition for compensatable tasks.

2.3 Compound Tasks

A compound task is a labeled directed graph where each node is a primitive task or a compound task.

In transaction management, each transaction requires that all of its tasks be executed successfully (commit) or not at all. TWFs allow the application programmer to relax the failure atomicity by allowing them to specify which tasks are critical to
Figure 5: State Transition for Compensatable Tasks

A TWF – that is, if a TWF commits, the critical tasks must all have committed while others may have aborted or not executed at all (the initial state). Also, one can specify which tasks need not be undone, even if a TWF aborts. We say that the components supporting TWFs are correct if they guarantee that all critical tasks are committed if the TWF commits; otherwise, when a TWF aborts, then only the effects of those tasks that do not require undo may persist. Thus, the label associated with each node in the TWF contains the two properties listed below.

- **Critical**: Boolean
- **Requiring Undo**: Boolean

The set of all labels in the TWF defines the level of failure atomicity that is required by the TWF. In [13], they only have the notion of critical tasks in a TWF. Our model, therefore, allows the application programmer more flexibility in specifying the level of consistency.

Note that these properties are specified in the context of a particular TWF rather than being associated with the task itself. Hence a task may have a compensation action defined, but for a particular TWF the task doesn’t need to be undone if the TWF execution aborts.

For TWFs to be useful, they must support a number of constructs such as alternative tasks, contingency plans, specifying the dependencies between tasks, conditional and loop constructs. These are described below.

**Order Dependencies** The edges in the directed graph specify a partial order for the execution of the compound task. Any execution must satisfy these ordering constraints.

A TWF may specify that a task \( tk_i \) be executed before \( tk_j \). This does not mean that \( tk_i \) must be committed before \( tk_j \) but \( tk_i \) must reach its ready state before \( tk_j \) executes. A schedule may execute the body of \( tk_i \), execute the body of \( tk_j \), commit \( tk_j \), and then commit \( tk_i \).

**Alternative Task** An alternative task is composed of a set of tasks, some of which may be primitive tasks while others are compound tasks. Alternatives specify a number of methods to achieving a goal. For example, there are two alternatives to purchasing a domestic airline ticket and they are either through the QANTAS’ or Ansett’s airline reservation systems. QANTAS’ and Ansett’s airline reservation system may have different commit, compensatable and forcible properties. The TWF management system can choose one of the alternatives such that it can guarantee, in the presence of failures, that the execution satisfies the user’s requirements.

Alternatives can also be preferred so that one can specify that which is the preferred reservation system.

**Contingency Plan** A contingency plan for a task is either a primitive task or a compound task. It allows the application programmer to specify an alternative if the task fails in its execution. For example, if the QANTAS’ reservation system is down or a request for a seat on a flight is rejected, then a contingency plan may be to try Ansett’s reservation system.

There is a subtle difference between a contingency plan and an alternative task. A contingency plan can not be executed until the task to the contingency plan has failed in its execution. Thus, there is a constraint on the TWF system when it processes a contingency. There are no such constraints when the TWF system processes an alternative task.

**Conditionals** In our TWF, we allow conditional constructs like if statements. They take the form of

```plaintext
if ( condition )
    task1();
else
    task2();
endif;
```

where `task1()` and `task2()` are TWFs.

For the conditional construct, the TWF manager and executor must check the condition so that they know which task to execute.

The case statement can also be supported in the TWF since it is a trivial generalisation of the if statement.

**Loops** Our TWF also supports loops but do not allow arbitrary loops (see figures 6 and 7). Loops must be of the form:

```plaintext
while ( condition )
    task();
```

where `task()` is a TWF.

3 Model

Given a TWF we will want to execute it. This means invoking the tasks in the TWF subject to
the ordering and other constraints specified by the TWF. This requires determining a schedule, which consists of a sequence of TWF state transitions.

In this section, we present a model for TWFs. The primary function of this model is to capture the state and all the properties of a TWF, thus enabling us to reason about them.

3.1 Composition Constructs

In our model, alternative tasks, contingency plans, loops and conditional are all represented as a compound task. Since a compound task may be a task in another compound task, it must also have the same set of property as a primitive task. We show, below, how these values are derived.

**Alternative Task** An alternative task is composed of a number of tasks and each specifies an alternative. The properties of the alternative task is, therefore, the properties of task that is executed. For example, if an alternative task is composed of two tasks and the first task is compensatable, not forcible and supports the 2PC protocol while the other is not compensatable, forcible and supports the 1PC protocol, then the alternative task’s properties is either the properties of the first task or the second task.

**Contingency Plans** We model a task $tk$ and its contingency plan as a compound task. The commit property of the compound task is the weakest commit property of task $tk$ and the contingency plan$^3$ where $0PC < 1PC < 2PC$. The compound task is forcible if either the task $tk$ or its contingency is forcible. If the task $tk$ is forcible, it is actually senseless to have a contingency since the task is guaranteed to commit. If the contingency is forcible, then we know that the compound task will commit eventually.

The compound task is compensatable if both the task $tk$ and its contingency are compensatable since one can not determine which will be executed until execution time.

**Conditionals** Associated with a conditional is two tasks. We model the conditional and the two tasks as a compound task. Its commit property is the weakest commit property of the two tasks. Also, the compound task is forcible if both tasks are forcible. Similarly for the compensatable property.

**Loops** A loop contains a condition and a subtask, which may be a compound task (see figure 6). We model the loop as a compound task and its property is dependent on the subtask’s property. The semantics of a loop is such that it commits if every invocation of the subtask in the loop commits, and aborts if all invocations abort. Therefore, unless the subtask supports 2PC or is forcible or is compensatable, then there are no possible schedules.

If we want to relax the condition that all invocations of the subtask in the loop must commit, we can do this by encapsulating the subtask within a compound task which consists of the original subtask plus a contingency plan. With appropriate use of counters and conditions, this combination can be used to produce a loop where the subtask commits only at least $n$ times after $m$ iterations rather than exactly $m$ times.

3.2 Compound Task State

The state of a compound task execution can be modelled by a vector consisting of the Cartesian product of the current state of each of the tasks comprising the compound task.

If the compound task is composed of $n$ nodes, then the state of the compound task is

$$\langle \text{State}(tk_1), \ldots, \text{State}(tk_n) \rangle$$

where $\text{State}(tk_i)$ is the state of the task in the $ith$ node in the compound task (according to some arbitrary but fixed order).

A Transaction is a special case of a TWF since a Transaction commits if and only if all the tasks have successfully executed. Thus, for Transactions, the commit termination state is:

$$\langle C, C, \ldots, C \rangle$$

and the abort termination state is:

$^3$The contingency plan is a primitive or compound task.
where \(C\) is the commit state and \(A\) is the abort state.

Any compound task may be a (compound) task in another compound task. Therefore a compound task has visible states which must be a subset of the possible visible states for a primitive task. For example, a compound task is in the initial state if all of its subtasks are in the initial state. For the other visible states, the mapping from the state of a TWF to a visible state is dependent on the properties of its subtasks as well as the labels on the task nodes. However, it is only a partial mapping since many compound task states are only intermediate states corresponding, for example, to the state of a primitive task while it is making the transition from the initial state to the ready state.

This is a powerful feature of our model since we can now abstract parts of a TWF and treat them as a single (compound) task.

### 3.3 Termination State

Each compound task defines the level of failure atomicity it requires. This is mapped to a set of acceptable termination states — that is, if the compound task terminates in one of these states, then the execution of the compound task is correct. A schedule should take a compound task from its initial state to one of the acceptable end states, even if failures occur. There are two types of termination states, commit and abort. One of the constraints on the schedule generated by the scheduler is that if no failures occur — all requests to execute tasks and to commit tasks succeed — then the TWF will end in one of the commit termination states.

A compound task composed of \(n\) nodes is in the commit termination state if all nodes which are labeled as critical are in the commit state. The remaining nodes must be in the abort or initial state. Thus, in the termination state, none of the nodes in the TWF are active. The compound task is in the abort termination state if all nodes are either in the initial or abort state or, if a node is labeled as not requiring \(amde\), then it may be in the commit state. Again, none of the nodes in the compound task are active.

### 3.4 Legal Transitions and States

The legal states of a compound task capture the constraints such as the execution order of the tasks. The directed graph which represents a compound task shows the ordering constraints. Let \(I, R, P, C\), \(A\) respectively denote the Initial, Ready, Prepared, Commit, Abort states. Suppose that there is a directed edge from node \(n_i\) to node \(n_j\), then the state \(\langle \cdots, State(n_i) = I, \cdots, State(n_j) = R, \cdots \rangle\) is illegal since the task in node \(j\) has executed before the task in node \(i\). However, the state \(\langle \cdots, State(n_i) = R, \cdots, State(n_j) = C, \cdots \rangle\) is not illegal since the order only specifies that the task in node \(n_i\) must execute before the task in node \(n_j\) but does not specify their commit order.

A legal transition for a compound task occurs when one of the task makes a transition from one visible state to another and the new state is a legal state of the compound task. The legal transitions of a task were described in section 2.

### 4 Example

We present an example in this section to highlight the way the model can represent a TWF.

Consider the following TWF for booking an airline ticket, hotel and car for a business or holiday trip. Suppose that there are two reservation systems available, one provided by QANTAS and the other by Ansett Australia, thus the task of reserving a seat using QANTAS’ system has a contingency plan — that is, if the request to make a reservation fails using the QANTAS reservation systems due to the system being down or the flights being full, then one can try to book on the other airline’s reservation system. Another task in the TWF is to book some accommodation at the destination. Since the hotel may be full, we may wish to try a fixed number of times to make a booking before we abort the TWF. This can be implemented using the loop construct. In our model, we represent the loop condition and the hotel reservation task as a compound task and label the node for the compound task as critical since we do not want to make the trip if we can’t find a place to sleep. There is a data flow from the airline reservation task to the hotel reservation task since a hotel reservation is required for any stopovers. Another node in the TWF is to reserve a rental car. There is no penalty for people who reserve a car and not show up to rent it. Thus, there is no need for undoing the car reservation if the TWF aborts. Similar to the hotel reservation, there is a data flow from the airline reservation node in the TWF to the car reservation node.

In our TWF, the node \(tk_1\) represents the airline reservation task, node \(tk_2\) represents the compound task of hotel reservation and node \(tk_3\) for the car reservation task. Their execution dependencies are shown in figure 8.

Suppose that all but the car reservation are critical. Then the acceptable commit termination states are:

\[
State(tk_1) = C; State(tk_3) = C;
\]
and the **abort** termination states are:

\[ State(tk_1) \in \{A, I\}; State(tk_2) \in \{A, I\}; \]
\[ State(tk_3) \in \{C, A, I\}. \]

where \( I \) is the initial state, \( C \) is the commit state and \( A \) the abort state.

From the ordering constraints, the illegal intermediate states for this TWF are:

- \( State(tk_1) = Initial \) and \( \forall i, 2 \leq i \leq 3, State(tk_i) \in \{R, C, A\} \)

The scheduler would examine the legal termination states of the TWF and determine that the TWF can reach a legal state with the following schedule:

Execute the airline reservation first, and then try to reserve a room at the hotel a number of times. If they all fail then the TWF is aborted and we cancel the airline reservation; otherwise the TWF commits. The car reservation node may be executed in parallel to the hotel reservation task. The TWF commits independent on the final state of the car reservation since the node is labeled as not critical and does not require undo.

5 Conclusion and Future Work

In this paper, we have given an overview of TWFs and a model to represent TWF. This model is a powerful tool since the TWF scheduler can reason and generate a schedule for a TWF.

The model can model the features of a TWF – that is, the level of failure atomicity that is required, conditional and loop constructs, the different commit protocols of tasks in the TWF, tasks being compensatable and forceable – simply and concisely.

We are currently using the model to implement a scheduler. Also, we are working on the design and implementation of the other components to support TWF.

References


