An Abstract Transaction Operator Serializing Concurrent Program Executions

Egon Börger[1] and Klaus-Dieter Schewe[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] Software Competence Centre Hagenberg, A-4232 Hagenberg, Austria
klaus-dieter.schewe@scch.at

Abstract. We define an abstract transaction controller and an operator which when applied to concurrent programs turns their behavior with respect to some abstract termination criterion into a transactional behavior. We prove that concurrent runs under the transaction controller are serialisable. For the sake of generality we specify the transaction controller and the operator in terms of Abstract State Machines.

1 Introduction

Transactions are a common means to control concurrent access to shared locations and to avoid that values stored at these locations are changed almost randomly. In general, a *transaction controller* interacts with concurrently running programs (read: sequential components of an asynchronous system), controls whether access to a shared location is granted or not, and thus ensures a certain form of consistency for the shared locations. A commonly accepted criterium for consistency is that the joint behavior of all transactions with respect to the shared locations is equivalent to a serial execution of the transactions. Thus, each transaction can be specified independently from the transaction controller, as if it had exclusive access to the shared locations.

As location sharing is common in concurrent systems, it would be cumbersome, if transactional behavior had to be specified each time. It would be desirable, if transactional behavior could be ignored in the specification and a transaction controller could simply be "plugged in".

Therefore, the goal of this paper is to define a scheme which allows one to turn the behavior of any component M of a given asynchronous system $\mathcal{M}$, i.e. of a set of concurrently working sequential programs M, into a transactional behavior. This involves the definition of a transaction controller TaCtl and of an operator $TA(M, \textsc{TaCtl})$ which turns the behavior of M into a transactional behavior under the control of TaCtl. For the sake of generality we define the operator and the controller in terms of Abstract State Machines (ASMs), which can be read and undestood as pseudo-code but come with an underlying precise semantics for which we refer the interested reader to [4].

In this paper we concentrate on transaction controllers that employ locking strategies such as the common two-phase locking protocol (2PL). That is, each transaction first has to acquire a (read- or write-) lock for a shared location, before the access is granted. Locks are released after the transaction has successfully committed and no more access to the shared locations is necessary. There are of course other approaches to transaction handling, see e.g. [5,10,12,13] and the extensive literature there covering classical transaction control for flat transactions, timestamp-based, optimistic and hybrid transaction control protocols, as well as non-flat transaction models such as sagas and multi-level transactions.

The transaction controller performs the lock handling, the deadlock detection and handling, the recovery mechanism (for partial recovery) and the commit of single machines. Thus we define it as consisting of four component specified in Sect. 3.

```
TaCtl =
  LockHandler
  DeadlockHandler
  Recovery
  Commit
```

The operator $\mathit{TA}(M, \textsc{TaCtl})$ transforms the components M of a concurrent system (read: asynchronous ASM) $\mathcal{M} = (M_i)_{i \in I}$ into components of a concurrent system $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ where each $\mathit{TA}(M_i, \textsc{TaCtl})$ runs as transaction under the control of TaCtl:

$$\mathit{TA}(\mathcal{M}, \textsc{TaCtl}) = ((\mathit{TA}(M_i, \textsc{TaCtl}))_{i \in I}, \textsc{TaCtl})$$

We prove in Sect. 4 that if all monitored or shared locations of any M_i are output or controlled locations of some other M_j and all output locations of any M_i are monitored or shared locations of some other M_j (closed system assumption)[1], each terminating run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is equivalent to a serialization of the terminating M_i-runs, namely the M_{i_1}-run followed by the M_{i_2}-run etc., where M_{i_j} is the j-th machine of $\mathcal{M}$ which performs a commit in the $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ run. To simplify the exposition (i.e. the formulation of statement and proof of the theorem) we only consider machine steps which take place under the transaction control, in other words we abstract from any step M_i makes before being Inserted into or after being Deleted from the set *TransAct* of machines which currently run under the control of TaCtl.

2 The Transaction Operator $\mathit{TA}(M, \textsc{TaCtl})$

The transaction controller TaCtl keeps a dynamic set *TransAct* of those machines M whose runs it currently has to supervise to perform in a transactional manner until M has *Terminated* its transactional behavior (so that it can Commit it).[2] To turn the behavior of a machine M into a transactional one, first

[1] This assumption means that the environment is assumed to be one of the component machines.

[2] In this paper we deliberately keep the termination criterion abstract so that it can be refined in different ways for different transaction instances.

of all M has to register itself with the controller TaCtl, read: to be inserted into its set of currently to be handled *TransAct*ions. To Undo as part of a recovery some steps M made already during the given transactional run segment of M, a last-in first-out queue $history(M)$ is needed which keeps track of the states the transactional run goes through; when M enters the set *TransAct* the $history(M)$ has to be initialized (to the empty queue).

The crucial transactional feature is that each non private (i.e. shared or monitored or output) location l a machine M needs to read or write for performing a step has to be $LockedBy(M)$ for this purpose; M tries to obtain such locks by calling the LockHandler. In case no *newLocks* are needed by M in its *currState* or the needed *newLocks* can be *Granted* by the LockHandler, M performs its next step; in addition, for a possible future recovery, the machine has to Record in its $history(M)$ the current values of those locations which are (possibly over-) written by this M-step together with the obtained *newLocks*. Then M continues its transactional behavior until it is *Terminated*. In case the needed *newLocks* are *Refused*, namely because another machine N in *TransAct* for some needed l has $W - Locked(l, N)$ or (in case M wants a W-(rite)Lock) has $R - Locked(l, N)$, M has to *Wait* for N; in fact it continues its transactional behavior by calling again the LockHandler for the needed *newLocks*—until these locations l locked by N are unlocked when N's transactional behavior is Commited, whereafter a new request for these locks this time may be *Granted* to M.

As a consequence *Deadlock*s may occur, namely when a cycle occurs in the transitive closure $Wait^*$ of the *Wait* relation, restricted to non *Victim*s. To resolve such deadlocks the DeadlockHandler component of TaCtl chooses some machines as *Victim*s for a recovery.[3] After a victimized machine M is *Recovered* by the Recovery component of TaCtl, so that M can exit its *waitForRecovery* state, it continues its transactional behavior.

This explains the following definition of $TA(M, \text{TaCtl})$ as a control state ASM, i.e. an ASM with a top level Finite State Machine control structure. We formulate it by the flowchart diagram of Fig. 1, which has a precise control state ASM semantics (see the definition in [4, Ch.2.2.6]). The components for the recovery feature are highlighted in the flowchart by a colouring that differs from that of the other components. The macros which appear in Fig. 1 and the components of TaCtl are defined below.

The predicate $NewLocksNeededBy(M)$ holds if in the current state of M at least one of two cases happens:[4] either M to perform its step in this state reads some shared or monitored location which is not yet $LockedBy(M)$ or M writes some shared or output location which is not yet $LockedBy(M)$ for writing.

[3] To simplify the serializability proof in Sect.3 and without loss of generality we define a reaction of machines M to their victimization only when they are in $ctl\_state(M) =$ TA-*ctl* (not in $ctl\_state(M) = waitForLocks$). This is to guarantee that no locks are *Granted* to a machine as long as it does *waitForRecovery*.

[4] See [4, Ch.2.2.3] for the classification of locations and functions.

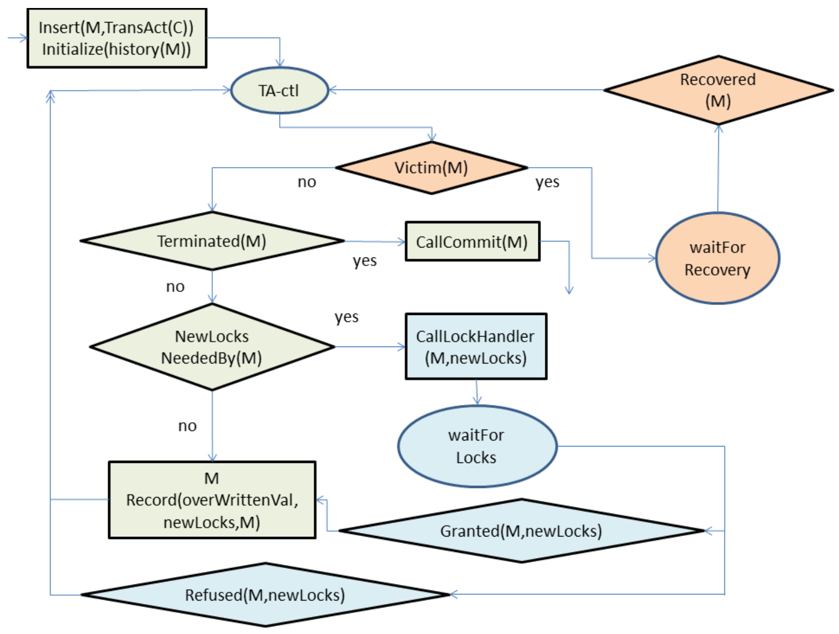

Fig. 1. TA(M,C)

A location can be $\mathit{LockedBy}(M)$ for reading ($\mathit{R\text{-}Locked}(l,M)$) or for writing ($\mathit{W\text{-}Locked}(l,M)$). Formally:

$$
\begin{aligned}
&\mathit{NewLocksNeededBy}(M) = \\
&\quad \mathit{newLocks}(M, \mathit{currState}(M))^{5} \neq (\emptyset, \emptyset) \\
&\mathit{newLocks}(M, \mathit{currState}(M))^{6} = (\mathit{R\text{-}Loc}, \mathit{W\text{-}Loc}) \\
&\quad \textbf{where} \\
&\quad \mathit{R\text{-}Loc} = \mathit{ReadLoc}(M, \mathit{currState}(M)) \cap (\mathit{SharedLoc}(M) \cup \mathit{MonitoredLoc}(M)) \\
&\qquad\qquad \cap \overline{\mathit{LockedBy}(M)}^{7} \\
&\quad \mathit{W\text{-}Loc} = \mathit{WriteLoc}(M, \mathit{currState}(M)) \cap (\mathit{SharedLoc}(M) \cup \mathit{OutputLoc}(M)) \\
&\qquad\qquad \cap \overline{\mathit{W\text{-}LockedBy}(M)} \\
&\mathit{LockedBy}(M) = \{l \mid \mathit{R\text{-}Locked}(l, M) \textbf{ or } \mathit{W\text{-}Locked}(l, M)\} \\
&\mathit{W\text{-}LockedBy}(M) = \{l \mid \mathit{W\text{-}Locked}(l, M)\}
\end{aligned}
$$

[5] For layout reasons we omit in Fig.1 the arguments of the functions *newLocks* and *overWrittenVal*.

[6] By the second argument $\mathit{currState}(M)$ of *newLocks* (and below of *overWrittenVal*) we indicate that this function of M is a dynamic function which is evaluated in each state of M, namely by computing in this state the sets $\mathit{ReadLoc}(M)$ and $\mathit{WriteLoc}(M)$; see Sect. 4 for the detailed definition.

[7] By $\overline{X}$ we denote the complement of X.

The *overWrittenVal*ues are the *currState*(M)-values (retrieved by the *eval*-function) of those shared or output locations (f, *args*) which are written by M in its *currState*(M). To RECORD the set of these values together with the obtained *newLocks* means to append the pair of these two sets to the *history* queue of M from where upon recovery the values and the locks can be retrieved.

```
$overWrittenVal(M, currState(M)) = \{((f, args), val) \mid$
  $(f, args) \in WriteLoc(M, currState(M)) \cap (SharedLoc(M) \cup OutputLoc(M))$
  **and** $val = eval(f(args), currState(M))\}$
$\textsc{Record}(valSet, lockSet, M) = \textsc{Append}((valSet, lockSet), history(M))$
```

To CALLLOCKHANDLER for the *newLocks* requested by M in its *currState*(M) means to INSERT(M, *newLocks*) into the LOCKHANDLER's set of to be handled *LockRequest*s. Similarly we let CALLCOMMIT(M) stand for insertion of M into a set *CommitRequest* of the COMMIT component.

```
$\textsc{CallLockHandler}(M, L) = \textsc{Insert}((M, L), LockRequest)$
$\textsc{CallCommit}(M) = \textsc{Insert}(M, CommitRequest)$
```

3 The Transaction Controller Components

A CALLCOMMIT(M) by machine M enables the COMMIT component. Using the **choose** operator we leave the order in which the *CommitRequest*s are handled refinable by different instantiations of TACTL.

COMMITing M means to UNLOCK all locations l that are *LockedBy*(M). Note that each lock obtained by M remains with M until the end of M's transactional behavior. Since M performs a CALLCOMMIT(M) when it has *Terminated* its transactional computation, nothing more has to be done to COMMIT M besides deleting M from the sets of *CommitRequest*s and still to be handled *TransAct*ions.[8]

Note that the locations *R-Locked*(l, M) and *W-Locked*(l, M) are shared by the COMMIT, LOCKHANDLER and RECOVERY components, but these components never have the same M simultaneously in their request resp. *Victim* set since when machine M has performed a CALLCOMMIT(M), it has *Terminated* its transactional computation and does not participate any more in any $(M, L) \in$ *LockRequest* or *Victim*ization.

```
$\textsc{Commit} =$
  **if** $CommitRequest \neq \emptyset$ **then**
    **choose** $M \in CommitRequest$ $\textsc{Commit}(M)$
  **where**
    $\textsc{Commit}(M) =$
      **forall** $l \in LockedBy(M)$ $\textsc{Unlock}(l, M)$
```

[8] We omit clearing the *history*(M) queue since it is initialized when M is inserted into *TransAct*(TACTL).

```
        DELETE($M$, $CommitRequest$)
        DELETE($M$, $TransAct$)
      UNLOCK($l, M$) =
        if $R\text{-}Locked(l, M)$ then $R\text{-}Locked(l, M) := false$
        if $W\text{-}Locked(l, M)$ then $W\text{-}Locked(l, M) := false$
```

As for COMMIT also for the LOCKHANDLER we use the **choose** operator to leave the order in which the *LockRequest*s are handled refinable by different instantiations of TACTL.

The strategy we adopt for lock handling is to refuse all locks for locations requested by M if at least one of the following two cases happens:

- some of the requested locations is *W-Locked* by another transactional machine $N \in TransAct$,
- some of the requested locations is a *WriteLoc*ation that is *R-Locked* by another transactional machine $N \in TransAct$.

This definition, which is specified below by the predicate *CannotBeGranted*, implies that multiple transactions may simultaneoulsy have a *R-Lock* on some location. To REFUSEREQUESTEDLOCKS it suffices to set the communication interface *Refused* of $TA(M, \text{TACTL})$; this makes M *Wait* for each location l that is $W\text{-}Locked(l, N)$ and for each *WriteLoc*ation that is $R\text{-}Locked(l, N)$ by some other transactional component machine $N \in TransAct$.

```
LOCKHANDLER =
  if $LockRequest \neq \emptyset$ then
    choose $(M, L) \in LockRequest$
      HANDLELOCKREQUEST($M, L$)
  where
    HANDLELOCKREQUEST($M, L$) =
      if $CannotBeGranted(M, L)$
        then  REFUSEREQUESTEDLOCKS($M, L$)
        else  GRANTREQUESTEDLOCKS($M, L$)
      DELETE($(M, L), LockRequest$)
    $CannotBeGranted(M, L)$ =
      let $L = (R\text{-}Loc, W\text{-}Loc), Loc = R\text{-}Loc \cup W\text{-}Loc$
        forsome $l \in Loc$  forsome $N \in TransAct \setminus \{M\}$
          $W\text{-}Locked(l, N)$ or
            ($l \in W\text{-}Loc$ and $R\text{-}Locked(l, N)$)
      REFUSEREQUESTEDLOCKS($M, L$) = ($Refused(M, L) := true$)
      GRANTREQUESTEDLOCKS($M, L$) =
        let $L = (R\text{-}Loc, W\text{-}Loc)$
          forall $l \in R\text{-}Loc$  ($R\text{-}Locked(l, M) := true$)
          forall $l \in W\text{-}Loc$  ($W\text{-}Locked(l, M) := true$)
        $Granted(M, L) := true$
```

A *Deadlock* originates if two machines are in a *Wait* cycle, otherwise stated if for some machine M the pair (M, M) is in the transitive (not reflexive) closure $Wait^*$ of *Wait* (restricted to non *Victims*). In this case the DEADLOCKHANDLER selects for recovery a (typically minimal) subset *vict* of *WaitingTransAct*ions—they are *Victim*ized to *waitForRecovery*, in which mode (control state) they are backtracked until they become *Recovered*—such that the set $WaitingTransAct \setminus vict$ of remaining transactions is deadlock free. The selection criteria are intrinsically specific for particular transaction controllers, driving a usually rather complex selection algorithm in terms of number of conflict partners, priorities, waiting time, etc. In this paper we leave their specification for TACTL abstract (read: refinable in different directions) by assuming a highly complex selection function *deadlockResolution* to perform the deadlock resolution among the non-victims in one step.

```
DEADLOCKHANDLER =
  if $Deadlock$ then
    let $vict = deadlockResolution(WaitingTransAct \cap \overline{Victim})$
      forall $M \in vict$ $(Victim(M) := true)$
  where
    $Deadlock =$ forsome $M \in WaitingTransAct$
      $Wait^*(M, M)$ and not $Victim(M)$
    $WaitingTransAct =$
      $\{M \in TransAct \mid$forsome $N \in TransAct$ $Wait(M, N)\}$
    $Wait^* =$
      $\{(M, M) \mid$forsome $M_1, \ldots, M_n \in TransAct \cap \overline{Victim}$  $Wait(M, M_1)$
        and forall $1 \leq i < n$ $Wait(M_i, M_{i+1})$ and $Wait(M_n, M)\}$
    $Wait(M, N) =$ forsome $l$ $Wait(M, l, N)$
    $Wait(M, l, N) =$
      $l \in newLocks(M, currState(M))$ and $N \in TransAct \setminus \{M\}$ and
      $W\text{-}Locked(l, N)$ or $(l \in W\text{-}Loc$ and $R\text{-}Locked(l, N))$
        where $newLocks(M, currState(M)) = (R\text{-}Loc, W\text{-}Loc)$
```

As for COMMIT and LOCKHANDLER also for the RECOVERY component we use the **choose** operator to leave the order in which the *Victims* are chosen for recovery refinable by different instantiations of TACTL. To be *Recovered* a machine M is backtracked by UNDO(M) steps until there is no $DeadlockWith(M)$ any more, in which case it is deleted from the set of *Victims*, so that be definition it is *Recovered*. This happens at the latest when $history(M)$ has become empty.

```
RECOVERY =
  if $Victim \neq \emptyset$ then
    choose $M \in Victim$ TRYTORECOVER$(M)$
where
  TRYTORECOVER$(M) =$
    if $NoDeadlockWith(M)$ then $Victim(M) := false$
    else $\textsc{Undo}(M)$
$\mathit{Recovered} =$
  $\{M \mid \mathit{ctl\text{-}state}(M) = \mathit{waitForRecovery}$ and $M \notin \mathit{Victim}\}$
$\mathit{NoDeadlockWith}(M) = (M, M) \notin \mathit{Wait}^*$
$\textsc{Undo}(M) =$
  let $(\mathit{ValSet}, \mathit{LockSet}) = \mathit{youngest}(\mathit{history}(M))$
    $\textsc{Restore}(\mathit{ValSet})$
    $\textsc{Release}(\mathit{LockSet})$
    $\textsc{Delete}((\mathit{ValSet}, \mathit{LockSet}), \mathit{history}(M))$
where
  $\textsc{Restore}(V) =$
    forall $((f, \mathit{args}), v) \in V$ $f(\mathit{args}) := v$
  $\textsc{Release}(L) =$
    let $L = (\mathit{R\text{-}Loc}, \mathit{W\text{-}Loc})$
      forall $l \in \mathit{Loc} = \mathit{R\text{-}Loc} \cup \mathit{W\text{-}Loc}$ $\textsc{Unlock}(l, M)$
```

Note that in our description of the DeadlockHandler and the (partial) Recovery we deliberately left the strategy for victim seclection and Undo abstract leaving fairness considerations to be discussed elsewhere. It is clear that if always the same victim is selected for partial recovery, the same deadlocks may be created again and again. However, it is well known that fairness can be achieved by choosing an appropriate victim selection strategy.

4 Correctness Theorem

In this section we state and prove that $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ satisfies the desired transactional behavior property, namely that each run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is equivalent to a serial run. Before doing this we have to make precise what a *serial* multi-agent ASM run is and what *equivalence* of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ runs means in the general multi-agent ASM framework.

Definition of run equivalence Let $S_0, S_1, S_2, \ldots$ be a (finite or infinite) run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$. In general we may assume that TaCtl runs forever, whereas each machine $M \in \mathcal{M}$ running as transaction will be terminated at some time – at least after commit M will only change values of non-shared and non-output locations[9]. For $i = 0, 1, 2, \ldots$ let $\varDelta_i$ denote the unique, consistent update set defining the transition from S_i to S_{i+1} [11, p.89]. By definition of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ the update set is the union of the update sets of the agents executing $M \in \mathcal{M}$ resp. TaCtl:

$$\varDelta_i = \bigcup_{M \in \mathcal{M}} \varDelta_i(M) \cup \varDelta_i(\textsc{TaCtl}).$$

[9] It is possible that one ASM M enters several times as a transaction controlled by TaCtl. However, in this case each of these registrations will be counted as a separate transaction, i.e. as different ASMs in $\mathcal{M}$.

$\Delta_i(M)$ contains the updates defined by the ASM $\mathit{TA}(M, \textsc{TaCtl})$ in state S_i[10] and $\Delta_i(\textsc{TaCtl})$ contains the updates by the transaction controller in this state. The sequence of update sets $\Delta_0(M)$, $\Delta_1(M)$, $\Delta_2(M)$, ... will be called the *schedule* of M (for the given transactional run).

To generalise for transactional ASM runs the equivalence of transaction schedules known from database systems [5, p.621ff.] we now define two *cleansing operations* for ASM schedules. By the first one (i) we eliminate all (in particular unsuccessful-lock-request) computation segments which are without proper M-updates; by the second one (ii) we eliminate all M-steps which are related to a later $\textsc{Undo}(M)$ step by the $\textsc{Recovery}$ component:

(i) Delete from the schedule of M each $\Delta_i(M)$ where one of the following two properties holds:

- $\Delta_i(M) = \emptyset$ (M contributes no update to S_i),
- $\Delta_i(M)$ belongs to a step of an M-computation segment where M in its $\mathit{ctl\_state}(M)$ =TA-*ctl* does $\textsc{CallLockHandler}(M, \mathit{newLocks})$ and in its next step moves from *waitForLocks* back to control state TA$-$*ctl* because the $\textsc{LockHandler}$ $\mathit{Refused}(M, \mathit{newLocks})$.[11]

In such computation steps M makes no proper update.

(ii) Repeate choosing from the schedule of M a pair $\Delta_j(M)$ with later $\Delta_{j'}(M)$ ($j < j'$) which belong to the first resp. second of two consecutive M-Recovery steps defined as follows:

- a (say M-RecoveryEntry) step whereby M in state S_j moves from TA-*ctl* to *waitForRecovery* because it became a *Victim*,
- the next M-step (say M-RecoveryExit) whereby M in state $S_{j'}$ moves back to control state TA-*ctl* because it has been *Recovered*.

In these two M-Recovery steps M makes no proper update. Delete:

(a) $\Delta_j(M)$ and $\Delta_{j'}(M)$,
(b) the $((\mathit{Victim}, M), \mathit{true})$ update from the corresponding $\Delta_t(\textsc{TaCtl})$ ($t < j$) which in state S_j triggered the M-RecoveryEntry,
(c) $\textsc{TryToRecover}(M)$-updates in any $\Delta_{i+k}(\textsc{TaCtl})$ between the considered M-RecoveryEntry and M-RecoveryExit step ($i < j < i + k < j'$),
(d) each $\Delta_{i'}(M)$ belonging to the M-computation segment from TA-*ctl* back to TA-*ctl* which contains the proper M-step in S_i that is UNDOne in S_{i+k} by the considered $\textsc{TryToRecover}(M)$ step; besides control state and $\textsc{Record}$ updates these $\Delta_{i'}(M)$ contain updates (ℓ, v) with

10 We use the shorthand notation $\Delta_i(M)$ to denote $\Delta_i(\mathit{TA}(M, \textsc{TaCtl}))$; in other words we speak about steps and updates of M also when they really are done by $\mathit{TA}(M, \textsc{TaCtl})$. Mainly this is about transitions between the control states, namely TA-*ctl*, *waitForLocks*, *waitForRecovery* (see Fig.1), which are performed during the run of M under the control of the transaction controller $\textsc{TaCtl}$. When we want to name an original update of M (not one of the updates of $\mathit{ctl\_state}(M)$ or of the $\textsc{Record}$ component) we call it a proper M-update.

11 Note that by eliminating this $\textsc{CallLockHandler}(M, L)$ step also the corresponding $\textsc{LockHandler}$ step $\textsc{HandleLockRequest}(M, L)$ disappears in the run.

$\ell = (f, (val_{S_i}(t_1), \dots, val_{S_i}(t_n)))$ where the corresponding UNDO updates are $(\ell, val_{S_i}(f(t_1, \dots, t_n))) \in \Delta_{i+k}(\text{TACTL})$,

(e) the HANDLELOCKREQUEST(M, *newLocks*)-updates in $\Delta_{l'}(\text{TACTL})$ corresponding to M's CALLLOCKHANDLER step (if any: in case *newLocks* are needed for the proper M-step in S_i) in state S_l ($l < l' < i$).

The sequence $\Delta_{i_1}(M), \Delta_{i_2}(M), \dots$ with $i_1 < i_2 < \dots$ resulting from the application of the two cleansing operations as long as possible – note that confluence is obvious, so the sequence is uniquely defined – will be called the *cleansed schedule* of M (for the given run).

Before defining the equivalence of transactional ASM runs let us remark that $TA(\mathcal{M}, \text{TACTL})$ has indeed several runs, even for the same initial state S_0. This is due to the fact that a lot of non-determinism is involved in the definition of this ASM. First, the submachines of TACTL are non-deterministic:

- In case several machines $M, M' \in \mathcal{M}$ request conflicting locks at the same time, the LOCKHANDLER can only grant the requested locks for one of these machines.
- Commit requests are executed in random order by the COMMIT submachine.
- The submachine DEADLOCKHANDLER chooses a set of victims, and this selection has been deliberately left abstract.
- The RECOVERY submachine chooses in each step a victim M, for which the last step will be undone by restoring previous values at updated locations and releasing corresponding locks.

Second, the specification of $TA(\mathcal{M}, \text{TACTL})$ leaves deliberately open, when a machine $M \in \mathcal{M}$ will be started, i.e., register as a transaction in *TransAct* to be controlled by TACTL. This is in line with the common view that transactions $M \in \mathcal{M}$ can register at any time to the transaction controller TACTL and will remain under its control until they commit.

Definition 1. Two runs $S_0, S_1, S_2, \dots$ and $S'_0, S'_1, S'_2, \dots$ of $TA(\mathcal{M}, \text{TACTL})$ are *equivalent* iff for each $M \in \mathcal{M}$ the cleansed schedules $\Delta_{i_1}(M), \Delta_{i_2}(M), \dots$ and $\Delta'_{j_1}(M), \Delta'_{j_2}(M), \dots$ for the two runs are the same and the read locations and the values read by M in S_{i_k} and S'_{j_k} are the same.

That is, we consider runs to be equivalent, if all transactions $M \in \mathcal{M}$ read the same locations and see there the same values and perform the same updates in the same order disregarding waiting times and updates that are undone.

Definition of serializability Next we have to clarify our generalised notion of a serial run, for which we concentrate on committed transactions – transactions that have not yet committed can still undo their updates, so they must be left out of consideration[12]. We need a definition of the read- and write-locations of

[12] Alternatively, we could concentrate on complete, infinite runs, in which only committed transactions occur, as eventually every transaction will commit – provided that fairness can be achieved.

M in a state S, i.e. $\mathit{ReadLoc}(M, S)$ and $\mathit{WriteLoc}(M, S)$ as used in the definition of $\mathit{newLocks}(M, S)$.

We define $\mathit{ReadLoc}(M, S) = \mathit{ReadLoc}(r, S)$ and analogously $\mathit{WriteLoc}(M, S) = \mathit{WriteLoc}(r, S)$, where r is the defining rule of the ASM M. Then we use structural induction according to the definition of ASM rules in [4, Table 2.2]. As an auxiliary concept we need to define inductively the read and write locations of terms and formulae. The definitions use an interpretation I of free variables which we suppress notationally (unless otherwise stated) and assume to be given with (as environment of) the state S. This allows us to write $\mathit{ReadLoc}(M, S)$, $\mathit{WriteLoc}(M, S)$ instead of $\mathit{ReadLoc}(M, S, I)$, $\mathit{ReadLoc}(M, S, I)$ respectively.

Read/Write Locations of Terms and Formulae. For state S let I be the given interpretation of the variables which may occur freely (in given terms or formulae). We write $\mathit{val}_S(\mathit{construct})$ for the evaluation of *construct* (a term or a formula) in state S (under the given interpretation I of free variables).

$$
\begin{array}{l}
\mathit{ReadLoc}(x, S) = \mathit{WriteLoc}(x, S) = \emptyset \text{ for variables } x \\
\mathit{ReadLoc}(f(t_1, \ldots, t_n), S) = \\
\quad \{(f, (\mathit{val}_S(t_1), \ldots, \mathit{val}_S(t_n)))\} \cup \bigcup_{1 \le i \le n} \mathit{ReadLoc}(t_i, S) \\
\mathit{WriteLoc}(f(t_1, \ldots, t_n), S) = \{(f, (\mathit{val}_S(t_1), \ldots, \mathit{val}_S(t_n)))\}
\end{array}
$$

Note that logical variables are not locations: they cannot be written and their values are not stored in a location but in the given interpretation I from where they can be retrieved.

We define $\mathit{WriteLoc}(\alpha, S) = \emptyset$ for every formula α because formulae are not locations one could write into. $\mathit{ReadLoc}(\alpha, S)$ for atomic formulae $P(t_1, \ldots, t_n)$ has to be defined as for terms with P playing the same role as a function symbol f. For propositional formulae one reads the locations of their subformulae. In the inductive step for quantified formulae $\mathit{domain}(S)$ denotes the superuniverse of S minus the Reserve set [4, Ch.2.4.4] and I_x^d the extension (or modification) of I where x is interpreted by a domain element d.

$$
\begin{array}{l}
\mathit{ReadLoc}(P(t_1, \ldots, t_n), S) = \\
\quad \{(P, (\mathit{val}_S(t_1), \ldots, \mathit{val}_S(t_n)))\} \cup \bigcup_{1 \le i \le n} \mathit{ReadLoc}(t_i, S) \\
\mathit{ReadLoc}(\neg\alpha) = \mathit{ReadLoc}(\alpha) \\
\mathit{ReadLoc}(\alpha_1 \wedge \alpha_2) = \mathit{ReadLoc}(\alpha_1) \cup \mathit{ReadLoc}(\alpha_2) \\
\mathit{ReadLoc}(\forall x \alpha, S, I) = \bigcup_{d \in \mathit{domain}(S)} \mathit{ReadLoc}(\alpha, S, I_x^d)
\end{array}
$$

Note that the values of the logical variables are not read from a location but from the modified state environment function I_x^d.

Read/Write Locations of ASM Rules.

$$
\begin{array}{l}
\mathit{ReadLoc}(\mathbf{skip}, S) = \mathit{WriteLoc}(\mathbf{skip}, S) = \emptyset \\
\mathit{ReadLoc}(t_1 := t_2, S) = \mathit{ReadLoc}(t_1, S) \cup \mathit{ReadLoc}(t_2, S) \\
\mathit{WriteLoc}(t_1 := t_2, S) = \mathit{WriteLoc}(t_1, S)
\end{array}
$$

$$\begin{aligned}
&\mathit{ReadLoc}(\mathbf{if}\ \alpha\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2, S) = \\
&\quad \mathit{ReadLoc}(\alpha, S) \cup \begin{cases} \mathit{ReadLoc}(r_1, S)\ \mathbf{if}\ \mathit{val}_S(\alpha) = \mathit{true} \\ \mathit{ReadLoc}(r_2, S)\ \mathbf{else} \end{cases} \\
&\mathit{WriteLoc}(\mathbf{if}\ \alpha\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2, S) = \begin{cases} \mathit{WriteLoc}(r_1, S)\ \mathbf{if}\ \mathit{val}_S(\alpha) = \mathit{true} \\ \mathit{WriteLoc}(r_2, S)\ \mathbf{else} \end{cases} \\
&\mathit{ReadLoc}(\mathbf{let}\ x = t\ \mathbf{in}\ r, S, I) = \mathit{ReadLoc}(t, S, I) \cup \mathit{ReadLoc}(r, S, I_x^{\mathit{val}_S(t)}) \\
&\mathit{WriteLoc}(\mathbf{let}\ x = t\ \mathbf{in}\ r, S, I) = \mathit{WriteLoc}(r, S, I_x^{\mathit{val}_S(t)})\ \text{// call by value} \\
&\mathit{ReadLoc}(\mathbf{forall}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = \\
&\quad \mathit{ReadLoc}(\forall x \alpha, S, I)\ \cup\ \bigcup_{a \in \mathit{range}(x,\alpha,S,I)} \mathit{ReadLoc}(r, S, I_x^a) \\
&\qquad \mathbf{where}\ \mathit{range}(x, \alpha, S, I) = \{d \in \mathit{domain}(S) \mid \mathit{val}_{S, I_x^d}(\alpha) = \mathit{true}\} \\
&\mathit{WriteLoc}(\mathbf{forall}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = \bigcup_{a \in \mathit{range}(x,\alpha,S,I)} \mathit{WriteLoc}(r, S, I_x^a)
\end{aligned}$$

In the following cases the same scheme applies to read and write locations:[13]

$$\begin{aligned}
&\mathit{Read}[\mathit{Write}]\mathit{Loc}(r_1\ \mathbf{par}\ r_2, S) = \\
&\quad \mathit{Read}[\mathit{Write}]\mathit{Loc}(r_1, S) \cup \mathit{Read}[\mathit{Write}]\mathit{Loc}(r_2, S) \\
&\mathit{Read}[\mathit{Write}]\mathit{Loc}(r(t_1, \ldots, t_n), S) = \mathit{Read}[\mathit{Write}]\mathit{Loc}(P(x_1/t_1, \ldots, x_n/t_n), S) \\
&\quad \mathbf{where}\ r(x_1, \ldots, x_n) = P\ \text{// call by reference} \\
&\mathit{Read}[\mathit{Write}]\mathit{Loc}(r_1\ \mathbf{seq}\ r_2, S, I) = \mathit{Read}[\mathit{Write}]\mathit{Loc}(r_1, S, I) \cup \\
&\quad \begin{cases} \mathit{Read}[\mathit{Write}]\mathit{Loc}(r_2, S + U, I)\ \mathbf{if}\ \mathit{yields}(r_1, S, I, U)\ \mathbf{and}\ \mathit{Consistent}(U) \\ \emptyset \qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{else} \end{cases}
\end{aligned}$$

For **choose** rules we have to define the read and write locations simultaneously to guarantee that the same instance satisfying the selection condition is chosen for defining the read and write locations of the rule body r:

```
**if** $range(x, \alpha, S, I) = \emptyset$ **then**
  $ReadLoc(\mathbf{choose}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = ReadLoc(\exists x \alpha, S, I)$
  $WriteLoc(\mathbf{choose}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = \emptyset$ // empty action
**else choose** $a \in range(x, \alpha, S, I)$
  $ReadLoc(\mathbf{choose}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) =$
    $ReadLoc(\exists x \alpha, S, I) \cup ReadLoc(r, S, I_x^a)$
  $WriteLoc(\mathbf{choose}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = WriteLoc(r, S, I_x^a)$
```

We say that M has or is committed (in state S_i, denoted $\mathit{Committed}(M, S_i)$) if step COMMIT($M$) has been performed (in state S_i).

Definition 2. A run of $\mathit{TA}(\mathcal{M}, \text{TACTL})$ is *serial* iff there is a total order $<$ on $\mathcal{M}$ such that the following two conditions are satisfied:

(i) If in a state M has committed, but M' has not, then $M < M'$ holds.
(ii) If M has committed in state S_i and $M < M'$ holds, then the cleansed schedule $\Delta_{j_1}(M'), \Delta_{j_2}(M'), \ldots$ of M' satisfies $i < j_1$.

That is, in a serial run all committed transactions are executed in a total order and are followed by the updates of transactions that did not yet commit.

[13] In $\mathit{yields}(r_1, S, I, U)$ U denotes the update set produced by rule r_1 in state S under I.

Definition 3. A run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is *serialisable* iff it is equivalent to a serial run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$.

Theorem 1. *Each run of* $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ *is serialisable.*

Proof. Let $S_0, S_1, S_2, \dots$ be a run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$. To construct an equivalent serial run let $M_1 \in \mathcal{M}$ be a machine that commits first in this run, i.e. $\mathit{Committed}(M, S_i)$ holds for some i and whenever $\mathit{Committed}(M, S_j)$ holds for some $M \in \mathcal{M}$, then $i \leq j$ holds. If there is more than one machine M_1 with this property, we randomly choose one of them.

Take the run of $\mathit{TA}(\{M_1\}, \textsc{TaCtl})$ starting in state S_0, say $S_0, S'_1, S'_2, \dots, S'_n$. As M_1 commits, this run is finite. M_1 has been DELETEd from *TransAct* and none of the TACTL components is triggered any more: neither COMMIT nor LOCKHANDLER because *CommitRequest* resp. *LockRequest* remain empty; not DEADLOCKHANDLER because *Deadlock* remains false since M_1 never *Wait*s for any machine; not RECOVERY because *Victim* remains empty. Note that in this run the schedule for M_1 is already cleansed.

We now define a run $S''_0, S''_1, S''_2, \dots$ (of $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$, as has to be shown) which starts in the final state $S'_n = S''_0$ of the $\mathit{TA}(\{M_1\}, \textsc{TaCtl})$ run and where we remove from the run defined by the cleansed schedules $\Delta_i(M)$ for the originally given run all updates made by steps of M_1 and all updates in TACTL steps which concern M_1. Let

$$\Delta''_i = \bigcup_{M \in \mathcal{M} - \{M_1\}} \Delta_i(M) \cup \{(\ell, v) \in \Delta_i(\textsc{TaCtl}) \mid (\ell, v) \text{ does not concern } M_1\}.$$

That is, in Δ''_i all updates are removed from the original run which are done by M_1—their effect is reflected already in the initial run segment from S_0 to S'_n—or are LOCKHANDLER updates involving a $\mathit{LockRequest}(M_1, L)$ or are $\mathit{Victim}(M_1) := \mathit{true}$ updates of the DEADLOCKHANDLER or are updates involving a $\textsc{TryToRecover}(M_1)$ step or are done by a step involving a $\textsc{Commit}(M_1)$.

Lemma 1. $S''_0, S''_1, S''_2, \dots$ *is a run of* $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$.

Lemma 2. *The run* $S_0, S'_1, S'_2, \dots, S'_n, S''_1, S''_2, \dots$ *of* $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ *is equivalent to the original run* $S_0, S_1, S_2, \dots$.

By induction hypothesis $S''_0, S''_1, S''_2, \dots$ is serialisable, so $S_0, S'_1, S'_2, \dots$ and thereby also $S_0, S_1, S_2, \dots$ is serialisable with $M_1 < M$ for all $M \in \mathcal{M} - \{M_1\}$.□

Proof.(Lemma 1) We first show that omitting in Δ''_i every update from $\Delta_i(\textsc{TaCtl})$ which concerns M_1 does not affect updates by TACTL in S''_i concerning $M \neq M_1$. In fact starting in the final M_1-state S''_0, $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$ makes no move with a $\mathit{Victim}(M_1) := \mathit{true}$ update and no move of $\textsc{Commit}(M_1)$ or $\textsc{HandleLockRequest}(M_1, L)$ or $\textsc{TryToRecover}(M_1)$

It remains to show that every M-step defined by $\Delta''_i(M)$ is a possible M-step in a $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$ run starting in S''_0. Since the considered M-schedule $\Delta_i(M)$ is cleansed, we only have to consider any proper update step of M in state S''_i (together with its preceding lock request step, if any). If in S''_i M uses *newLocks*, in the run by the cleansed schedules for the original run the locks must have been granted after the first COMMIT, which is done for M_1 before S''_0. Thus these locks are granted also in S''_i as part of a $\mathit{TA}(\mathcal{M} - \{M_1\}, \text{TACTL})$ run step. If no *newLocks* are needed, that proper M-step depends only on steps computed after S''_0 and thus is part of a $\mathit{TA}(\mathcal{M} - \{M_1\}, \text{TACTL})$ run step. □

Proof.(Lemma 2) The cleansed machine schedules in the two runs, the read locations and the values read there have to be shown to be the same. First consider any $M \neq M_1$. Since in the initial segment $S_0, S'_1, S'_2, \ldots, S'_n$ no such M makes any move so that its update sets in this computation segment are empty, in the cleansed schedule of M for the run $S_0, S'_1, S'_2, \ldots, S'_n, S''_1, S''_2, \ldots$ all these empty update sets disappear. Thus this cleansed schedule is the same as the cleansed schedule of M for the run $S'_n, S''_1, S''_2, \ldots$ and therefore by definition of $\Delta''_i(M) = \Delta_i(M)$ also for the original run $S_0, S_1, S_2, \ldots$ with same read locations and same values read there.

Now consider M_1, its schedule $\Delta_0(M_1), \Delta_1(M_1), \ldots$ for the run $S_0, S_1, S_2, \ldots$ and the corresponding cleansed schedule $\Delta_{i_0}(M_1), \Delta_{i_1}(M_1), \Delta_{i_2}(M_1), \ldots$. We proceed by induction on the cleansed schedule steps of M_1. When M_1 makes its first step using the $\Delta_{i_0}(M_1)$-updates, this can only be a proper M_1-step together with the corresponding RECORD updates (or a lock request directly preceding such a $\Delta_{i_1}(M_1)$-step) because in the computation with cleansed schedule each lock request of M_1 is granted and M_1 is not *Victim*ized. The values M_1 reads or writes in this step (in private or locked locations) have not been affected by a preceding step of any $M \neq M_1$—otherwise M would have locked before the non-private locations and keep the locks until it commits (since cleansed schedules are without UNDO steps), preventing M_1 from getting these locks which contradicts the fact that M_1 is the first machine to commit and thus the first one to get the locks. Therefore the values M_1 reads or writes in the step defined by $\Delta_{i_0}(M_1)$ (resp. also $\Delta_{i_1}(M_1)$) coincide with the corresponding location values in the first (resp. also second) step of M_1 following the cleansed schedule to pass from S_0 to S'_1 (case without request of *newLocks*) resp. from S_0 to S'_1 to S'_2 (otherwise). The same argument applies in the inductive step which establishes the claim. □

5 Conclusion

In this article we specified a transaction operator that turns the behaviour of a set of Abstract State Machines into a transactional one under the control of a transaction controller TACTL. In this way the locations shared by the ASMs are accessed in a well-defined matter. For this we proved that all runs of the combined asynchronous ASM are serialisable.

The relevance of the transaction operator is that it permits to concentrate on the specification of ASMs and to ignore any problems resulting from the use of shared locations. That is, specifications can be written in a way that shared locations are treated as if they were exclusively used by a single ASM. This add-on to ASMs is very valuable for many applications, as shared locations (in particular, locations in a database) are common, and random access to them is hardly ever permitted. Though in the article at hand we concentrated on transaction control based on locking, generalisations to other approaches to serialisability exploiting time-stamp oriented, optimistic or hybrid protocols are possible – e.g. see [12] for an ASM-based treatment of multi-level transaction control.

Furthermore, by shifting transaction control into the rigorous framework of Abstract State Machines we made several extensions to transaction control as known from the area of databases [5]. In the classical theory schedules are sequences containing read- and write-operations of the transactions plus the corresponding read- and write-lock and commit events, i.e., only one such operation or event is treated at a time. In our case we exploited the inherent parallelism in ASM runs, so we always considered an arbitrary update set with usually many updates at the same time. Under these circumstances we generalised the notion of schedule and serialisability using the common terminology from ASMs. In this way we stimulate also more parallelism in transactional systems.

We would like to see further detailings of the proof to a mechanically verified one, e.g. using the ASM theories developed in KIV (see [1] for an extensive list of relevant publications) and PVS [6,9,8] or the (Event- [3]) B [2] theorem prover for an (Event-) B transformation of $TA(\mathcal{M}, \textsc{TaCtl})$ (as suggested in [7]).

References

1. The KIV system. http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/.
2. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
3. J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
4. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
5. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2006.
6. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322. Springer-Verlag, 2000.
7. U. Glässer, S. Hallerstede, M. Leuschel, and E. Riccobene. Integration of Tools for Rigorous Software Construction and Analysis (Dagstuhl Seminar 13372). *Dagstuhl Reports*, 3(9):74–105, 2014.
8. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzson, editor, *Int. Conf. on Compiler Construction, Proc. Poster Session of CC'96*, Linköping, Sweden, 1996. IDA Technical Report LiTH-IDA-R-96-12.
9. G. Goos, H. von Henke, and H. Langmaack. Project Verifix. http://www.info.uni-karlsruhe.de/projects.php/id=28&lang=en.
10. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

11. J. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.
12. M. Kirchberg, K.-D. Schewe, and J. Zhao. Using Abstract State Machines for the design of multi-level transaction schedulers. In J.-R. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis – Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday*, volume 5115 of *LNCS Festschrift*, pages 65–77. Springer, 2009.
13. K.-D. Schewe, T. Ripke, and S. Drechsler. Hybrid concurrency control and recovery for multi-level transactions. *Acta Cybernetica*, 14(3):419–453, 2000.