The ACID Model Checker and Code Generator for transaction processing

Anakreon Mentis Panagiotis Katsaros

Department of Informatics, Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
tel.: +30-2310-998532, fax: +30-2310-998419
{anakreon, katsaros}@csd.auth.gr

Abstract

Traditional transaction processing aims in delivering the ACID properties (Atomicity, Consistency, Isolation, Durability), that in our days are often relaxed, due to the need for transaction models that suit modern computing environments and workflow management applications. Typical examples are the requirements of long-running transactions in mobile computing or in the web, as well as the requirements of business-to-business collaborative applications. However, there is lack of tools for automatically verifying correctness of transaction model implementations. This work presents the ACID Model Checker and Code Generator, which plays a vital role in developing correct simulation models for the ACID Sim Tools environment. In essence, our contribution introduces an approach for automatically generating provably correct implementations of transaction management, for the transaction model of interest.

Key Words – transaction & concurrency control, modelling simulation & evaluation techniques, fault tolerance, model-checking

1. Introduction

In network centric information systems, where applications are partitioned into several tiers (e.g. presentation, logic and data), transactions offer the mechanisms needed to reach a mutually-agreed outcome, which will be observed consistently across the transaction participants. Transaction processing is expected to offer the typical or relaxed ACID guarantees (Atomicity, Consistency, Isolation and Durability) [1] in diverse and often heterogeneous computational environments with different requirements.

In web information systems for example, there is a need to reduce the amount of concurrency within an application, due to the unpredictable network latency. The solutions that cope with this problem [2] belong to a category of transaction models that are called *advanced transaction models*. In mobile computing, transaction processing faces frequent disconnections between clients and servers, as well as rigid resource constraints in processing power, memory and battery capacity. Moreover, there is a need to transfer responsibility of transaction management from one server to another upon handoffs. As a consequence of these problems, we are interested in transaction models [3], where resources acquired within a transaction can be released, before the transaction is completed (e.g. *nested transactions*, *split-join transactions* etc).

Implementing transaction processing is a complex task, due to the need to handle both synchronous and asynchronous transaction events that implies immense bookkeeping to achieve the intended behavior.

To the best of our knowledge, the ACID Model Checker and Code Generator is the first attempt to cope with this problem, by automatic verification of the transaction model of interest and by generating event management code. The presented tool was developed in the framework of *ACID Sim Tools* [4, 5], which is an integrated simulation environment for studying the performance and recovery tradeoffs in transaction processing architectures (e.g. [6] and [7]).

However, our contribution can potentially ease the development of transaction management systems and its applicability extends beyond ACID Sim Tools. The fundamental difference of our approach, when compared with a widespread model checker like SPIN [9, 10], is the capability of generating code for the management of transaction events. In fact, instead of verifying an abstract model of the transaction processing system, we provide a verified implementation of it.

This is similar to the compiler generators' concept, where the tool checks the syntax specification and subsequently generates a complete parser by integrating appropriate user-supplied code.

Section 2 lays the foundations for a state-machine based specification of a transaction model. This specification is necessary both for model-checking and code

generation. In section 3, we present the adopted approach for defining the transaction events generated by the operations declared in the described specification. This definition is required for verifying the intended transaction guarantees. Section 4 describes the path exploration algorithm for proving the correctness properties of interest. Section 5 shows the model checking of correctness properties for a 2PC protocol implementation. Section 6 describes the code generation for the ACID Sim Tools environment and the paper concludes with an overview of the overall contribution and the future research prospects.

2. Specification of transaction models

A transaction model is defined in terms of different *roles*, where each role is specified as a non-deterministic state machine. The two roles encountered in the 2PC protocol, for example, are the transaction coordinator and the worker. In nested transactions, we distinguish between parent and child transactions, thus resulting in four different roles, i.e. two roles for the parent transactions and another two roles for the child transactions. In most advanced transaction models we normally have more than two roles.

Non-determinism generally provides a convenient form of specification and therefore it is also used by the input specification languages of most well-known model checkers [9, 10]. In our case, non-deterministic role specification is the only feasible approach, because state transitions in transaction models may be determined based on information that is available only at runtime. For example, when the coordinator in the 2PC protocol collects the workers' votes for an ongoing transaction, either sends the decision made for the transaction (if it has already got all workers' votes) or waits for the remaining votes. As we will see in Section 4, our analysis resolves this non-determinism by taking into account the already executed state transitions.

In a role specification, the alphabet of the state machine is the set of all possible *transaction events*. Beyond the change in the current state, a transition for a given event also invokes one or more *operations*. An invoked operation in turn, creates or cancels transaction events or simply performs an assigned computation.

The transition relations for all roles are specified in a text file with five (5) comma-separated columns:

Role: The first column defines the state machine, in which the specified transition is part of. If we refer to the simplest transaction model, i.e. the 2PC protocol with only two roles, these roles are represented by "*c*" for the coordinator and "*w*" for the worker. The wildcard symbol "_" is used in transitions that belong to all transaction roles.

Source state: The state where the transition is enabled.

Event: The transaction event which triggers the transition. It represents a message from those included in the transaction protocol.

Next state: The target state for the specified transition.

Operations: A list of operation names that may be accompanied by one or more identifiers enclosed in brackets. The identifiers represent parameter names (e.g. message receiver, reference to a transactional job), that are used as placeholders for code generation. The first operation parameter is implicitly considered to be a unique transaction identifier, which is not written in the specification. In the provided list, operation names are separated by " : ". We use the character "–" for defining transitions with no operations.

We provide a specification excerpt taken from the 2PC implementation for the ACID Sim Tools:

```
//when a lock is acquired
1  _,  ST_LOCK,    LOCKED,              ST_JOB,     startJob
2  c,  ST_EMPTY    INIT,                ST_INIT,    sendInitLog:
       ,                                            scheduleTimeout
3  c,  ST_VOTES,   OUTCOME_ASKED,       ST_VOTES,   -
4  c,  ST_VOTES,   VOTE_LOGGED,         ST_VOTES,   collectVote{id}
5  w,  ST_WAIT,    START_JOB,           ST_LOCK,    workerLock{job}
6  w,  ST_JOB      JOB_FINISHED,        ST_WAIT,    removeJobMes-
                                                    sage{msg}
```

Line 1 defines a state transition that applies to all transaction roles. When the transaction participant is in state *ST_LOCK* and receives message *LOCKED*, then the state machine moves to state *ST_JOB* and the transition invokes the operation named *startJob* that starts a transactional job.

Line 2 defines a state transition from *ST_EMPTY* that for all roles specifies the initial state. This transition includes an event called *INIT*, which is the only event that is not caused by an operation of some transition. In fact, this event is sent by the transaction management system to start the processing of a transaction. Operation *sendInitLog* appends a log entry in stable storage for recovery purposes. If the log entry is successfully stored, the coordinator state machine receives the event *INIT_LOGGED*, which is not shown in the given excerpt.

The transition of line 3 is a typical example of a message that is ignored. Transitions that do not have any impact in the execution of the specified model are required to be explicitly defined, in order to ensure that there are no neglected transitions that may be feasible in certain circumstances. One of the uses of the ACID Model Checker is to detect forgotten state transitions,

which may be attributed to design flaws or specification omissions.

Table 1 describes all states of the specification of the 2PC protocol. Figure 1 visualizes the state transitions of a 2PC worker and Figure 2 shows the coordinator state machine (that also acts as a worker for the jobs processed locally). Vertices of the shown graphs represent the states of the corresponding roles, while edges signify state transitions caused by the event that labels the transition. Every transaction starts in the initial state called *ST_EMPTY* and its execution is completed at the final state called *ST_FINISH*.

Table 1: States of the 2PC transaction protocol

Role	State	Description
Both	ST_ABORT	Wait for the ABORT log entry to be stored
Both	ST_COMMIT_LOG	Wait for the COMMIT log entry to be stored
Both	ST_JOB	Wait for a job to complete
Both	ST_LOCK	Wait until a lock is acquired
Both	ST_LOG_END	Wait for the "END" log entry to be stored
Both	ST_RECOVER	Recovering the object states after an abort
Both	ST_WAIT	Wait for the next job or enter the voting phase
c	ST_INIT	Transaction processing was just started at the coordinator's site
c	ST_VOTES	Collecting votes from the workers
c	ST_FABORT	The transaction is aborted due to recovery from failure
c	ST_AKN_ABORT	Wait for abort acknowledgment from the workers
c	ST_WAK	Abort and acknowledgment of the decision are pending
c	ST_WAKR	Abort, recovery and acknowledgment of the decision are pending
w	ST_VOTE	Prepare to vote for the transaction
w	ST_DEC	Wait for the decision from the coordinator
w	ST_WAR	Abort and recovery are pending

The specification shown in this section suffices for generating code for the management of transaction events. This code, together with the user-provided code for the named operations yields a complete implementation of the 2PC transaction model. However, the provided information allows only trivial checks, such as the existence of unreachable states. For more sophisticated checks we also need to provide information regarding the creation and cancelling of transaction events, as a consequence of state transition operations.

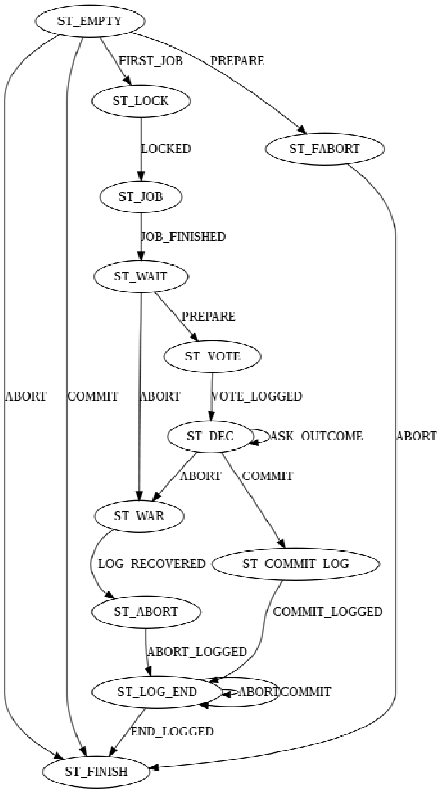

Figure 1. State machine for the worker in a 2PC transaction

3. Specification of transaction events

Beyond the INIT event, which is sent by the transaction management system, all other events are created by an operation invoked in one or more state transitions. This ensures that every possible event is a consequence of a past state transition. For model checking purposes, path exploration requires that apart from the state transition relation described in the previous section, we also have to complement the specification with the events generated or cancelled in each state transition operation. All operations that either create or cancel an event are specified in a text file with four (4) comma-separated columns:

Operation name: The name of the operation, without the parameters (if any). If an operation is invoked in state transitions of different roles, then a separate line is used for each role.

Event: The name of a transaction event. If the invoked operation creates an event selected from a set of alternatives, then all possible events are enumerated and are separated by "|". If the operation cancels an event the event name is prefixed by "-" (e.g. -*TIMEOUT*).

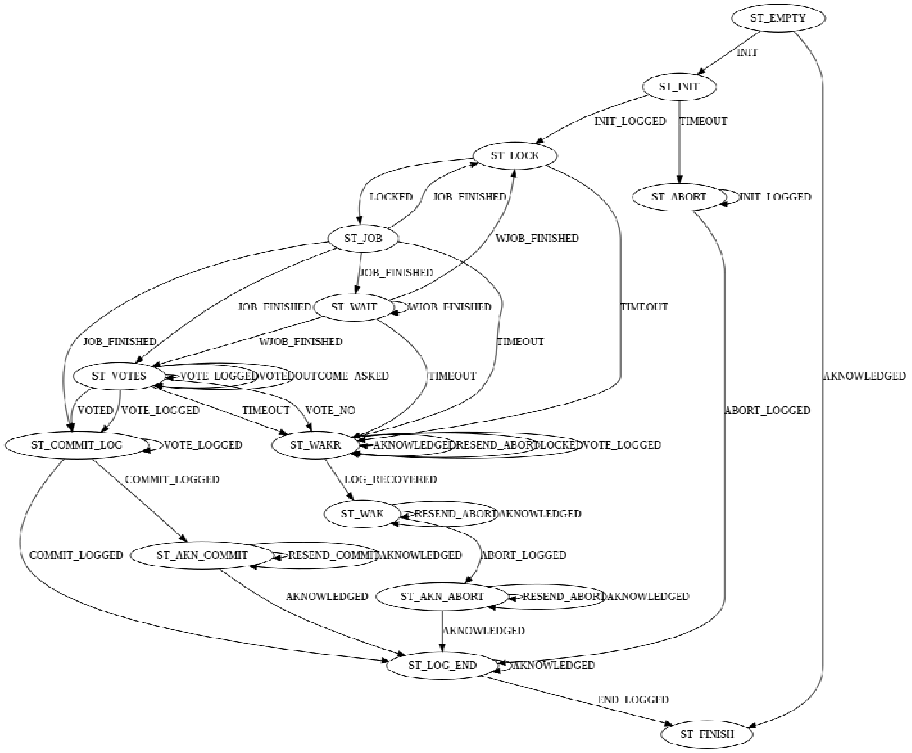

Figure 2. Non-deterministic state machine for the coordinator in a 2PC transaction

Receiver: The role of the event consumer (in 2PC it may be either "*c*" or "*w*").

Sender: The event producer, which may be either a specified role or any other *implementation specific component*. In 2PC for example, the necessary transaction processing components are concurrency control denoted by *lc*[*r*] and stable storage, denoted by *lg*[*r*], where $r \in \{$"*c*", "*w*"$\}$).

A specification excerpt taken from the 2PC implementation for the ACID Sim Tools follows:

```
1  startJob,        JOB_FINISHED,          c,   c
2  startJob,        JOB_FINISHED,          w,   w
3  sendInitLog,     INIT_LOGGED,           c,   lgc
4  workerLock,      LOCKED,                w,   lcw
5  cancelTimeout,   -TIMEOUT,              c,   c
6  nextWorkerJob,   FIRST_JOB | START_JOB  w,   c
```

Lines 1 and 2 demonstrate a typical case of an operation invoked by state transitions of two different roles. Moreover, we see that in both cases the produced event *JOB_FINISHED* is created and consumed by the same role, therefore representing a message sent to the role itself. In line 3, as a consequence of the invocation of operation *sendInitLog* by the coordinator role (see line 2 of previous specification excerpt), the "*lgc*" component replies with event *INIT_LOGGED*. Here, we note that we are only interested in specifying the events consumed by the defined roles and for this reason we do not need to include the event that causes "*lgc*" to respond with *INIT_LOGGED* (i.e. the coordinator's state transition `(ST_EMPTY, INIT)` → `ST_INIT`). Line 5 shows an operation that cancels an event. When the shown operation is invoked, if *TIMEOUT* has been previously produced, then this event is cancelled and is never consumed. Finally, line 6 shows an operation that non-deterministically produces either the event named *FIRST_JOB* or the event *START_JOB*.

At this point, we have a complete specification of the transaction model of interest, with two sources of non-determinism. The first source is the role specification, where non-determinism is introduced as discussed in the previous section and the second source is the operations with non-deterministic event production, as the one shown in line 6. Path exploration for model checking correctness properties is possible, only if these two kinds of non-determinism are resolved.

4. Path exploration

Path exploration was implemented in Haskell. It is based on computing the *reachability graph* for the

synchronized product of the role state machines [8]. Reachable states are accessed,

- through the manipulation of a list of produced transaction events, that from now on will be called *future event list* (*fev*)
- by consuming all events in *fev* that are produced by implementation specific components – i.e. not roles – in FIFO order. At the same time, reachable states include all possible event interleavings in *fev*, for all events produced by the specified roles
- by synchronizing the cartesian product of the defined state machines, on the basis of a set of *synchronizing events* [8] including all messages exchanged by the protocol participants.

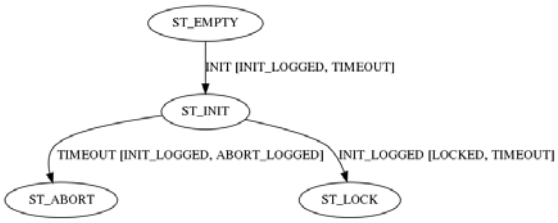

Figure 3. Reachability graph for the synchronized product of the role state machines

Let us consider the partial reachability graph derived from the synchronized product of roles "c" and "w" shown in Figure 3 that basically displays the first transitions of the coordinator role for 2PC (specification of section 2). All edges are labelled with pairs, where the first item denotes the consumed event that causes the transition and the second item is the updated *fev* list. Event *TIMEOUT* in the *fev* of the first transition is produced by "*c*" and *INIT_LOGGED* is produced by the "*lgc*" component. The reachability graph includes all possible interleavings between the two events in the list:

- One possible transition takes place by first consuming the *TIMEOUT* event. The event is removed from the previous *fev* and its name becomes the first item of the new transition label. The updated *fev* now includes *ABORT_LOGGED* that is produced by the operation invoked by the executed transition.
- The other transition takes place by first consuming the *INIT_LOGGED* event.

The *fev* of the first mentioned transition now includes two events sent by an implementation specific component that is not a role, but provides log recording in stable storage. These two events in *fev* will be consumed in FIFO order. Figure 4 shows the complete reachability graph for the synchronized roles in 2PC. The graph justifies the need for automatic verification, due to the considerable complexity that is evident even in the simple case of the 2PC transaction model.

Algorithm for exploring maximal paths (path_explorer)

Input:

- A tupple (q_r, fev_r) for each role r, where q_r is the current local state and fev_r the local future event list. Item q_r is initialized with the initial state of the role state machine and fev_r is initialized with [].
- A list r_trans with the transitions specified as described in Section 2 for all roles.
- A list ops with the operations specified as described in Section 3.
- The event *ev* consumed for the triggered transition (first consumed event is called *INIT*).
- The computed (non-maximal) path denoted by path, which is initialized by []. Each item of path is a 3-tupple (q_{pre}, *e*, q_{post}) representing the transition $q_{pre} \xrightarrow{e} q_{post}$.

Output:

- The computed maximal paths max_paths, initialized by [].

Description:

1. Find in r_trans all $q_r \xrightarrow{ev} q_r'$ that consume event *ev*. If there is no transition for event *ev*, *r* has an incomplete transition relation and the user is prompted to define the missing transition. If there are multiple transitions such as $(q_r \xrightarrow{ev} q_r') \notin$ path, then we have a non-deterministic definition, which must be resolved by a user-supplied predicate over path.
2. For each transition $t : (q_r \xrightarrow{ev} q_r') \notin$ path, let ev_t be the set of events caused by the operations in ops for transition *t*. If some operation of *t* produces more than one event, then a user-supplied predicate over path selects the valid events in this context.
   1. Let $fev_r' = [x : x \in fev_r, \neg x \notin ev_t] \; ++ [x : x \in ev_t, \neg cancel(x)]$ be the new future event list. We note that the concatenation operator (++) preserves the event order in the list.
   2. If q_r' is a *final state*, set q_r' to the initial role state and keep from fev_r' only the events produced by the other roles.
3. Set path′ = path + *t*
4. For each event *e* in fev_r':
   1. If *e* is the first event sent by a certain implementation specific component that is not a role
   2. OR *e* is an event produced by some role

   Call path_explorer with fev_r' / [*e*] instead of fev_r, q_r' instead of q_r, *e* as the consumed event for the triggered transition and path′ instead of path.
5. If fev_r' = [], then path′ is a maximal path and it is appended to max_paths.

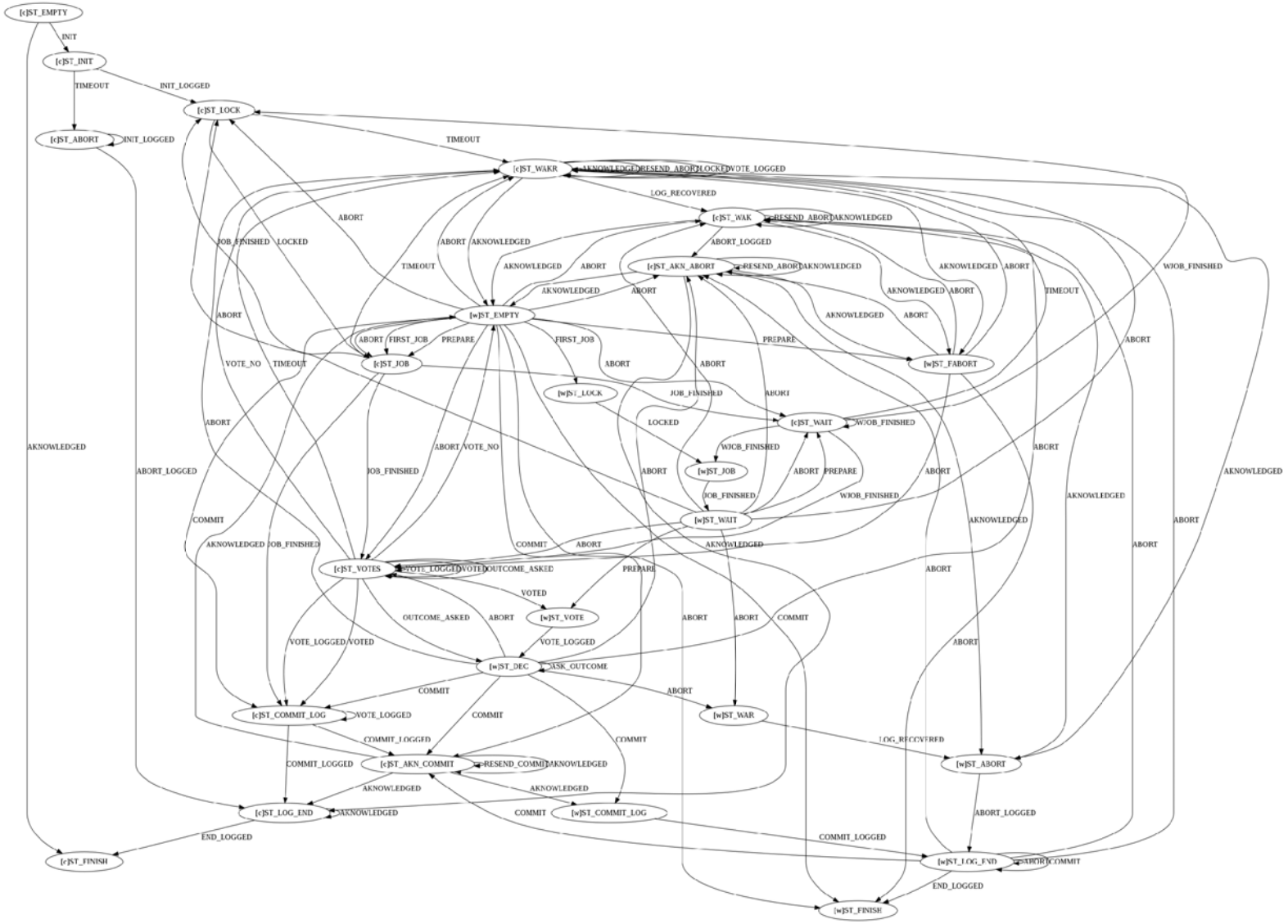

Figure 4. Reachability graph for the 2PC transaction model

A *path* is a sequence of state transitions and is maximal, if the source state of the first transition is an initial role state and there is no event to be consumed from the target state of the last transition in the path.

If a transaction correctness property is violated in a non-maximal path, say `path`, this does not imply that the property is also violated in some maximal path prefixed by `path`, because `path` does not represent a complete execution. Therefore, all correctness properties are checked upon the maximal paths of the synchronized product, which are computed by the recursive algorithm `path_explorer`.

We note that `path_explorer` utilizes two user-defined predicates in steps 1 and 2 to resolve the two sources of non-determinism discussed previously.

Our tool detects all non-deterministic state transitions and operations and prompts the user to define an appropriate predicate over `path`. The non-determinism is thus resolved according to the semantics of the transaction model. In 2PC, we have only five (5) cases of non-determinism caused by the state transitions of the coordinator (*c*) and one (1) caused by an operation.

For example, when role *c* collects the votes, it may either decide the outcome of the ongoing transaction or wait for the votes of the remaining workers. For model checking purposes it suffices to assume the minimum number of participants materializing the interaction between the roles. An appropriate predicate examines the previous transitions recorded in `path` and determines if the required number of votes is collected.

The non-determinism of operation *nextWorkerJob* (see the specification excerpt of section 3) that produces either the event *FIRST_JOB* or *START_JOB*, is resolved by examining if the event *FIRST_JOB* exists in `path`. If *FIRST_JOB* is found in `path`, then the operation produces the event *START_JOB*, otherwise the event *FIRST_JOB* is produced. From our experience, the definition of the discussed predicates is straightforward.

5. Model checking transaction guarantees

The ACID Model checker detects two kinds of specification errors. *Structural errors* include unreachable states, unreachable transitions and incomplete definition of a transition relation. *Protocol specific correctness properties* are expressed by user-defined functions of type `path` → *Bool* written in Haskell. Function *f*: `path` → *Bool* should return `True`, if the path violates the property of interest. Let

$$inv = \{p \in \max\_paths : f(p) = True\}$$

be the set of invalid paths, with respect to some correctness properties. The checked properties are not violated, if *inv* is the empty set. Otherwise, the elements of *inv* represent counterexamples of violated properties.

In 2PC, let us consider the correctness property:

"Workers conform to the decision of the coordinator, which is either commit or abort."

The two predicates shown in the following Haskell code excerpt, when combined by OR, result in a predicate that verifies the aforementioned property.

```
both_commit path = exists worker_role path
                   &&
                   exists (\x -> role x == Coordinator
                         && sname x == "ST_COMMIT_LOG") path
                   &&
                   not_exist (\x -> receiver x == Fsm Worker
                           && ename x == "COMMIT") path

both_abort path = exists worker_role path
                  &&
                  exists (\x -> role x == Coordinator
                        && sname x == "ST_WAKR") path
                  &&
                  not_exist (\x -> receiver x == Fsm Worker
                          && ename x == "ABORT") path
```

Beyond the function `exists`, we have also defined function `preceed`, which is used to express temporal relations. Other correctness properties that were verified are:

- Workers and the coordinator reach exactly one of two possible decisions – *abort* or *commit*.
- The commit decision can be reached, only if all participants have voted "commit".
- Participants eventually reach a decision, even in the presence of communication or system failures.

6. Code generation for the ACID Sim Tools

Although the code generation described in current section is specific to the ACID Sim Tools framework, we believe that the general principles are the same for any potential implementation of a transaction management system.

The ACID Sim Tools framework provides a state-machine-guided transaction execution mechanism. Our mechanism uses the verified event management code, which is automatically generated from the provided descriptions that were introduced in sections 2 and 3.

At runtime, the transaction execution mechanism monitors an ongoing transaction in terms of the traversed protocol-specific states of the provided state-machine. The performed state transitions invoke the code of the associated operations that are implemented in the class hierarchies of ACID Sim Tools for simulating essential services like for example log recording in stable storage, handling of lock requests for concurrency control and so on.

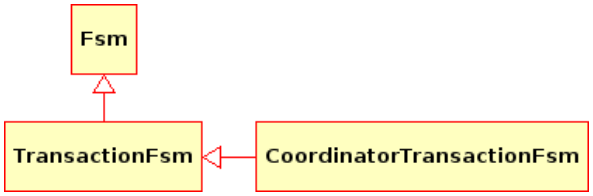

Figure 5. Class hierarchy for the state-machine-guided execution mechanism

For every new transaction request, the receiver coordinates the execution of the requested transaction and first creates a new instance of the class `CoordinatorTransactionFsm`, which acts as a transaction execution monitor. The class hierarchy of Figure 5 provides all abstractions needed for monitoring the execution of the simulated transactions. The base class is called `Fsm` and implements read and write access to the current state. `TransactionFsm` provides access to additional information related to the transaction class, the set of jobs executed on the server and to data that are used for drawing statistics.

When a transaction monitor controls the actions performed in a coordinator module, it is an instance of class `CoordinatorTransactionFsm` that encapsulates also a pointer to a `Transaction` object, the number of transaction managers expected to acknowledge the decision for the transaction and two sets of identifiers: the first set representing the transaction managers that have already acknowledged the decision and the second set representing the transaction managers that voted for the transaction outcome.

We noted that the provided specifications for the generation of transaction execution monitors define non-deterministic state machines. Non-deterministic state transitions are detected by our code generation tool, when the transaction specification includes two or more transitions that are triggered from the same outgoing state for a given event name. The generator produces an appropriate handler method that returns the next state by resolving non-determinism at runtime. This particular method is named by concatenating the label of the outgoing state with the event name that causes non-determinism and accepts as parameter a `CoordinatorTransactionFsm` in case of coordinator processing or a `TransactionFsm` in case of worker processing. As an example, if there is non-determinism at the state *ST_VOTES* for the event *VOTED*, the generated handler method is called `resolveVotesVoted`.

7. Conclusion

The ACID Model Checker and Code Generator makes it possible to deliver provably correct implementations of traditional or advanced transaction models. We presented the adopted approach for specifying, model checking and eventually generating code for the management of the transaction events. Our proposal was demonstrated by the implementation of the 2PC transaction model in the ACID Sim Tools framework. We described how the well-known 2PC correctness properties are verified in our tool.

In the near future, we plan to release the ACID Model Checker and Code Generator as open source software available at the ACID Sim Tools web site [4]. Also, we are going to investigate the possibility to utilize the constructs of the ACTA formalism [11] for the specification and verification of advanced transaction models in our state machine based approach.

References

[1] Weikum, G. and G. Vossen, Transactional Information Systems, Morgan Kaufmann Publishers, San Francisco, 2002

[2] Little, M., Freund, T., A comparison of Web services transaction protocols, on developerWorks (online: http://www.ibm.com/developerworks/webservices/library/ws-comproto/), 2003

[3] Serrano-Alvarado, P., Roncancio, C., Abiba, M., A survey of mobile transactions, Distributed & Parallel Databases, Vol. 16 (2), 2004, pp. 193-230

[4] ACID Sim Tools Site, http://mathind.csd.auth.gr/acid/html/index.html (last access: 31st of January 2009)

[5] Mentis, A., Katsaros, P. and Angelis, L. ACID Sim Tools: A simulation framework for distributed transaction processing architectures, In Proc. of the 1st Int. Conf. on Simulation Tools & Techniques (Simulation Works Industry Track), Marseille, France, (http://proceedings-online.org/?eudlQuery=SimulationWorks%2008), 2008

[6] Object Management Group, Transaction Service Specification, version 1.3, OMG Technical Committee Document ptc/2003-03-08, March 2003

[7] Sun Microsystems, Enterprise JavaBeans, Version 2.1 (Final Release), Nov. 2003

[8] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Achnoebelen, Ph., McKenzie, P. Systems and Software Verification, Springer, 2001

[9] The SPIN Model Checker web site, http://spinroot.com/spin/whatispin.html (last access: 31st of January 2009)

[10] Ben-Ari, M. Principles of the Spin Model Checker, Springer, 2008

[11] Chrysanthis, P., Ramamritham, K. Synthesis of extended transaction models using ACTA, ACM Transactions on Database Systems, Vol. 19 (3), 1994, pp. 450-491