Model checking and code generation for transaction processing software

Anakreon Mentis and Panagiotis Katsaros
e-mail: anakreon@csd.auth.gr, katsaros@csd.auth.gr
Department of Informatics, Aristotle University of Thessaloniki

Abstract

In modern transaction processing software, the ACID properties (Atomicity, Consistency, Isolation, Durability) are often relaxed, in order to address requirements that arise in computing environments of today. Typical examples are the long-running transactions in mobile computing, in service oriented architectures and B2B collaborative applications. These new transaction models are collectively known as advanced or extended transactions. Formal specification and reasoning for transaction properties has been limited to proof-theoretic approaches, despite the recent progress in model checking. In this work, we present a model-driven approach for generating a provably correct implementation of the transaction model of interest. The model is specified by state machines for the transaction participants, which are synchronized on a set of events. All possible execution paths of the synchronized state machines are checked for property violations. An implementation for the verified transaction model is then automatically generated. To demonstrate the approach, the specification of nested transactions is verified, because it is the basis for many advanced transaction models.

Keywords transaction processing; model-driven development; code generation; model-checking

1 INTRODUCTION

Many applications are partitioned into several tiers (e.g. presentation, logic and data), potentially distributed in a network of heterogeneous computing environments. Transactions are used to ensure that a mutually agreed outcome is observed consistently across all involved parties in the performed computations.

However, the characteristics (Atomicity, Consistency, Isolation and Durability) of the initially proposed transaction model [23] set it inappropriate for the requirements of modern distributed computing. Thus, in web information systems we need to reduce the degree of concurrency within an application, due to the unpredictable network latency [22]. In mobile computing, we face frequent disconnections between clients and servers, as well as rigid resource

constraints in processing power, memory and battery capacity [4]. Also, upon hand-offs there is a need to transfer the transaction management from one server to another [19, 21].

Advanced or extended transaction models (e.g. nested transactions, split-join transactions etc) aim to cope with the aforementioned problems [22, 19]. In essence, the fundamental ACID properties of the typical transaction model are relaxed.

A transaction model consists of a workflow with role specifications for the transaction parties that interact via asynchronous messages and protocol actions that enable computations upon receipt of a message. Implementing transaction processing is a complex task, due to the asynchronous events that imply immense bookkeeping. This work introduces a model-driven development approach [20] for model checking [7] the anticipated transaction properties and automatically generating implementations of the verified models. Related work on specifying and reasoning for transaction properties [14, 6, 10] presumes the use of specialized formalisms. In our approach, transaction parties are defined intuitively by the use of non-deterministic state machines with transitions that trigger asynchronous events. Correctness properties are checked on all feasible execution paths of the synchronized product of the state machines. Automated verification does not require formal analysis skills and the verified specification is then used to automatically generate code for the management of transaction events. This feature distinguishes our approach from other model checking alternatives that may be based on general-purpose model checkers (e.g. SPIN [9]). Our proposal resembles the compiler generators' concept, where the tool checks the specification of the language syntax and subsequently generates a complete parser by integrating appropriate user-supplied code.

The verification and code generation functions have been implemented in a tool called ACID Model Checker & Code Generator [1, 15]. The tool generates code for ACID Sim Tools [16], a simulator for studying the performance and recovery trade-offs [12] in transaction processing. However, our contribution can be employed in the development of transaction management systems and therefore its applicability extends beyond ACID Sim Tools.

Preliminary results for the basic transaction model were published in [15]. In this article, we verify the properties of nested transactions [8], a well known advanced transaction model. Nested transactions involve more interacting parties, thus resulting in a significantly larger state space. We introduced a pre-processing phase that removes transitions which do not affect the verification result. The improved path exploration algorithm employs user–supplied guards that resolve certain cases of non–determinism.

Section 2 outlines the proposed model-driven development process and the nested transaction model. Sections 3 and 4 introduce the state-machine based specification of transaction models. Section 5 describes the path exploration algorithm for verifying the correctness properties of interest. Section 6 shows the model checking of correctness properties for the nested transaction model. Section 7 describes the code generation for ACID Sim Tools. Section 8 reviews related work and the paper concludes with an overview of the overall contribu-

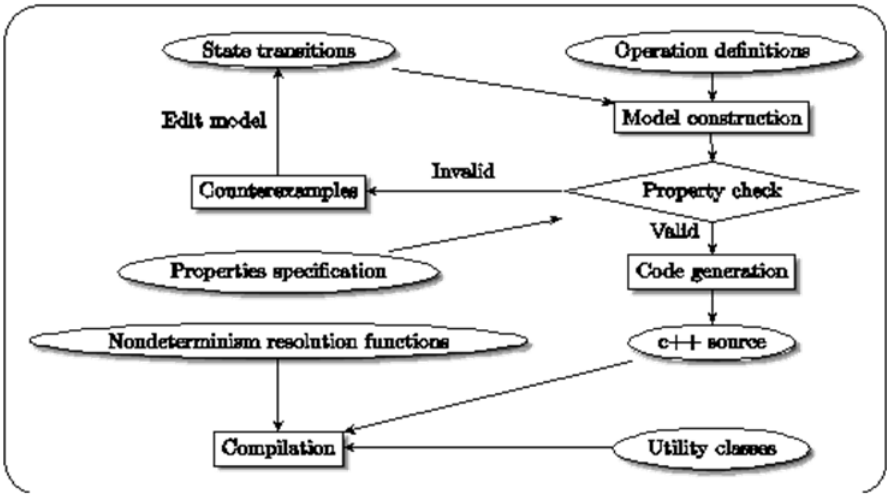

Figure 1: Model-driven verification and implementation of transaction models

tion and the future research prospects.

2 MODEL-DRIVEN DEVELOPMENT OF NESTED TRANSACTIONS PROCESSING

The process (Fig. 1) for verifying the correctness properties of a transaction model and automatically generating the event management code involves model definition, property specification, path exploration for property verification and code generation.

A model consists of the state machines for the transaction roles and the events that are dispatched, scheduled or canceled by operations invoked upon state transitions. Model construction computes the reachability graph for the synchronized product [3] of the role state machines. The synchronized product is defined over the Cartesian product of the state machines, restricted by the synchronization events, i.e. all events consumed by a role other than the producer. The graph includes all event interleavings and it is used for discovering structural errors (e.g. unreachable states or transitions, incomplete transition relation) and violations of transaction properties. Properties are defined in terms of set operators and basic temporal relations over the computed execution paths. When a property violation is detected, a counterexample is provided for the correction of the role state machines.

Code generation produces a C++ method that handles the state transitions of the defined roles. The method determines the next role state and executes the associated operations based on the current state, the received event and the specified state transitions. Non–determinism is introduced because of lack of in-

formation that becomes available only at runtime. In model construction, non–determinism is resolved by considering all possible execution histories. For code generation, the developer is expected to supply custom code that determines the successor state, based on information which is accessible during execution.

Nested transactions [8] were the first advanced transaction model adapted to the specific requirements of distributed computing environments. It inspired recently proposed advanced transaction models in mobile computing [19], web transactions [22] and other areas. A transaction is composed of sub–transactions in a hierarchical manner. A sub–transaction can initiate other sub–transactions for which it acts as a coordinator, i.e. it orchestrates the global decision reaching process. A parent transaction can only commit after all its children are completed. If the parent chooses to abort, a child transaction also aborts. However, a child transaction can abort independently of its parent. If a sub–transaction aborts, the parent might trigger another sub–transaction as an alternative.

3 ROLE SPECIFICATION

A transaction model is defined in terms of different roles, where each role is represented by a non–deterministic state machine. The two roles encountered in the 2PC protocol [15] are the transaction coordinator and the worker. In nested transactions, there are four roles that distribute transaction processing in different levels: the coordinator of the top–level transaction, the coordinator of a sub–transaction and a worker for each coordinator. Worker roles have identical behavior. As we already noted, non–deterministic specification is the only feasible approach, because of information that emerges at runtime. For example, during vote collection the coordinator either announces the transaction outcome if all workers have voted or waits for the remaining votes. As we will see in Section 5, this non-determinism is resolved.

The alphabet of a role state machine is the set of all possible transaction events. A transition changes the current state and invokes one or more operations. Operations create or cancel events or simply perform an assigned computation. The transition relations for all roles are specified in a text file with five comma–separated columns:

- **Role**: The first column defines the state machine in which the specified transition is part of. In the nested transaction model, the roles are represented by “c” for the top-level coordinator, “cn” for the coordinator of a sub-transaction and “w” for the worker.
- **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.

Table 1: A specification excerpt of nested transactions roles

	Source state	Event	Next state	Operations
1 c,	ST_EMPTY,	INIT,	ST_INIT,	sendInitLog:scheduleTimeout
2 cn,	ST_EMPTY,	SPAWN,	ST_INIT,	sendSubInitLog:scheduleTimeout
3 c,	ST_VOTES,	OUTCOME_ASKED,	ST_VOTES,	−
4 c,	ST_VOTES,	VOTE_LOGGED,	ST_VOTES,	collectVoteid
5 w,	ST_WAIT,	START_JOB,	ST_LOCK,	workerLockjob
6 w,	ST_COMMIT_LOG,	COMMIT_LOGGED,	ST_FINISH,	clearLocks:aknowledgeCommit
7 w,	ST_JOB,	JOB_FINISHED,	ST_WAIT,	removeJobMessagemsg
8 c,	ST_WAIT,	JOB_FINISHED,	ST_WAIT,	nextJob:moveNext
9 c,	ST_WAIT,	JOB_FINISHED,	ST_VOTES,	sendPrepare:sendFinalCommit
10 c,	ST_WAIT,	JOB_FINISHED,	ST_SUB_WAIT,	startSubTransaction:moveNext

- **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. the message receiver and the reference to a transactional job) 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. Operation names are separated by ":". Character "−" defines transitions with no operations.

Line 1 of Table 1 defines a state transition from `ST_EMPTY` that for all roles is the initial state. This transition is triggered by `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. When the coordinator receives the message, the state machine moves to state `ST_INIT` and the transition invokes the operation `sendInitLog` that appends a log entry in stable storage for recovery purposes. If the log entry is successfully stored, the coordinator receives the event `INIT_LOGGED`, which is not shown in the given excerpt. The other invoked method, `scheduleTimeout`, schedules a `TIMEOUT` event that is received if the transaction is not completed on time. Line 2 defines a transition for role "cn". In contrast to the top–level coordinator, sub–transactions are initiated by `SPAWN` instead of `INIT` and use a different log entry to record in stable storage that a sub–transaction is initiated.

The transition of line 3 is an example of a message that is ignored. Transitions that do not have any impact in model execution are required to be explicitly specified, in order to ensure that there are no neglected transitions. 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.

Line 6 demonstrates the behavior of a worker upon a transaction commit: the acquired locks are released and the coordinator's decision is acknowledged. Finally, lines 8, 9 and 10 demonstrate a non–deterministic transition. Depending on the structure of the transaction, the coordinator either starts a new sub-transaction or submits a new job to a worker or enters the voting phase.

The shown specification suffices for generating code for the management of

transaction events. Together with the user–provided code for the named operations a complete implementation of the transaction model is produced. However, the provided information allows only trivial checks, such as the existence of unreachable states in each role. For complete model verification we also need to provide information regarding the creation and canceling of transaction events, as a consequence of transition operations.

4 SPECIFICATION OF TRANSACTION EVENTS

Apart from the `INIT` event, which is sent by the transaction management system, all other events are created by an operation invoked in a state transition. Thus, every event is a consequence of a past state transition. For model checking purposes, we require a specification of the events that are generated or canceled by the operations invoked in a state transition. These operations are specified in a text file with four comma-separated columns:

- **Operation**: The name of the operation, without the parameters. Separate lines are used for each role that invokes the operation.
- **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 by "|". The event name is prefixed by "-", if the operation cancels it (e.g. `-TIMEOUT`).
- **Receiver**: The role of the event consumer ("c", "cn" or "w").
- **Sender**: The event producer, which may be either a specified role or any other implementation specific component. In nested transactions for example, the necessary transaction processing components are concurrency control denoted by `lcw` and stable storage, denoted by `lg[r]`, where $r \in$ {"c", "cn", "w"}.

Lines 1–2 of Table 2 demonstrate a typical case of an operation invoked by state transitions of two different roles. In line 3, as a consequence of the invoked operation `sendInitLog` by the coordinator role (see line 1 of Table 1), the "lgc" component replies with event `INIT_LOGGED`. Here, we note that we are only interested for defining the events consumed by the roles and thus 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`) $\rightarrow$ `ST_INIT`). Line 5 shows an operation that when invoked, cancels the scheduled event `TIMEOUT`. Finally, line 6 shows an operation that non-deterministically produces either the event `FIRST_JOB` or the event `START_JOB`.

At this point, we have a complete specification of the transaction model, with two sources of non–determinism. The first source is the role specification, where non–determinism is introduced as discussed in the previous section. The second source of non–determinism are all operations that produce alternative events (as in line 6). The events produced by the defined operations may depend on

Table 2: Specification excerpt of transaction events

	Operation	Event	Receiver	Sender
1	startSubTransaction,	SPAWN,	cn,	c
2	startSubTransaction,	SPAWN,	cn,	cn
3	sendInitLog,	INIT_LOGGED,	c,	lgc
4	workerLock,	LOCKED,	w,	lcw
5	cancelTimeout,	-TIMEOUT,	c,	c
6	nextWorkerJob,	FIRST_JOB | START_JOB	w,	c

Table 3: Guard definition in ACID Model Checker & Code Generator

```
valid path (Event "SPAWN"     "c"  "cn") = (not . has_event_name "SPAWN") path
valid path (Event "SPAWN"     "cn" "cn") = False
valid path (Event "FIRST_JOB" "c"  "w")  = (not . has_role "w") path
valid path (Event "START_JOB" "c"  "w")  = has_event (Event "FIRST_JOB" "c" "w") path
```

information available at runtime, e.g. the structure of an executed transaction. This non–determinism is eliminated by means of guards that impose constraints on the events generated by the defined operation, based on the execution history. In the current version of the ACID Model Checker there is no syntactic support for guard definition. Instead, guards are provided in a compact declarative form (Table 3) in the same language used for the tool implementation.

In the first line of Table 3 the guard prevents the generation of a `SPAWN` event by the top–level coordinator if another `SPAWN` event has previously occurred. In line 2, the guard states that a sub-transaction coordinator cannot create a child sub–transaction. If this condition was omitted the depth of the transaction tree structure would be infinite. The last two lines resolve the non–determinism introduced by the operation `nextWorkerJob` in line 6 of Table 2. For the nested transaction model, 6 cases of non–determinism exist among 32 operations that generate events. In case of omitted guards, the tool detects "omitted" state transitions that in fact cannot occur, because the event that triggers the transition is not feasible.

5 PATH EXPLORATION

Path exploration is the computation of the reachability graph for the synchronized product of the role state machines [3]. Reachable states are accessed,

- through the manipulation of a list of produced transaction events, referred as future event list (*fev*);
- by consuming in FIFO order the *fev* events that are not produced by

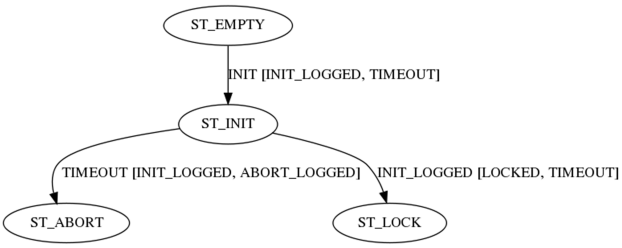

Figure 2: Reachability graph for the synchronized product of the role state machines

roles, while at the same time, reachable states represent all possible event interleavings in *fev*;

- by synchronizing the Cartesian product of the defined state machines on the synchronization events.

Consider the partial reachability graph of Fig. 2 that displays the first transitions of role "c". Edges are labelled with pairs, where the first item denotes the event that triggers the transition and the second item shows 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:

- One possible transition takes place by first consuming the `TIMEOUT` event. The event is removed from the previous *fev*. The updated *fev* now includes `ABORT_LOGGED` that is produced by the operation invoked in this 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 - not a role - that provides log recording in stable storage. These two events in *fev* will be consumed in FIFO order.

In the discussed transaction model there are transitions that do not alter the current state nor produce or cancel events. They contribute into developing a totally defined transition relation and we call them empty transitions. Such transitions are safely removed under the following conditions:

1. If an event is non–deterministic for a given state, the empty transitions for that state are irrelevant. They are ignored and only the other transitions are taken into account.
2. When an event triggers only empty transitions in a sub–path from some state to the final role state, this event and all subsequent empty transitions are ignored.

In effect, this pre–processing reduces the number of transitions that are taken into account and improves the model checking performance. The expected improvement depends on the number of empty transitions that fulfill the aforementioned conditions.

A path is maximal, if the source state of the first transition is the initial state and there is no event to be consumed from the target state of the last transition. Maximal paths represent complete executions of a transaction and therefore all correctness properties are checked upon the maximal paths of the synchronized product, which are computed by the recursive algorithm *path_explorer* (Fig. 3).

In step 2 of *path_explorer* only the transitions that have not been previously selected are taken into account. In this way, multiple occurrences of cycles are excluded, which otherwise would result in infinite maximal paths. The rationale for this choice is that at runtime, cycles are followed finitely many times and are eventually broken. For the worker role, a chain of transitions that are followed for each job assigned to the same worker forms a cycle. However, transactions with more or less jobs for a worker do not differ with respect to the correctness of transaction properties. The proposed algorithm explores a directed acyclic graph, for which it can be shown that contains a finite number of paths. Hence, the algorithm eventually terminates.

If the events communicated between the interacting model entities are delivered in FIFO order (e.g. TCP network protocol), the model definition would be simpler and it would be verified in a shorter time. For the sake of generality, we did not assume that role events are delivered in FIFO order, as opposed to those produced by implementation specific components (e.g. `lgw`, `lcw`) that reside in the same system with the role consuming the events without network communication. Nonetheless, a slight change in step 4 of Fig. 3 would realize FIFO delivery of role events.

An inherent problem of model checking is the possibility of state space explosion. In this case, the memory needed for the reachability graph or the time required for building it, renders the verification impractical. ACID Model Checker was implemented in Haskell [11], which is a functional language. Due to the lazy evaluation of functions, property violations and model errors are reported without having to complete the construction of the reachability graph. Instead, a maximal path is checked, as soon as it is constructed. Validated paths are not stored in memory hence resulting in low memory consumption. In effect, even a partial model checking run reveals property violations and model errors.

The algorithm takes into account all event interleavings, even if some of them have the same effect with respect to the checked properties. This is not an optimal solution, since for transaction models with many roles the time required for a full verification can be exceptionally long, thus rendering the approach impractical. Partial order reduction [2] delivers proven techniques for reducing the number of interleavings that need to be analyzed, thus opening a promising perspective towards improving the mentioned shortcoming.

For model checking is sufficient to consider the minimum number of participants materializing all possible role interactions. In the nested transaction model, we took into account the interactions between a top-level coordinator, a

Algorithm path_explorer

Input:

- A tuple (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 [ ] (the empty list).
- A list r_trans with the transitions specified as described in Section 3 for all roles.
- A list ops with the operations specified as described in Section 4.
- A list $ignored$ with the states and the events that may be ignored for some state.
- 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-tuple (q_{pre}, e, q_{post}) representing the transition $q_{pre} \xrightarrow{e} q_{post}$.

Output:

- A list of computed maximal paths, max_paths.

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 is resolved in the next step.
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.
   (a) Let $fev'_r = [x : x \in fev_r, -x \notin ev_t] ++ [x : x \in ev_t, \neg cancel(x) \wedge P(x) \wedge (q_r, x) \notin ignored]$ be the new future event list, where P is a guard. We note that the concatenation operator $(++)$ preserves the event order in the list, which is important for the FIFO executed events.
   (b) 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 produced by a role and for the first event of each implementation specific component: 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 $fev'_r = [\,]$ and $\forall l \in roles - \{r\} fev_l = [\,]$, then $path'$ is a maximal path and it is appended to max_paths.
   Else if $\exists l \in roles - \{r\} : fev_l \neq [\,]$ then apply step 4 with fev_l instead of fev'_r.

Figure 3: Algorithm for exploring maximal paths

Table 4: Haskell functions used in property specifications

has_event_name e path	an event with name e occurs in path
has_event e path	event e occurs in path
has_role r path	a transition concerning role r exists in path
has_state s path	a state s occurs in path
exists x path	existential operator
absent x path	negation of exists
before t1 t2 path	t1 occurs before t2 in path
after t1 t2 path	t1 occurs after t2 in path

coordinator for a sub-transaction and two workers, each one being subordinate of one of the two coordinator roles.

6 VERIFICATION OF TRANSACTION CORRECTNESS PROPERTIES

The ACID Model checker detects structural errors and violations of transaction properties. Structural errors include unreachable states or transitions and incomplete definition of the transition relation. Transaction correctness includes properties related to the temporal behavior of the considered transaction model.

A role state is unreachable if there is no maximal path in which it is accessed. The set of unreachable states is computed by subtracting the set of all accessed states in the traversed maximal paths from the set of the defined role states. If the result is not the empty set, the unreachable states are reported. A transition is unreachable if it does not occur in any maximal path. The set of unreachable transitions is computed in a similar manner.

If *path_explorer* accesses a role state s where the role receives an event e for which there is no defined transition (step 1 in Fig. 3), then the role state machine is incomplete. In this case, ACID Model checker reports the path where the algorithm discovered the aforementioned omission. Incomplete definitions are reported as soon as they are detected, without awaiting the computation of all maximal paths. From our experience, incomplete definition was a frequent source of errors in the developed transaction models. In the absence of the proposed model-driven approach, an incomplete transition relation would cause incorrect behavior at runtime. It is generally difficult to attribute the incomplete transition relation as the cause of the erroneous behavior, based solely on the observed effects in the execution of the transactions.

In ACID Model checker, protocol specific correctness properties are expressed by user-defined functions of type $path \rightarrow Bool$ written in Haskell. Property specifications are defined by reusing the functions of Table. 4, combined with the boolean operators `and`, `or`, `not` etc.

Safety properties express that an undesirable event never occurs [3]. Func-

tion $f : path \rightarrow Bool$ should return `True`, if the property of interest does not hold in the given path.

Liveness properties express that some particular event will ultimately occur. A function $f : path \rightarrow Bool$ should return `True`, if the expected event does not occur in the given path.

Reachability correctness properties express that some particular situation can be reached. In essence, a reachability property is the negation of a safety property, i.e. the function $f : path \rightarrow Bool$ should return `True`, if the property of interest holds.

Let $checked\_paths = \{p \in max\_paths : f(p) = True\}$ be the set of maximal paths, with respect to some correctness property. For safety and liveness properties, if *checked_paths* is not empty, its elements represent counterexamples where the property does not hold. Reachability properties do not hold if the set is empty.

Correctness property 1 Workers conform to the coordinator decision (either commit or abort). This correctness property should be verified for the basic 2PC protocol [15], as well as for the nested transaction model. The following two functions return `TRUE` for the paths, where the decision of the coordinator is not applied by the worker (negation of property 1).

```
both_commit p =          has_role "w" p                 &&
          has_state      (State "ST_COMMIT_LOG" "c") p  &&
          has_not_event  (Event "COMMIT" "c" "w") p}
both_abort p =           has_role "w" p                 &&
          has_state      (State "ST_AKN_ABORT" "c") p   &&
          has_not_event  (Event "ABORT" "c" "w") p
```

Two similar functions verify that the same property also holds for the sub-transaction coordinator (*cn*) and its corresponding worker.

Correctness property 2 Workers (and coordinators) reach exactly one of the two possible decisions, *abort* or *commit*. The following function verified property 2 for the worker role. In a similar manner, the same property was also verified for the coordinators (*c* and *cn*).

```
worker_finished p =      has_role "w" p                 &&
          has_event      (Event "COMMIT" "c" "w") p     &&
          has_event      (Event "ABORT" "c" "w") p
```

Correctness property 3 If the top-level transaction aborts, sub-transactions are also aborted [5].

```
top_nested_abort p =     has_role "cn" p                &&
          has_event      (Event "ABORT" "c" "cn")  p    &&
          has_not_state  (State "ST_AKN_ABORT" "cn") p
```

Correctness property 4 The top-level transaction commits, after sub-transactions are completed (committed or aborted) [5]. In a path where the property is violated the coordinator of the sub-transaction would decide to commit or abort before the top-level coordinator. The following function returns `TRUE` for such paths.

```
top_after_nested p =      has_role "cn" p                 &&
          (before_st      (State "ST_AKN_COMMIT" "c")
                          (State "ST_AKN_COMMIT" "cn") p
          || before_st    (State "ST_AKN_COMMIT" "c")
                          (State "ST_AKN_ABORT"  "cn") p)
```

Correctness property 5 A sub-transaction can abort independently from the top-level transaction (top-level can commit even when sub-transaction aborts) [5]. This is a reachability property and holds if there is at least one path for which the following function returns `TRUE`.

```
independent_abort p =     has_role "cn" p                 &&
          has_state       (State "ST_AKN_COMMIT" "c") p   &&
          has_state       (State "ST_AKN_ABORT" "cn") p
```

Correctness property 6 All roles eventually commit or abort the transaction, even in the presence of communication or system failures (liveness property).

```
coord_commit_abort p =    has_not_state (State "ST_AKN_ABORT"  "c") p &&
                          has_not_state (State "ST_AKN_COMMIT" "c") p
```

7 CODE GENERATION

Although code generation is specific to the ACID Sim Tools, the same principles apply in any transaction management system. Transactions are monitored by the generated code in terms of the states traversed in the role state machines. Upon a state transition the associated operations are invoked. In ACID Sim Tools, these operations use simulated services like log recording in stable storage, handling of lock requests for concurrency control and so on.

Upon a transaction request, a new instance of the class `CoordinatorTransactionFsm` is created for monitoring the execution of the transaction. This instance has access to the transaction structure, the number of transaction managers expected to acknowledge the decision and two sets of identifiers: the first contains those that have acknowledged the decision and the second those that voted for the transaction outcome. The participants of the newly arrived transactions that are not coordinators create an instance of `TransactionFsm` that stores runtime information relevant to the worker role.

```
for each role r in the model do
  if r is the event receiver then
    for each event ev that can be accepted by r do
     if ev is the received event then
      if transition from the current state s for event ev is deterministic then
         set the current state of the role to the state defined by the transition
         invoke the operations associated with the transition
      else
       let s‘ be the result of the user-provided method that resolves non-determinism
       for each state s‘‘ accessible by ev from s do
        if s‘‘ = s‘ then
         set the current state of the role to the state defined by the transition
         invoke the operations associated with the transition
```

Figure 4: Code template for transaction execution

When there are two or more transitions from a given state triggered by the same event, an appropriate handler method is expected to return the next state, thus resolving non–determinism. The method name is automatically created by concatenating the name of the outgoing state with the event name and accepts as parameter either a `CoordinatorTransactionFsm` or a `TransactionFsm`. Information stored in the monitor instance is used by the method to determines the next role state. For example, for the non–deterministic transitions for state `ST_VOTES` and event `VOTED`, the generated handler method is called `resolveVotesVoted`. The invoked method inspects the number of participants expected to vote (stored in the monitor instance) and decides if all participants have voted or there are still pending votes, in which case the role state machine remains in the same state.

Fig. 4 introduces the pseudocode of the template used for generating the implementation of the verified transaction model. The generated code handles transaction events for all roles and invokes the operations associated with state transitions as well as the user–defined methods that resolve non-determinism.

8 RELATED WORK

There are several proposals for the specification of (advanced) transaction models and the verification of their properties, although we are not aware of works that combine verification with automatic code generation. Most existing approaches presume knowledge of a proprietary formal language and they support a proof-theoretic verification instead of our algorithmic model checking approach which is based on an intuitive state machine definition.

In [14], communicating I/O automata are used for specifying the behavior of the entities involved in nested transaction processing. Correctness properties are established by mathematical proofs. However, the article focuses on concurrency control and serializability properties, which are not addressed in present work.

From this point of view our approach is complementary to the one presented in [14].

In [10] the authors apply the Temporal Logic of Actions (TLA) [13] for standardizing the Web Services Atomic Transaction protocol (WS-AT) that is based on the well-known two-phase commit (2PC) protocol with some non-standard features. Correctness of the protocol is expressed by invariance of a state predicate, which is verified by the TLC model checker. Model implementation is not a concern, since the primary goal is precision in the standard specifying the complete behavior of the WS-AT protocol.

ACTA [5] is a framework used to specify and reason about the effects of transactions on objects and the interactions between them. In [6] the properties of nested transactions are investigated, as well as other advanced transaction models.

From the aforementioned related work, our proposal resembles the TLA based approach that also targets verification of safety properties. However, in our case one can check liveness and reachability properties as well. If the cost for conducting formal proofs of advanced transaction models can be afforded, then the broadest in scope in terms of the properties proved reasoning alternative is the one offered by ACTA.

9 CONCLUSION

There is a growing interest for developing advanced transaction models that relax the typical ACID properties of the 2PC protocol. We introduced a model-driven development approach that incorporates intuitive specification and algorithmic verification of transaction properties, as well as automatic generation of event handling code in a transaction model implementation. We justified the need of non-determinism, when specifying a transaction model and we developed techniques that resolve it during model verification and code generation.

The proposed approach was successfully applied to the verification of the 2PC protocol [15] and its implementation in the ACID Sim Tools simulator [16], for studying the performance and recovery tradeoffs in transaction processing architecture [17, 18]. In this article, we applied the approach to the development of an implementation of the nested transaction model. This gave us the opportunity to study a wider range of correctness properties that were grouped into three categories namely safety, reachability and liveness.

Also, due to the fact that the studied model is more complex than the one developed for the 2PC protocol, we realized the need to impose constraints on event generation by defining conditions over the execution history. We call these conditions guards and we will provide syntactic support in the next version of the ACID Model Checker and Code Generator. Another future research goal is the development of the presented model checking algorithm towards integrating advanced state space reduction techniques.

Aknowledgments: We acknowledge the anonymous referees for their helpful comments for improving this article.

References

[1] ACID Model Checker & Code Generator.
http://mathind.csd.auth.gr/acid/html/ [27 September 2010].

[2] Baier, C, Katoen, J-P. *Principles of model checking* The MIT Press, Cambridge, MA, 2008.

[3] Berard B. et al. *Systems and software verification* Springer, Berlin, Heidelberg, 2001.

[4] Chrysanthis PK, Pitoura E. Data broadcasting, caching and replication in mobile computing. In *Encyclopedia of Database Systems* 2009; 557–561.

[5] Chrysanthis, P, Ramamritham, K. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the the 1990 ACM SIGMOD International Conference on Management of Data,* ACM Press, NY, 1990; 194–203.

[6] Chrysanthis P, Ramamritham K. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems* 1994; **19**(3):450–491.

[7] Clarke EM, Grumberg O, Peled DA. *Model Checking* The MIT Press, Cambridge, Massachusetts, 2000.

[8] Eliot J, Moss B. *Nested transactions: and approach to reliable distributed computing* Tech. Report MIT/LCS/TR-260, Massachusetts Institute of Technology, 1981.

[9] Holzmann GJ. *The SPIN model checker primer and reference manual* Addison-Wesley, Boston, MA, 2003.

[10] Johnson JE, Langworthy DE, Lamport L, Vogt F. Formal specification of a web services protocol. *Electronic Notes in Theoretical Computer Science* 2004; **105**:147–158.

[11] Jones P, Simon L et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming* 2003; **13**(1):0–255.

[12] Katsaros P, Angelis L, Lazos C. Performance and effectiveness trade-off for checkpointing in fault tolerant distributed systems. *Concurrency and Computation: Practice and Experience* 2007; **19**(1):37–63.

[13] Lamport, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 1994; **16**(3):872–923.

[14] Lynch NA, Merritt M. Introduction to the theory of nested transactions. In *Proceedings of the International Conference on Database Theory,* Vol 243, LNCS. Springer, London, UK, 1986; 278–305.

[15] Mentis A, Katsaros P. The ACID model checker and code generator for transaction processing. In *Proceedings of the 2009 High Performance Computing & Simulation Conference (HPCS),* IEEE Press, 2009; 138–144.

[16] Mentis A, Katsaros P, Angelis L. ACID Sim Tools: A simulation framework for distributed transaction processing architectures. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SimulationWorks Industry Track),* ICST, 2008.

[17] Mentis A, Katsaros P, Angelis L. Synthetic metrics for evaluating performance of software architectures with complex trade-offs. In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA),* IEEE Computer Society, Los Alamitos, CA, 2009; 237–242.

[18] Mentis A, Katsaros P, Angelis L, Kakarontzas, G. Quantification of interacting runtime qualities in software architectures: insights from transaction processing in client-server architectures. *Information and Software Technology* 2010; **52**(12):1331–1345.

[19] Serrano-Alvarado P, Roncancio C, Adiba M. A survey of mobile transactions. *Distributed and Parallel Databases* 2004; **16**(2):192–230.

[20] Stahl T, Volter M. *Model-driven software development: Technology, Engineering, Management* Wiley, Chichester, UK, 2005.

[21] Veijalainen J, Terziyan V, Tirri H. Transaction management for m-commerce at a mobile terminal. *Electronic Commerce Research and Applications* 2006; **5**(3):229–245.

[22] Wang T, Vonk J, Kratz, B, Grefen P. A survey on the history of transaction management: from flat to grid transactions. *Distributed and Parallel Databases* 2008; **23**(3):235–270.

[23] Weikum G, Vossen G. *Transactional Information Systems* Morgan Kaufmann Pub., San Francisco, 2002.