ABSTRACT
Today computing environments are often composed of heterogeneous subsystems like mainframes, workstations and personal computers. The various subsystems typically use database systems with different transaction models. For such subsystems to cooperate smoothly, it is necessary to have a precise understanding of each other's transaction models. In particular, a TP-monitor trying to coordinate transactions of subsystems needs precise models of the transactions offered by these underlying subsystems.

This paper introduces nondeterministic, atomic computing primitives which are used to construct A-nets for describing transaction models precisely. A-nets and the computing primitives can be applied recursively to describe transactions at arbitrary levels of refinement.

The techniques are demonstrated by describing a number of well-known and widely used transaction models, in particular ACID-transactions, distributed transactions with 2-phase commit, Sagas, and LU6.2 transactions.

1 Non-Deterministic Computing Primitives

In this paper I will introduce a model of non-deterministic computation based on atomic primitives. Changing the level of abstraction and looking at the fine structure of our atomic computing step we will apply the computation model recursively.

This computation model seems suitable to describe activities of our daily life: Going to lunch to a particular restaurant, we usually don't know what we will eat, our choice being the non-deterministic result of this activity. If we ask our travel agent to book a flight, we do not determine the exact airline or seat number on the plane, leaving certain aspects of this activity to circumstances beyond our control, which renders these activities for all practical purposes non-deterministic at least to some degree. Yet, when we "go to lunch" or "book a flight", we start our activity with a clear purpose in mind and accept the inherent non-determinism in the result.

The formal abstractions to model such activities in computers are usually called "transactions". Our computing model will hopefully generalize and improve our understanding of what transactions are or ought to be.

To introduce the formal model, let us try to conceive the simplest possible non-deterministic computing element.
Some thought yields the insight that we can further simplify our model by identifying $R_2=S$.

![Diagram](image)

**Figure 1-4**

An interesting question is, how we know, whether we started $t$ resulting in $A$ or never started $t$ at all. We defer this question until later.

We observe, that our "minimal" non-deterministic computing primitive is equivalent to an ACID-Transaction [3]. We leave it to the reader to identify the ACID-properties atomicity, consistency, isolation and durability for our minimal, non-deterministic computing primitive. Durability is important, meaning that a state does not change unless a computation is started somehow, typically triggered by an outside event like a command "START $t$" or "RUN $t$".

A truly ACID transaction can only have one interaction with a user: the user says "run transaction $t$" and gets a non-deterministic result back. Usually ACID transactions as they are described in the literature have a non-atomic internal structure exposed to the user who runs the transaction. We call this user the owner $O$ of the transaction. $O$ typically sees an additional internal state, which we call $W$ for "working", which allows him to issue further commands like a series of SQL statements. Therefore from $O$’s point of view an ACID transaction looks as follows:

![Diagram](image)

**Figure 1-5**

The meaning of the states is:
- $S$ external starting state
- $A$ external aborted state
- $C$ external committed state
- $W$ internal working state

Note that in this transaction model there are several non-deterministic computation steps and one deterministic step (triggered by command "abort $t$". For $O$ the transaction $t$ is not atomic at all, for another user $U$ it is atomic and non-deterministic of the form:

![Diagram](image)

**Figure 1-6**

Thus atomicity is a relative concept depending on our point of view and on the resolution of our vision. It is surprising to observe that most descriptions of transactions offer the interface:

- `begin-transaction`
- `end-transaction`
- `abort-transaction`
- `issue DDL-statement`
- `issue DML-statement`

A genuine ACID-interface should offer only one command, namely `run-transaction` with two non-deterministic results, "committed" or "aborted".

### 2 REFINED TRANSACTION MODELS

#### 2.1 Some Refinements

For the cooperation between several transactions, e.g., distributed transactions using 2-phase commit, additional states are needed, in particular the "prepared" state $P$. Let us assume that only the database system (TP-Monitor) can see state $P$, normally the owner $O$ cannot see $P$.

From the point of view of TP-Monitor the transaction model then has certain external states visible to the owner, and additional internal states visible to TP-Monitor only:

![Diagram](image)

**Figure 2-1**

The application program or owner $O$ issue the commands in uppercase letters, TP-Monitor those in lower-case letters. In reality, user commands (like "END-TRANSACTION $t$") are translated by TP-Monitor into one or several internal commands like "prepare $t$" to all subtransactions of a distributed transaction and "commit $t$".
Since TP-Monitor sees the internal state P, it can coordinate a distributed transaction and issue "commit t", when it sees that all subtransactions are in state P. Now let us consider a transaction model with an externalized P-state, i.e.:

\[ A \rightarrow \text{ABORT t} \]

\[ S \rightarrow \text{P COMMIT t} \rightarrow C \]

**Figure 2-2**

Now any subtransaction c of a group G of subtransactions making up t may play coordinator, if c follows the rule: if all \( g \in G \) are in state P, then c may broadcast "COMMIT t" to every \( g \in G \). If c sees one subtransaction in state A, it may broadcast ABORT t to everybody else.

Therefore, with the externalized P-state, coordination between transactions need not be done by a TP-Monitor, but can be left to any subtransaction taking the initiative, if it follows the rules of the 2-phase commit-protocol.

An interesting technical question is, how to handle "dangling messages" which arise e.g., if several subtransactions take the initiative and play commit coordinator. The design of a robust protocol must also solve the question of dangling messages.

The following is a set of rules for the 2-phase commit protocol:
1. If c is in state P it must not broadcast "ABORT t"
2. Any \( g \in G \) including TP-Monitor seeing all \( g \in G \) in state P may broadcast "COMMIT t"

**Note:** The TP-Monitor can be seen as a member of G, which does not enter state P until all \( g \in G \) are in state P, thus reserving the ABORT-right for himself.

### 2.2 Dependent Transactions

Some transaction models use additional internal or external states to improve the cooperation between transactions. TP-Monitor sees these states and coordinates the transactions, how they may proceed to achieve a common goal, like to commit together or to abort together.

We consider a finished state F meaning that t has done all its work except commit processing. A typical dependency rule between two transactions \( t_1 \) and \( t_2 \) might be:

"If \( t_2 \) may enter state F only after \( t_1 \)."

With an internal F-state only the TP-Monitor can coordinate \( t_1 \) and \( t_2 \).

With an externalized F-state, \( t_2 \) may follow the protocol:

- If \( t_2 \) sees \( F_1 \) then \( t_2 \) may enter state \( F_2 \)
- If \( t_2 \) sees \( P_1 \) then \( t_2 \) may enter \( F_2 \);
  - enter \( F_2 \);
  - broadcast "commit";
  - enter \( C_2 \);

Dependencies between transactions can be seen as legal combinations of states of the dependent transactions. The protocols must guarantee that only legal state-combinations will be reached. If the states involved in such dependencies are externalized, i.e., visible to other transactions, then an interesting problem is the design of robust distributed democratic coordination protocols. If these states are internal, then somebody who can see them (typically the TP-Monitor) must coordinate the activities.

### 3 Modeling Sagas with A-Nets

#### 3.1 Saga Steps

Sagas [1] are used to model long lasting activities. A single Saga-Step consists of an ACID-Transaction. We assume the simplest model for a Saga-Step:

\[ A \rightarrow \text{RUN} \rightarrow C \]

**Figure 3-1**

We summarize the essential properties of our computing model:
1. The states only change due to explicit commands (internal or external) triggering certain computations.
2. A command triggers a specific computation, so the selection and the start of that computation is assumed to be deterministic (deterministic control).
3. The computation itself is non-deterministic with a finite number of potential result states. These result states may carry additional information, e.g., in result parameters or in a database.

Since sagas are composed of several steps and must be semantically atomic (all steps must be performed or none), [1] introduces compensation steps, which are themselves ACID-transactions of an underlying database system or resource manager RM. We denote the states of the step \( s \) by the index \( s \), the states of the compensation step \( c \) by the index \( c \).

A simple saga step has no compensation, a general saga
step has a compensating transaction c and would have an A-Net representation as follows, in which state U (undo step) is the start state for the compensation:

\[
\begin{align*}
A & \xrightarrow{\text{RUN c}} A_c \\
S & \xrightarrow{\text{RUN c}} C_c \\
\end{align*}
\]

Figure 3-2

Saga descriptions [1] assume that compensations do not fail. If they fail nevertheless, a catastrophic failure has occurred (state cf) and exception handling must be started, usually requiring manual repair of the system. Introducing the state cf and assuming that exception handling eventually results in the success of the compensation, one obtains the A-Net for a general saga-step:

\[
\begin{align*}
A & \xrightarrow{\text{repair}} cf \\
S & \xrightarrow{\text{cf}} C \\
\end{align*}
\]

Figure 3-3

In this model, S and U are initial states (in automata theory terminology) and A, C are final states. Since exception handling should not be visible to the user (except for time delays) we made it an internal state.

If one considers other failures too then the exception handling might also lead from state cf to another of the states with the meaning:

- in state S: the saga step s could be repeated
- in state C: the saga step s was committed
- in state U: the step s was successfully compensated
- in state A: the step s was aborted and can be re-tried

In addition it is useful to know, which saga step had a catastrophic failure, which can be expressed by distinguishing the different cf states by the index s. From now on we will omit the state cf from the general saga steps and use the A-Net representation.

3.2. Linear Composition of Sagas

We can compose saga steps linearly by identifying the commit state of step i with the start state of step i+1, the abort state of step i+1 with the undo state of step i. For the last step in a saga we do not need compensations [1]. This results in the following A-net for linearly composed sagas:

\[
\begin{align*}
A & \xrightarrow{s_1} U \xrightarrow{s_2} \cdots \xrightarrow{s_n} C \\
\end{align*}
\]

Figure 3-5

Note that a composed saga has 3 external states S, A, C just like a simple saga step. Thus a saga either commits or aborts automatically and successfully from the point of view of a saga user. This also means that a single saga step can be refined by replacing it by a linearly composed saga, which leads to hierarchical composition of sagas.

3.3 Hierarchical Composition

We now show hierarchical composition, refining a forward single step, here s2 of Figure 3.5 by a linearly composed subsaga:

\[
\begin{align*}
A & \xrightarrow{s_2} U \\
\end{align*}
\]

Figure 3-6

Splitting up the transaction s2 into several transactions requires the introduction of additional internal states and
of compensating transactions. It is illustrative to think about the reasons why the compensation $c_2$ is kept and why no compensation for $c_3$ is required.

It is clear now that linear and hierarchical refinement and composition of sagas can be applied recursively down to arbitrary levels of detail, using our minimal non-deterministic computing primitive at the lowest level of refinement and resolution.

### 3.4 Saga Monitor

If we accept sagas as a natural generalization of transactions, the question arises: "What is the generalization of a TP-Monitor to a Saga-Monitor?"

In a saga programming environment sagas are designed and constructed. They are handed over, e.g., in the formal representation of A-Nets, to a Saga-Monitor for interpretation. Therefore a Saga-Monitor must perform at least the following essential tasks:

1. Know the sagas and their execution state
2. Know whether a saga-step is a forward or a compensation step
3. Trigger the next forward or compensation step depending on the non-deterministic result of the previous step
4. Make the external states $S$, $C$, $A$ known to the outside, e.g., by raising certain events or sending messages
5. Notify exception handling in case of a catastrophic failure (internal state $c_f$)

Since the saga steps are actually transactions in one or several underlying RM, the Saga-Monitor is primarily a message or event manager for messages between RMs and between RMs and users and a transaction coordinator.

It is clear that the activities of a Saga-Monitor depend heavily on the transaction models offered by the underlying RMs. Here we considered only the simplest case of ACID-transactions with external states S, A, C. Much more can be done with additional states like F and P. The transaction models of the RMs should be described as A-Net's.

Through the standard interface of an RM the user gets little information about the state of his transactions, in particular in case of failures, deadlocks, etc. A Saga-Monitor could do more by interpreting the logs of the RMs and using the information contained therein as a meta-database about the sagas. In addition, a Saga-Monitor must of course keep whatever information it needs about the sagas it runs in its own meta-database. To store this meta-database, the Saga-Monitor can itself use one of the RMs. This yields the following architecture for a Saga-System:

---

Figure 3-7

An analogous architecture can be used for other generalizations of transactions beyond sagas.

### 4 Transaction Model of LU6.2

Here we use the usual LU6.2 terminology [2] and model 4 different abstraction levels of LU6.2 transactions. The ACID-transactions participating in a distributed transaction are called "agents". We use LU6.2 verbs for the commands issued. Uppercase letters are used for external commands which are expected from the outside of our agent in an externalized state. Lowercase letters are used for commands issued from the inside when the agent is in an internal state. Adapting to LU6.2 terminology we use the following abbreviations for states:

- N Not existing yet
- W Working
- F Forgotten, i.e. either aborted or committed previously
- C committed
- P Prepared

#### 4.1 Abstract Model of LU6.2 Transactions

A transaction $t$ run by application App1 of owner O1 looks to another application App2 or another user O2 as follows:

---

Figure 4-1

Note that the interface "Run $t$" is not offered by LU6.2, although the above is an abstract model of an LU6.2 transaction.
4.2 Model of an LU6.2 Agent as seen from the owner-application

![Figure 4-2](image)

Note that LU6.2 knows many more verbs usable by an application, but they do not seem to be relevant to transaction models and syncpoint processing.

Note that A, N, F are states visible to the outside, i.e., to App2 and 02, whereas the additional internal state W is visible to App1 and 01.

4.3 Model of an LU6.2 Agent as seen by an LU6.2 Monitor

Now the external states are those visible by the owner of the agent. Additional internal states are needed and visible by an LU6.2 Monitor, in order to perform commit processing and commit coordination between the various agents who are active on behalf of a distributed LU6.2 transaction.

In general several agents, AG1 ... AG2, are active for an application AP due to several ALLOCATE commands issued by AP. The SYNC_POINT command of AP must be translated into the appropriate command "prepare" for every agent except the last one who receives the "request commit" command. The commit-protocol is an optimization of LU6.2 to minimize the number of message sent within the system. It is obvious from the A-Net of an agent that exactly 2 (message, answer) pairs are required for commit processing per agent.

![Figure 4-3](image)

LU6.2 must know or decide who is the "last" agent in a group of agents. Also LU6.2 must require and keep track of the messages properly.

4.4 Model of an LU6.2 Agent as seen by a Resource Manager

In general LU6.2 will employ the services of underlying resource managers RM or database systems who will actually run the agents on behalf of LU6.2. Therefore, the RM has an even more detailed model of an agent. In this model, the states visible to LU6.2 are external, all commands, so far, are external commands issued by LU6.2 to the RM. The model of an agent can of course look quite different for different RMs, using different internal states. The external states however must all be the same, since LU6.2 probably expects a uniform agent model.

For the purpose of refining our model further, certain assumptions about the underlying RM must be made. For a detailed refinement see [4]. In this refinement very interesting optimization techniques of typical LU6.2 implementations become apparent.

5 Dependencies of Transactions

5.1 Coordination

If an application has a transaction model like in Fig. 5-1

![Figure 5-1](image)

it cannot coordinate several parallel transactions to perform 2-phase commit or to have a so called "strong mutual commit dependency". Still it may inform the TP-Monitor, who hopefully has more detailed control, like an externalized P state, that it wants mutual strong commit dependencies means that application AP accepts only (A1, A2) or (C1, C2) as legal final state combinations of Ag1 and Ag2, but rejects (A1, C2) and (C2, A2) as illegal. This application activity could be modeled with the following A-Net (Fig. 5-2).

The coordinator, be it the application itself or a TP-Monitor, needs a more detailed model of an agent to enforce mutual strong commit dependencies (Fig. 5-3).
5.2 Refinement of COMMIT Step

The commit-processing-step of the application-activity-net must then be refined again and can be described by a more detailed A-Net. For details see [4]. This refinement could be done by the TP-Monitor or the application. Clearly there are big advantages to handle such complex A-Net refinements and executions within the TP-Monitor and not to burden the application with them. The refined A-Net can then be optimized in various ways, e.g. to save messages or to speed up commit processing. Some of these optimizations lead to dangling messages and to the problem of handling them properly.

6 A Monitor for A-Nets

6.1 Duties of Monitor

A-Nets are descriptions of non-deterministic, but sequential programs. If an A-net execution reaches a state, exactly one well defined next computation step will be triggered by external or internal commands. This computation step may, however, have a nondeterministic result reflected through the state reached after the step.

Parallel activities are modeled by forking a child A-Net in a way analogous to forking a UNIX-process. The forking step is a computation step of the parent with a nondeterministic result: Child-Net forked successfully or not. But the execution of the Child-Net then runs asynchronously in parallel with the parent-net.

This forking technique must not be confused with the refinement of a single step by a more detailed A-Net. The latter corresponds to a synchronous procedure call, which causes no parallelism between parent-net and refinement-net, actually the refinement-net becomes part of the parent-net.

When a forked child-net CN reaches an external state, the parent-net PN is notified by a corresponding message from CN to PN. Alternatively PN may keep polling its children for the external states they reached (this is an implementation variant).

The underlying A-Net-Monitor creates A-Net incarnations, schedules and interprets them, handles messages between them much like an operating system does for families of processes.

In addition to operating system services an A-Net-Monitor provides services like synchronization, logging, recovery and commit processing for agents. In principle these services could be performed by the parents on behalf of their children. It is easier and more economical to establish these services once and for all and to delegate them to an A-Net-Monitor.

6.2 Forking and Joining of Children

When a child-net is forked its type is known and its identity is returned to the parent by the A-net-monitor. Thereby the parent can communicate about the external states of its children.

The dependencies between children should be of no concern when they are forked. Join conditions, like strong commit-dependencies or finish-dependencies should be specified at the time of joining. The joining of children is itself a particular computation step in the parent-net with a desired, but non-deterministic result. A typical join variant would specify certain desirable combinations of external states of children, e.g.,

- COMMIT (Ag1, Ag2) or
- FINISH (Ag1, Ag3, Ag4) or
- PREPARE (Ag1, Ag3, Ag5)

The A-Net-Monitor then has to coordinate the child-agents involved in order to reach the desired combination of states or to reach an acceptable combination, like "all agents involved were aborted".

Whether a particular join variant is possible or not depends on details of the A-Nets of the involved agents, in particular on their internal and external states and on the possible state transitions. We investigated several examples in [4]. The join-variant is checked and coordinated by the A-Net-Monitor, the result is then reported to the parent issuing the join-request. Various formal and very useful results on join-processing and commit-processing are presented in [4].
Acknowledgement

This research was triggered by the stimulating environment of DEC/tpwest. The results were clarified and improved by many discussions, in particular with Dieter Gawlick, Johannes Klein, Adel Ghoneimy, Ron Obermarck, Charly Kleissner, Ed Cheng, Alberto Lutgardo, Pat O'Neil, Bob Taylor.

References


