Serialisable Multi-Level Transaction Control: A Specification and Verification

Egon Börger[a,*], Klaus-Dieter Schewe[b], Qing Wang[c]

[a] *Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy*
[b] *Software Competence Centre Hagenberg, A-4232 Hagenberg, Austria*
[c] *Research School of Computer Science, Australian National University, Australia*

Abstract

We define a programming language independent controller TaCtl for multi-level transactions and an operator *TA*, which when applied to concurrent programs with multi-level shared locations containing hierarchically structured complex values, turns their behavior with respect to some abstract termination criterion into a transactional behavior. We prove the correctness property that concurrent runs under the transaction controller are serialisable, assuming an *Inverse Operation Postulate* to guarantee recoverability. For its applicability to a wide range of programs we specify the transaction controller TaCtl and the operator *TA* in terms of Abstract State Machines (ASMs). This allows us to model concurrent updates at different levels of nested locations in a precise yet simple manner, namely in terms of partial ASM updates. It also provides the possibility to use the controller TaCtl and the operator *TA* as a plug-in when specifying concurrent system components in terms of sequential ASMs.[1]

Keywords: Abstract State Machines, Multi-Level Transactions, partial updates, serializability

1. Introduction

This paper is about the use of generalized multi-level transactions as a means to control the consistency of concurrent access of programs to shared locations,

[*] Corresponding author

Email addresses: boerger@di.unipi.it (Egon Börger), klaus-dieter.schewe@scch.at (Klaus-Dieter Schewe), qing.wang@anu.edu.au (Qing Wang)

[1] The research reported in this paper results from the project *Behavioural Theory and Logics for Distributed Adaptive Systems* supported by the **Austrian Science Fund (FWF): [P26452-N15]**. The research reported in this paper has further been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.
The first author, Humboldt research prize awardee in 2007/08, gratefully acknowledges partial support by a renewed research grant from the Alexander von Humboldt Foundation in 2014.

which may contain hierarchically structured complex values, and to avoid that values stored at these locations are changed almost randomly. According to Beeri, Bernstein and Goodman [6] most real systems with shared data have multiple levels, where each level has its own view of the data and its own set of operations, such that operations on one level may be conflict-free, while they require conflicting lower-level operations.

A multi-level *transaction controller* interacts with concurrently running programs (i.e., sequential components of an asynchronous system) to control whether access to a possibly structured shared location can be granted or not, thus ensuring a certain form of consistency for these locations. This includes in particular the resolution of low-level conflicts by higher-level updates as provided by multi-level transactions [6, 34, 35] in distributed databases [7, 30]. A commonly accepted consistency criterion is that the joint behavior of all transactions (i.e., programs running under transactional control) with respect to the shared locations is equivalent to a serial execution of those programs. Serialisability guarantees that each transaction can be specified independently from the transaction controller, as if it had exclusive access to the shared locations.

It is expensive and cumbersome to specify transactional behavior and prove its correctness again and again for components of the great number of concurrent systems. Our goal is to define once and for all an abstract (i.e. programming language independent) transaction controller TaCtl which can simply be "plugged in" to turn the behavior of concurrent programs (i.e., components M of any given asynchronous system $\mathcal{M}$) into a transactional one. This involves to also define an operator $\mathit{TA}(\bullet, \textsc{TaCtl})$ that transforms a program M into a new one $\mathit{TA}(M, \textsc{TaCtl})$, by means of which the programs M are forced to listen to the controller TaCtl when trying to access shared locations. To guarantee recoverability where needed we use an *Inverse Operation Postulate* (Sect.4.4) for component machines M; its satisfaction is a usage condition for submitting M to the transaction controller.

For the sake of generality we define the operator and the controller in terms of Abstract State Machines (ASMs), which can be read and understood as pseudo-code so that TaCtl and the operator *TA* can be applied to code written in any programming language (to be precise: whose programs come with a notion of single step, the level where our controller imposes shared memory access constraints to guarantee transactional code behavior). The precise semantics underlying the pseudo-code interpretation of ASMs (for which we refer the reader to [12]) allows us to mathematically prove the correctness of our controller and operator.

Furthermore, we generalize the strictly hierarchical view of multiple levels by using the partial update concept for ASMs developed in [23] and further investigated in [26] and [33]. This abstraction by partial updates simplifies the transaction model, as it allows us to model databases with complex values and to provide an easy-to-explain, yet still precise model of *multi-level* transactions, where dependencies of updates of complex database values are dealt with in terms of compatibility of appropriate value changing operators (see also [27]). In fact, technically speaking the model we define here is an ASM refinement (in the sense of [8]) of some of the components of the model published in [10], namely by a) generalizing the flat transaction model to multi-level transactions which increase the concurrency in transactions and b) including an ABORT mechanism. Accordingly, the serializability proof is a refinement of the proof in [10], as the refined model is a conservative extension of the model for flat transactions.[2]

We concentrate on transaction controllers that employ locking strategies such as the common two-phase locking protocol (2PL) [31]. That is, each transaction first has to acquire a (read- or write- or more generally operator-) lock for a shared, possibly nested location, whereby the access to the location to perform the requested operations 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. [14, 21, 27, 32] and the extensive literature there covering classical transaction control for flat transactions, timestamp-based, optimistic and hybrid transaction control protocols, as well as other non-flat transaction models such as sagas. To model each of these approaches would fill a book; our more modest goal here is to concentrate on one typical approach to illustrate with rigour and in full detail a method by which such transaction handling techniques can be specified and proved to be correct. For the same reason we do not consider fairness issues, though they are important for concurrent runs.

In Section 2 we first give a more detailed description of the key ideas of multi-level transactions and their relationship to partial updates. We define TACTL and the operator *TA* in Section 3 and the TACTL components in Section 4. In Section 5 we prove the correctness of these definitions.

We assume the reader to have some basic knowledge of ASMs, covering the definitions—provided 20 years ago in [**?**] and appearing in textbook form in [12, Sect.2.2/4]—for what are ASMs (i.e. their transition rules) and how their execution in given environments performs state changes by applying sets of updates to locations. Nevertheless at places where some technical details about ASMs need to be refered to we briefly describe their notation and their meaning so that the paper can be understood also by a more general audience of readers who view ASMs as a semantically well-founded form of pseudo-code that performs computations over arbitrary structures.

2. Multi-Level Transactions and Partial Updates

While standard flat transaction models start from a view of operation sequences at one level, where each operation reads or writes a shared location—in less abstract terms these are usually records or pages in secondary storage—the multi-level transaction model [6, 34, 35] relaxes this view in various ways. The key idea is that there are multiple levels, each with its own view of the data and its own set of operations.

[2] For a detailed illustration of combined model and proof refinement we refer the reader to the Java compiler correctness verification in [5].

The operations on a higher level may be compatible with one another, whereas operations on a lower level implementing them are not. As a motivating example pages in secondary storage and records stored in these pages can be considered. Updating two different records in the same page should be compatible, but not simultaneous writing of the whole page. When updating a particular record, this record should be locked for writing; as writing the record requires also writing the page, the page should also be locked. However, the page lock could immediately be released after writing, as it is sufficient to block updates to the record until commit. So another transaction could get access to a different record on the same page with a long lasting lock on the record and another *temporary* lock on the page.

A second key idea of the multi-level transaction model stressed in [32, 34, 35] is that some high-level operations may even be compatible when applied to the same shared location. Standard examples are addition, subtraction or insertion of values into a set. For instance, if a field in a record is to be updated by adding 3 to the stored value, then another operation subtracting 2 could be executed as well without causing inconsistencies. Consequently, the strictness of a lock can be relaxed, as a plus-lock can co-exist with another plus-lock, but must prevent an arbitrary update or a times-lock (for multiplication).

We will demonstrate in the following sections that these key ideas of the multi-level transaction model can be easily and precisely captured by refinement of the ASM-based transaction handler in [10]. Since to execute a step a component ASM M computes a set of updates (on which the transaction controller TACTL can speculate for lock handling etc.), it suffices to incorporate partial updates (as handled in [33]) into the model developed in [10]. For the first idea of the multi-level transaction model we exploit the *subsumption* relation between locations defined in [33]: a location l subsumes a location l' iff in all states S the value of l, i.e. $eval(l, S)$, uniquely determines the value of l', i.e. $eval(l', S)$. For instance, a value of a page uniquely determines the values of the records in it, but also a tree value determines the values of subtrees and leaves. The notion of subsumption offers a simple realization of the concept of temporary locks: temporary locks are needed on all subsuming locations.

The second idea of compatible operations can be captured by introducing particular operation-dependent locks, which fine-tune the exclusive write locks. Some of these operation-locks may be compatible with each other, such that different transactions may execute simultaneously operations on the same location. Naturally, this is only possible with partial updates defined by an operator op and an argument v. The new value stored at location l is obtained by evaluating $op(eval(l, S), v)$. If operators are compatible in the sense that the final result is independent from the order in which the operators are applied, then several such partial updates can be executed at the same time.

Thus, the refinement of the concurrent ASM in [10] for handling flat transactions affects several aspects:

- Each component machine $TA(M, \text{TACTL})$ resulting from the transaction operator will have to ask for more specific operation-locks and to execute partial updates together with other machines.

- Each component machine $TA(M, \textsc{TaCtl})$ will also have to release temporary locks at the end of each step.
- In case already the partial updates of M itself are incompatible, i.e. are such that they cannot be merged into a single genuine update, the machine $TA(M, \textsc{TaCtl})$ should not fire at all; instead, it must be completely ABORTed, i.e., all its steps will have to be undone immediately.
- The LOCKHANDLER component requires a more sophisticated condition for granting locks, which takes subsumption into account.
- The RECOVERY component will have to be extended to capture UNDOing also partial updates, for which inverse operations are required.
- The DEADLOCKHANDLER and COMMIT components remain unaffected.

While these refinements with partial updates to capture multi-level transactions require only a few changes—which also extend easily to the serializability proof—they also highlight some not so obvious deficiencies in the model of multi-level transactions itself. In [6] it is claimed that each higher-level operation is implemented by lower-level ones. For instance, an update of a record requires reading and writing a page. This is also true for object-oriented or complex value systems. For instance, in [32] it is anticipated that there could be levels for objects, records and pages, such that an operation on an object would require several update operations on records storing parts of the object. However, in the light of partial updates it is the object that subsumes the record. This implies that the definition of level-specific conflict relations [34, 32] with the condition that a high-level conflict must be reflected in a low-level one, but not vice versa, is too specific. It is true for fields, records and pages, but cannot be applied any more, when the higher-level locations subsume the lower-level ones. On the other hand, using subsumption for the definition of levels does not work either, as objects and pages are conceptually different and should not be considered as residing on the same level. To this end the use of subsumption between locations makes the idea behind multi-level transactions much clearer and formally consistent. In particular, the notion of level itself becomes irrelevant in this setting, so in a sense the transaction model formalised in this article can be seen as a moderate generalisation of the multi-level transaction model.

A second strengthening and generalisation of the concept of multi-level transactions realized in our model comes from the observation that in order to undo a partial update inverse operations are not just nice to have, but must exist, because otherwise recoverability cannot be guaranteed. This also shows that a transaction model cannot be treated in isolation without taking recovery into account at the same time.

3. The Transaction Controller and Operator

As explained above, a transaction controller performs the lock handling, the deadlock detection and handling, the recovery mechanism (for partial recovery) d the commit or abortion of single machines—we use Astract State Machines to describe programs. Thus we define TaCtl as consisting of five components specified in Sect. 4. We use SmallCaps for rules and *italics* for functions, sets, predicates, relations.

TaCtl =
  LockHandler
  DeadlockHandler
  Recovery
  Commit
  Abort

3.1. The Transaction Operator TA(•, TaCtl*)*

The operator $\mathit{TA}(\bullet, \textsc{TaCtl})$ transforms the component machines M of any concurrent system (in particular an asynchronous, concurrent ASM [11]) $\mathcal{M} = (M_i)_{i \in I}$ into components of a concurrent system $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$, where each component $\mathit{TA}(M_i, \textsc{TaCtl})$ runs as transaction under the control of TaCtl. Thus $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is defined as follows:[3]

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

It remains to expalin the definition of $\mathit{TA}(M, \textsc{TaCtl})$ below. TaCtl keeps a dynamic set *TransAct* of those machines M, whose runs it currently has to supervise. This is to guarantee that M operates in a transactional manner, until it has *Terminated* its transactional behavior (so that it can Commit it).[4] To turn the behavior of a machine M into a transactional one, first of all M has to register itself with the controller TaCtl, i.e., to be inserted into the set of *TransAct*ions currently to be handled. UndOing some steps M made in the given transactional run as part of a recovery, a last-in first-out queue $\mathit{history}(M)$ is needed, which for each step of M keeps track of the newly requested locks and of the recovery updates needed to Restore the values of the locations M changed in this step. When M enters the set *TransAct*, the $\mathit{history}(M)$ has to be initialized (to the empty queue).

The crucial transactional feature is that each non-private (i.e. shared or monitored or output)[5] location l a machine M needs to read or write for performing a step has to be $\mathit{LockedBy}(M)$ 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 LockHandler $\mathit{GrantedLocksTo}(M)$, M *canGo* to try to perform its next step: if it cannot fire (due to an inconsistency of

[3] For notational economy we use the same letters *TA* once to denote an operator applied to a set of component machines and TaCtl, once to denote an operator applied to single component machines and TaCtl. From the context it is always clear which *TA* we are talking about.

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

[5] See [12, Ch.2.2.3] for the classification of locations and functions.

the set *aggregatedUpdSet*[6] of updates computed by M from the assignment and the partial update instructions of M, see below) it calls the Abort component. If *CanFire*(M) holds, we require Fire(M) to perform the M-step together with one step of all *Partner*(M)-machines, i.e. of machines N that *CanFire*(N) simultaneously with M and share some locations to be updated with M (possibly via some compatible update operations on those locations, see below).[7] This means to Aggregate the (below called genuine) updates M yields in its *currState*(M) together with the partial updates of M together with the genuine updates and partial updates of all *Partner*(M)-machines. In addition a RecoveryRecord component has to Record for each of these machines N in its *history* the obtained *newLocks* together with the *recoveryUpd*ates needed should it become necessary to Undo the updates contributed by N to this Aggregate-step. Then M continues its transactional behavior until it is *Terminated*. In case the LockHandler *RefusedLocksTo*(M), namely because another machine N in *TransAct* has some of these locks, M has to *Wait* for N; in fact it continues its transactional behavior by calling again the LockHandler for the needed *newLocks*—until the needed locked locations are unlocked, when N's transactional behavior is Commited, whereafter a new request for these locks *GrantedLocksTo*(M) may become true.[8]

As a consequence deadlocks may occur, namely when a cycle occurs in the transitive closure $Wait^*$ of the *Wait* relation. To resolve such deadlocks the DeadlockHandler component of TaCtl chooses some machines as *Victim*s for a recovery.[9] After a victimized machine M is *Recovered* by the Recovery component of TaCtl it can exit its *waitForRecovery* mode and continue its transactional behavior.

This explains the following definition of *TA*(M, 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 [12, Ch.2.2.6]).[10] The macros which appear in Fig. 1 are defined in the rest of this section.

[6] We borrow the name from CoreASM [16].

[7] This view of concurrency is an instance of the general definition of concurrent ASMs provided in [11].

[8] A refinement (in fact a desirable optimization) consists in replacing such a waiting cycle by suspending M until the needed locks are released. Such a refinement can be obtained in various ways, a simple one consisting in letting M simply stay in *waitForLocks* until the *newLocks CanBeGranted* and refining LockHandler to only choose pairs $(M, L) \in$ *LockRequest* where it can GrantRequestedLocks(M, L) and doing nothing otherwise (i.e. defining RefuseRequestedLocks$(M, L) =$ **skip**). See Sect. 4.

[9] To simplify the serializability proof in Sect.4 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*.

[10] The components for the recovery feature are highlighted in the flowchart by a different colouring.

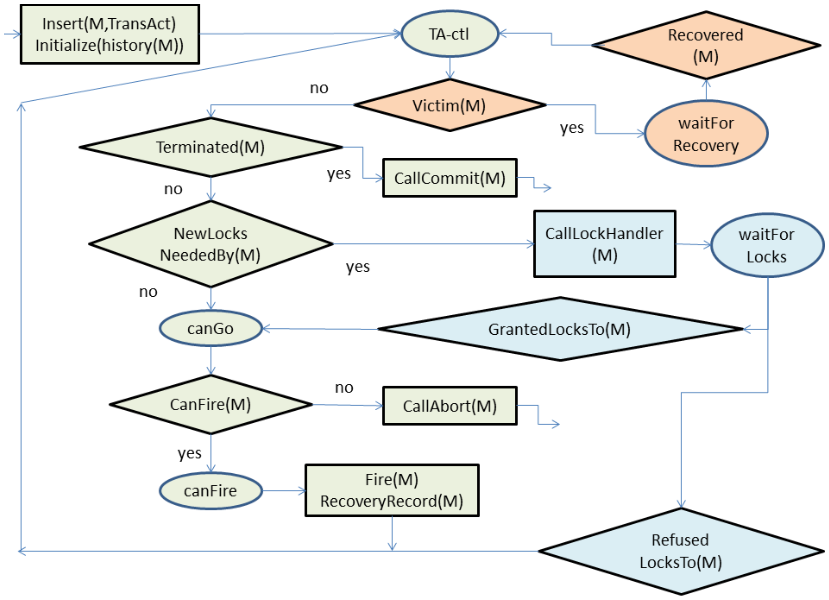

Figure 1: TA(M,TaCtl)

3.2. The TA(M, TaCtl) Macros

The predicate *NewLocksNeededBy*(M) holds, if in the current state of M at least one of two cases happens: either M reads some shared or monitored location, which is not yet *LockedBy*(M) for reading or writing, or M writes some shared or output location which is not yet *LockedBy*(M) for the requested write operation. We compute the set of such needed, but not yet *Locked* locations by a function *newLocks* (whose arguments we omit for layout reasons in Fig.1).

$$\begin{aligned}&\mathit{NewLocksNeededBy}(M) =\\&\quad \mathit{newLocks}(M, \mathit{currState}(M)) \neq \emptyset\end{aligned}$$

Whether a lock for a location can be granted to a machine depends on the kind of read or write operation the machine wants to perform on the location.

Updates and partial updates. In basic ASMs a write operation is denoted by *assignment instructions* of form

$$s := t$$

resulting for $s = f(t_1, \ldots, t_n)$ in any given state S in an update of the location $l = (f, (\mathit{eval}(t_1, S), \ldots, \mathit{eval}(t_n, S))$ by the value $\mathit{eval}(t, S)$ [12, pg.29]. Here $\mathit{eval}(t, S)$ denotes the evaluation of t in state S (under a given interpretation I of free variables). We call such updates (l, val) *genuine* (in [33] they are called exclusive) to distinguish them from partial updates. The reader who is not knowledgeable about ASMs may interpret locations $(f, args)$ correctly as array variables with variable name f and index *args*.

Analogously, we denote write operations that involve partial updates via an operation *op* by *update instructions* of form

$$s :=_{op} t$$

which require an overall update of the location $(f, (eval(t_1, S), \ldots, eval(t_n, S))$ by a value to which $s :=_{op} t$ contributes by the value $op(eval(s, S), eval(t, S))$. A typical example of such operations is parallel counting (used already in [9]) where say seven occurences of a partial update instruction

$$x :=_{parCount} x + 1$$

in a state S are aggregated into a genuine update $(x, eval(x + 7, S))$. Other examples are tree manipulation by simultaneous updates of independent subtrees or more generally term rewriting by simultaneous independent subterm updates, etc., see [32, 33]. AGGREGATE (which is implemented as a component in CoreASM [16]) specifies how to compute and perform the desired overall update effect, i.e. the genuine update set yielded by the set of all genuine and the multiset of all partial updates involving any location l and all other higher or lower level location updates the new value of l may depend upon due to an update to be performed at that level by some machine in the considered step.

Therefore, a location can be $LockedBy(M)$ for reading $(Locked(l, M, Read))$ or for writing $(Locked(l, M, Write))$ via a genuine update or for a partial update using operation *op* $(Locked(l, M, op))$. We also use $Locked(l, M, temp)$ for a temporary lock of a location l. Same as a genuine write-lock such a temporary lock blocks location l for exclusive use by M. However, such temporary locks will be immediately released at the end of a single step of M. As explained in Section 2 the purpose of such temporary locks is to ensure that an implied write operation on a subsuming location (i.e., a partial update) can be executed, but the lock is not required any more after completion of the step, as other non-conflicting partial updates should not be prohibited.

Even if $Locked(l, M, op)$ temporarily (case $op = temp$) or for a partial update operation (case $op \neq Read, Write$) machine M still needs a lock to be allowed to Read l because for a partial update location a different machine could acquire another compatible operation lock on l that is not controllable by M alone. For this reason partial update operations are defined below to be incompatible with Read and genuine Write.

$$\begin{array}{l} newLocks(M, currState(M))^{11} = \\ \quad \{(l, Read) \mid l \in R\text{-}Loc \textbf{ and} \\ \quad\quad \textbf{not } Locked(l, M, Read) \textbf{ and not } Locked(l, M, Write)\} \end{array}$$

[11] By the second argument $currState(M)$ of *newLocks* 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 $ReadLoc(M)$ and $WriteLoc(M)$; see Sect. 5 for the detailed definition.

```
    $\cup \{(l, o) \mid l \in W\text{-}Loc$ **and** $o \in \{Write\} \cup Opn$ **and not** $Locked(l, M, o)\}$
    $\cup \{(l, temp) \mid \exists l' \in W\text{-}Loc\ l \neq l'$ **and** $l$ subsumes $l'\}$
  **where**
  $R\text{-}Loc = ReadLoc(M, currState(M)) \cap (SharedLoc(M) \cup MonitoredLoc(M))$
  $W\text{-}Loc = WriteLoc(M, currState(M)) \cap (SharedLoc(M) \cup OutputLoc(M))$
$LockedBy(M) =$
  $\{l \mid Locked(l, M, Read)$ **or** $Locked(l, M, Write)$ **or** $Locked(l, M, temp)$ **or**
    $Locked(l, M, op)$ **forsome** $op \in Opn\}$
```

To CallLockHandler for the *newLocks* requested by M in its $currState(M)$ means to Insert(M) into the LockHandler's set of to be handled *LockRequest*s. Similarly we let CallCommit(M) resp. CallAbort(M) stand for insertion of *M* into a set *CommitRequest* resp. *AbortRequest* of the Commit resp. Abort component.

```
CallLockHandler$(M) =$ Insert$(M, LockRequest)$
CallCommit$(M) =$ Insert$(M, CommitRequest)$
CallAbort$(M) =$ Insert$(M, AbortRequest)$
```

Once a machine *canGo* because it has acquired all needed locks for its next proper step, it must be checked whether the $aggregatedUpdSet(M, currState(M))$ it yields in its current state is consistent so that $CanFire(M)$: if this is not the case, M is Abort ed whereby it interrupts its transactional behavior.

$$CanFire(M) = Consistent(aggregatedUpdSet(M, currState(M))).$$

Here $aggregatedUpdSet(M, S)$ is defined as the set of updates M yields[12] in state S, once the resulting genuine updates have been computed for all partial updates to be performed by M in S.[13] If this update set is consistent, to Fire(M) Aggregate performs not only the (genuine and partial) updates of M, but also those of any other $Partner(M)$-machine N which shares some to-be-updated location with M and $CanFire(N)$ simultaneously with M.

```
Fire$(M) =$
  **forall** $N \in Partner(M)$ **do** $N$
  ReleaseTempLocks$(M)$
**where**
  $Partner(M) = \{N \mid ShareUpdLocs(M, N)$ **and** $mode(N) = canFire\}$
  ReleaseTempLocks$(M) =$
    **forall** $l$ $Locked(l, M, temp) := false$
```

The constraints defined in the next section for $GrantedLocksTo(M)$ and the consistency condition for *aggregatedUpdSet*s guarantee that Fire(M) computes and performs a consistent update set.

[12] See the definition in [12, Table 2.2 pg.74].

[13] In CoreASM [16] this computation is done by corresponding plug-ins.

Remark on notation. As usual with programming languages, for ASMs we consider (names for) functions, rules, locations, etc., as elements of the universe for which sets (like *ReadLoc*, *WriteLocs*) and relations (like subsumption) can be mathematically defined and used in rules. In accordance with usual linguistic reflection notation we also quantify over such elements, e.g. in **forall** $N \in$ *SetOfAsm* **do** N, meaning that **do** N stands for an execution of (a step of) the ASM denoted by the logical variable N.

The RecoveryRecord(M) component has to Record for each *Partner*(M)-machine its *recoveryUpd*ates (defined below where we need the details for the Recovery machine) and the obtained *newLocks*.

```
RecoveryRecord($M$) = forall $N \in Partner(M)$
  Record($recoveryUpd(N, currState(N))$,
    $newLocks(N, currState(N)), N$)
Record($recUpdSet, lockSet, N$) =
  Append($(recUpdSet, lockSet), history(N)$)
```

Remark on nondeterminism. The ASM framework provides two ways to deal with nondeterminism. ‘True’ nondeterminism can be expressed using the **choose** construct to define machines of form

$M =$**choose** x **with** α **in** $r(x)$

where r has to be an ASM rule. Nondeterminism can also be modeled ‘from outside’ by using choice functions, say *select*, in machines of form

$N = r(select(\alpha))$

where in the view of the transition rules everything is deterministic once a definition of the choice function is given. Using one or the other form of nondeterminism influences the underlying logic for ASMs (see [12, Ch.8.1]).

The locks acquired for a machine M as above depend on the chosen value a for x so that when M performs its next step it must have the same value a for x to execute $r(x)$. To ‘synchronize’ this choice of a for x for lock acquisition and rule execution we assume here nondeterminism in component machines $M \in$ *TransAct* to be expressed by choice functions.

4. The Transaction Controller Components

4.1. The Commit *Component*

A CallCommit(M) by machine M enables the Commit component, which handles *CommitRequest*s one at a time. For this we use the **choose** operator, so we can 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 $LockedBy(M)$.[14] 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.[15]

```
Commit =
  if $CommitRequest \neq \emptyset$ then
    choose $M \in CommitRequest$ Commit($M$)
  where
    Commit($M$) =
      forall $l \in LockedBy(M)$  Unlock($l, M$)
      Delete($M, CommitRequest$)
      Delete($M, TransAct$)
    Unlock($l, M$) = forall $o \in \{Read, Write\} \cup Opn$
      $Locked(l, M, o) := false$
```

The locations $(Locked, (l, M, o))$ are shared by the Commit, LockHandler and Recovery components, but these components never have the same M simultaneously in their request or *Victim* set, respectively: When M has performed a CallCommit(M), it has *Terminated* its transactional computation and does not participate any more in any *LockRequest* or *Victim*ization. Furthermore, by definition no M can at the same time issue a *LockRequest* (possibly triggering the LockHandler component) and be a *Victim* (possibly triggering the Recovery component).

4.2. The LockHandler *Component*

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

The strategy we adopted in [10] for lock handling with only genuine updates was to refuse all locks for locations requested by M, if at least one of the following two cases occurs:

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

In other words, read operations of different machines are compatible and upgrades from read to write locks are possible. In the presence of partial updates, which have to be simultaneously performed by one or more transactional

[14] Unlock is called only in states where M has no *temp*orary lock.

[15] We omit clearing the $history(M)$ queue since it is initialized when M is inserted into $TransAct$(TaCtl).

machines this compatibility relation has to be extended to partial operations, but guaranteeing the consistency of the result of the AGGREGATE mechanism which binds together shared updates to a same location. We adopt the following constraints defined in [33]:

- A genuine Write is incompatible with a Read or genuine Write or any partial operation $op \in Opn$ of any other machine.
- A Read is incompatible with any Write (whether genuine or involving a partial operation $op \in Opn$).
- Two partial operations $op, op' \in Opn$ are incompatible on a location l if in some state applying to update l first op then op' yields a different result from first applying op' then op.

However, to guarantee the serializability of transactions in the presence of partial updates of complex data structures consistency is needed also in case one update concerns a substructure of another update. Therefore we stipulate that a lock request for l *CannotBeGranted* to a machine M as long as a location l' which subsumes l is *Locked* by another machine N. The subsumption definition is taken from [33, Def.2.1]:

l' subsumes l = **forall** S $eval(l', S)$ uniquely determines $eval(l, S)$

To REFUSEREQUESTEDLOCKS it suffices to set the communication interface *RefusedLocksTo*(M) of *TA*(M, TACTL); this makes M *Wait* for each location l and operation o for which the lock *CannotBeGranted* to M.

```
LOCKHANDLER =
  if $LockRequest \neq \emptyset$ then
    choose $M \in LockRequest$ HANDLELOCKREQUEST$(M)$
  where
    HANDLELOCKREQUEST$(M)$ =
      let $L = newLocks(M, currState(M))$
        if $CannotBeGranted(M, L)$
          then REFUSEREQUESTEDLOCKS$(M, L)$
          else GRANTREQUESTEDLOCKS$(M, L)$
        DELETE$((M, L), LockRequest)$
    $CannotBeGranted(M, L)$ =
      forsome $(l, o) \in L$ $CannotBeGranted(l, M, o)$
    $CannotBeGranted(l, M, o)$ =
      forsome $N \in TransAct \setminus \{M\}$  Blocks $(N, l, o)$
     Blocks $(N, l, o)$ = forsome $o'$
      $Locked(l, N, o')$ and not $Compatible(o, o', l)$
        or forsome $l'$ $Locked(l', N, o')$ and $l'$ subsumes $l$
    REFUSEREQUESTEDLOCKS$(M, L)$ = $(RefusedocksTo(M) := true)$
    GRANTREQUESTEDLOCKS$(M, L)$ =
      forall $(l, o) \in L$  $Locked(l, M, o) := true$
      $GrantedLocksTo(M) := true$
```

4.3. The DeadlockHandler *Component*

A *Deadlock* originates if two machines are in a *Wait* cycle, i.e., they wait for each other. In other words, a deadlock occurs, when for some (not yet *Victim*ized) machine M the pair (M, M) is in the transitive (not reflexive) closure $Wait^*$ of *Wait*. In this case the DeadlockHandler selects for recovery a (typically minimal) subset of *Deadlocked* transactions *toResolve*—they are *Victim*ized to *waitForRecovery*, in which mode (control state) they are backtracked until they become *Recovered*. 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 using the **choose** operator.

```
DeadlockHandler =
  if $Deadlocked \cap \overline{Victim} \neq \emptyset$ then  // there is a Wait cycle
    choose $toResolve \subseteq Deadlocked \cap \overline{Victim}$
      forall $M \in toResolve$ $Victim(M) := true$
  where
    $Deadlocked = \{M \mid (M, M) \in Wait^*\}$
    $Wait^* =$ TransitiveClosure($Wait$)
    $Wait(M, N) =$ forsome $(l, o)$ $Wait(M, (l, o), N)$
    $Wait(M, (l, o), N) =$
      $(l, o) \in newLocks(M, currState(M))$ and $N \in TransAct \setminus \{M\}$
        and  Blocks $(N, l, o)$
```

4.4. The Recovery *Component*

Also for the Recovery component we use the **choose** operator to leave the order in which the *Victim*s are chosen for recovery refinable by different instantiations of TaCtl. In order to be *Recovered* a machine M is backtracked by Undo(M) steps until M is not *Deadlocked* any more, in which case it is deleted from the set of *Victims*, so that by 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 $M \notin Deadlocked$ then $Victim(M) := false$
      else Undo($M$)
  $Recovered =$
    $\{M \mid ctl\text{-}state(M) = waitForRecovery$ and $M \notin Victim\}$
```

To define an Undo(M) step we have to provide the details of the function *recoveryUpd* used above in RecoveryRecord. This function collects for any given machine M and state S first of all the *genuineRecoveryUpd*ates by which one can RESTORE the overwritten values in *GenuineWriteLoc*ations (i.e. locations to which M in S writes via an assignment instruction); in [10] where we considered only genuine updates this function was called *overWrittenVal*.

In addition, for each to be AGGREGATEd update instruction $s :=_{op} t \in UpdInstr(M, S)$ *recoveryUpd* collects the information to compute the 'inverse' update for M, information that is needed when the controller has to UNDO at the concerned location the effect of that partial update by M (but not of simultaneous partial updates concerning the same location by other machines). This information consists in an operation op' with the appropriate value v' for its second argument, whereas the first argument is provided only when the UNDO takes place. For the approach to ASMs with partial updates defined in [33] and adopted here one has to postulate that such operations and values (op', v') which are *inverse* to partial update operations (op, v) (where $v = eval(t, currState(M))$) are defined and satisfy the following constraint for partial update instructions:

Inverse Operation Postulate
forall $s :=_{op} t \in UpdInstr(M)$ **forall** (op, v) **thereIs** (op', v') **with**
forall w $op'(op(w, v), v') = w$

This postulate can be justified by the requirement that any transaction should be *recoverable* [32]. If recoverability cannot be guaranteed, a transaction controller must (in principle) be able to undo updates that were issued long ago, which would be completely infeasible for any real system where once a transaction has committed, it can leave the system, and none of its updates can be undone any more. As partial updates operations (op_1, v_1) and (op_2, v_2) from two different machines M and N could be executed simultaneously, for each of these operations it must be forseen that it may be undone, even if the issueing transaction for the other operation has already committed—this situation has become possible. As the original value at location l at the time of the partial update by M using (op, v) is no longer available—anyway, it may have been updated many times by other compatible partial updates—M must be able to undo its part of the update independently from all other updates including UNDOne ones, i.e. to say after UNDOing (op, v), the resulting values at location l must be just the one that would have resulted, if only all other (not yet UNDOne) partial updates had been executed. This is guaranteed by the inverse operation postulate.

In the original work on multi-level transactions including [6] recovery is not handled at all, which leads to misleading conclusions that commutativity of high-level operations—those that can be defined by partial updates—is sufficient for obtaining increased transaction throughput by means of additional permitted schedules. However, commutativity (better called *operator compatibility*, see [33]) has to be complemented by the inverse operation postulate to ensure recoverability. Inverse operators are claimed in the MLR recovery system [29], but no satisfactory justification was given.

There may be more than one update instruction M performs for the same location so that the corresponding inverse *partialRecoveryUpd*ates have to be Aggregated with the *genuineRecoveryUpd*ates by Restore.

```
$recoveryUpd(M, S) =$
  $(genuineRecoveryUpd(M, S), partialRecoveryUpd(M, S))$
$genuineRecoveryUpd(M, S) = \{((f, args), val) \mid$
  $(f, args) \in GenuineWriteLoc(M, S)$ and $val = eval(f(args), S)\}$
$partialRecoveryUpd(M, S) = \{(l, (op', v')) \mid$
  forsome $f(t_1, \ldots, t_n) :=_{op} t \in UpdInstr(M, S)$
    $l = (f, (eval(t_1, S), \ldots, eval(t_n, S)))$ and
      $(op', v') = inverse(op, eval(t, S))\}$

Undo$(M) =$
  let $(Upds, Locks) = youngest(history(M))$
    Restore$(Upds, M)$
    Release$(Locks, M)$
    Delete$((Upds, Locks), history(M))$
where
  Restore$((G, P), M) =$ Aggregate$(G \cup$ // NB. P is a multiset
    $\{\mid ((f, args), op'(eval(f(args), currState(M)), v') \mid$
      $((f, args), (op', v')) \in P \mid\})$ // NB. multiset notation $\{\mid \mid\}$
  Release$(L, M) =$ forall $l \in L$ Unlock$(l, M)$
```

The inverse operation postulate cannot guarantee that the inverse operations commute with each other in general. However, it can be guaranteed that on the values, to which the inverse operations are applied in Undo steps, commutativity holds: For this let (op'_i, v'_i) be inverse for (op_i, v_i) for $i = 1, 2$, such that both operations op_i are compatible and both inverse operations op'_i have to execute simultaneously on location l. That is, if v is the actual value of l in the current state, we need to show $op'_1(op'_2(v, v'_2), v'_1) = op'_2(op'_1(v, v'_1), v'_2)$. As these are Undo operations, we can assume that (op_i, v_i) for $i = 1, 2$ have been executed on some previous value of location l. Thus, due to commutativity we must have $v = op_1(op_2(v', v_2), v_1) = op_2(op_1(v', v_1), v_2)$ for some value v'. From this we get

$$op'_1(op'_2(v, v'_2), v'_1) = op'_1(op'_2(op_2(op_1(v', v_1), v_2), v'_2), v'_1) =$$
$$op'_1(op_1(v', v_1), v'_1) = v' = op'_2(op_2(v', v_2), v'_2) =$$
$$op'_2(op'_1(op_1(op_2(v', v_2), v_1), v'_1), v'_2) = op'_2(op'_1(v, v'_1), v'_2)$$

Note that in our description of the DeadlockHandler and the (partial) Recovery we deliberately left the strategy for victim selection and Undo abstract, so fairness considerations will have 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.

4.5. The ABORT Component

The ABORT component can be succinctly defined as turbo ASM [12, Ch.4.1]:

```
ABORT = forall $M \in AbortRequest$
  iterate UNDO($M$) until $history(M) = \emptyset$
  $Delete(M, TransAct)$
```

We use the **iterate** construct only here and do this for notational convenience to avoid tedious programming of an iteration. We do not use **iterate** to form component ASMs which go into *TransAct*.

5. Correctness Theorem

In this section we show the desired correctness property: 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)[16], each run of $TA(\mathcal{M}, \text{TACTL})$ 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 $TA(\mathcal{M}, \text{TACTL})$ 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.

First of all we have to make precise what a *serial* multi-agent ASM run is and what *equivalence* of $TA(\mathcal{M}, \text{TACTL})$ runs means in the general multi-agent ASM framework.

5.1. Definition of run equivalence

Let $S_0, S_1, S_2, \ldots$ be a (finite or infinite) run of $TA(\mathcal{M}, \text{TACTL})$. In general we may assume that TACTL runs forever, whereas each machine $M \in \mathcal{M}$ running as transaction will be *Terminated* or ABORTed at some time – once COMMITed M will only change values of non-shared and non-output locations[17]. To simplify the proof but without loss of generality we assume that each update concerning an ABORTed machine is eliminated from the run. For $i = 0, 1, 2, \ldots$ let Δ_i, Γ_i denote the unique set of genuine updates resp. multiset of partial updates leading to an AGGREGATEd consistent set of updates defining the transition from S_i to S_{i+1}. By definition of $TA(\mathcal{M}, \text{TACTL})$ each Δ_i, Γ_i is the union

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

[17] 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 $\mathcal{M}$.

of the corresponding sets resp. multisets[18] of the agents executing $M \in \mathcal{M}$ resp. TaCtl:

$$\Delta_i = \bigcup_{M \in \mathcal{M}} \Delta_i(M) \cup \Delta_i(\textsc{TaCtl}) \quad \Gamma_i = \overset{+}{\bigcup_{M \in \mathcal{M}}} \Gamma_i(M) \cup^+ \Gamma_i(\textsc{TaCtl}).$$

$\Delta_i(M)$ contains the genuine and $\Gamma_i(M)$ the partial updates defined by the machine $TA(M, \textsc{TaCtl})$ in state S_i[19], $\Delta_i(\textsc{TaCtl})$ resp. $\Gamma_i(\textsc{TaCtl})$ contain the genuine resp. partial updates by the transaction controller in this state. The sequence

$$\Delta_0(M), \Gamma_0(M), \Delta_1(M), \Gamma_1(M), \Delta_2(M), \Gamma_2(M) \ldots$$

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 [14, 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 Undo(M) step by the Recovery component:

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

- $\Delta_i(M) = \Gamma_i(M) = \emptyset$ (M contributes no update to S_i),
- $\Delta_i(M)$ belongs to a step of an M-computation segment where M in its *ctl_state*(M) =TA-*ctl* does CallLockHandler(M) and in its next step moves from *waitForLocks* back to control state TA$-$*ctl* because the LockHandler *RefusedLocksTo*(M).[20]

In such computation steps M makes no proper update.

(ii) Repeat choosing from the schedule of M a pair $\Delta_j(M), \Gamma_j(M)$ with later $\Delta_{j'}(M), \Gamma_{j'}(M)$ ($j < j'$) which belong to consecutive M-Recovery entry resp. exit 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*.

[18] We indicate multiset operations by an upper index +

[19] We use the shorthand notation $\Delta_i(M)$ to denote $\Delta_i(TA(M, \textsc{TaCtl}))$, analogously $\Gamma_i(M)$; in other words we speak about steps and updates of M also when they really are done by $TA(M, \textsc{TaCtl})$. 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 TaCtl. When we want to name an original update of M (not one of the updates of *ctl_state*(M) or of the Record component) we call it a proper M-update.

[20] By eliminating this CallLockHandler(M) step also the corresponding LockHandler step HandleLockRequest(M) disappears in the run.

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

(a) $\Delta_j(M), \Gamma_j(M)$ and $\Delta_{j'}(M), \Gamma_{j'}(M)$,

(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) TryToRecover(M)-updates in any $\Delta_{i+k}(\textsc{TaCtl}), \Gamma_{i+k}(\textsc{TaCtl})$ between the considered M-RecoveryEntry and M-RecoveryExit step (for i as below with $i < j < i + k < j'$),

(d) each $\Delta_{i'}(M), \Gamma_{i'}(M)$ 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 TryToRecover(M) step. Besides control state and Record updates these $\Delta_{i'}(M)$ contain genuine updates (ℓ, v) with $\ell = (f, (\mathit{eval}(t_1, S_i), \ldots, \mathit{eval}(t_n, S_i)))$ where the corresponding Undo updates are

$$(\ell, \mathit{eval}(f(t_1, \ldots, t_n), S_i)) \in \Delta_{i+k}(\textsc{TaCtl})$$

the $\Gamma_{i'}(M)$ contain partial updates

$$(f, (\mathit{eval}(t_1, S), \ldots, \mathit{eval}(t_n, S))), \mathit{op}(\mathit{eval}(f(t_1, \ldots, t_n), S)), \mathit{eval}(t), S))$$

for update instructions $f(t_1, \ldots, t_n) :=_{op} t$ of M in $S_{i'}$ whose effect is UndOne when Recovery Aggregates the M-specific partial update $(l, (\mathit{op}', v'))$ with the inverse operation (op', v') to $(\mathit{op}, \mathit{eval}(t, S_{i'}))$ on l.

(e) the HandleLockRequest(M)-updates in $\Delta_{l'}(\textsc{TaCtl})$ 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 $\Delta_{i_1}(M), \Gamma_{i_1}(M), \Delta_{i_2}(M), \Gamma_{i_1}(M), \ldots$ with $i_1 < i_2 < \ldots$ resulting from the application of the two cleansing operations as long as possible will be called the *cleansed schedule* of M (for the given run). Note that the sequence is uniquely defined because confluence results from the fact that the deletion order chosen in step (i) or step (ii) does not matter.

Before defining the equivalence of transactional ASM runs let us remark that $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ 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 is Undo together with releasing corresponding locks.

Second, the specification of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ 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 5.1. Two runs $S_0, S_1, S_2, \ldots$ and $S'_0, S'_1, S'_2, \ldots$ of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ are *equivalent* iff for each $M \in \mathcal{M}$ the cleansed schedules

$$\Delta_{i_1}(M), \Gamma_{i_1}(M), \Delta_{i_2}(M), \Gamma_{i_2}(M), \ldots$$

and

$$\Delta'_{j_1}(M), \Gamma'_{j_1}(M), \Delta'_{j_2}(M), \Gamma'_{j_2}(M), \ldots$$

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.

5.2. 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[21]. As stated above AbortEd transactions are assumed to be eliminated from the run right at the beginning. We need a definition of the read- and write-locations of 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 [12, 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.

5.2.1. Read/Write Locations of Terms and Formulae.

$\mathit{ReadLoc}(x, S) = \mathit{WriteLoc}(x, S) = \emptyset$ for variables x
$\mathit{ReadLoc}(f(t_1, \ldots, t_n), S) =$
$\quad \{(f, (\mathit{eval}(t_1, S), \ldots, \mathit{eval}(t_n, S)))\} \cup \bigcup_{1 \leq i \leq n} \mathit{ReadLoc}(t_i, S)$
$\mathit{WriteLoc}(f(t_1, \ldots, t_n), S) = \{(f, (\mathit{eval}(t_1, S), \ldots, \mathit{eval}(t_n, S)))\}$

[21] 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.

Logical variables (to be distinguished from programming variables which are treated in the ASM framework as 0-ary functions and thus stand for locations) appear in the **let**, **forall**, **choose** constructs and 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 [12, Ch.2.4.4] and I_x^d the extension (or modification) of I where x is interpreted by a domain element d.

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

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

5.2.2. Read/Write Locations of ASM Rules.

$$\begin{aligned}
&\mathit{ReadLoc}(\mathbf{skip}, S) = \mathit{WriteLoc}(\mathbf{skip}, S) = \emptyset \\
&\mathit{ReadLoc}(t_1 := t_2, S) = \mathit{ReadLoc}(t_1 :=_{op} t_2, S) = \\
&\quad \mathit{ReadLoc}(t_1, S) \cup \mathit{ReadLoc}(t_2, S) \\
&\mathit{WriteLoc}(t_1 := t_2, S) = \mathit{WriteLoc}(t_1 :=_{op} t_2, S) = \mathit{WriteLoc}(t_1, S) \\
&\mathit{ReadLoc}(\mathbf{if}\ \alpha\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2, S) = \\
&\quad \mathit{ReadLoc}(\alpha, S) \cup \begin{cases} \mathit{ReadLoc}(r_1, S) & \mathbf{if}\ \mathit{eval}(\alpha, S) = \mathit{true} \\ \mathit{ReadLoc}(r_2, S) & \mathbf{else} \end{cases} \\
&\mathit{WriteLoc}(\mathbf{if}\ \alpha\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2, S) = \begin{cases} \mathit{WriteLoc}(r_1, S) & \mathbf{if}\ \mathit{eval}(\alpha, S) = \mathit{true} \\ \mathit{WriteLoc}(r_2, S) & \mathbf{else} \end{cases} \\
&\mathit{ReadLoc}(\mathbf{let}\ x = t\ \mathbf{in}\ r, S, I) = \mathit{ReadLoc}(t, S, I) \cup \mathit{ReadLoc}(r, S, I_x^{\mathit{eval}(t,S)}) \\
&\mathit{WriteLoc}(\mathbf{let}\ x = t\ \mathbf{in}\ r, S, I) = \mathit{WriteLoc}(r, S, I_x^{\mathit{eval}(t,S)}) \ \text{// call by value} \\
&\mathit{ReadLoc}(\mathbf{forall}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = \\
&\quad \mathit{ReadLoc}(\forall x \alpha, S, I) \cup \textstyle\bigcup_{a \in \mathit{range}(x, \alpha, S, I)} \mathit{ReadLoc}(r, S, I_x^a) \\
&\qquad \mathbf{where}\ \mathit{range}(x, \alpha, S, I) = \{d \in \mathit{domain}(S) \mid \mathit{eval}(\alpha, (S, I_x^d)) = \mathit{true}\} \\
&\mathit{WriteLoc}(\mathbf{forall}\ x\ \mathbf{with}\ \alpha\ \mathbf{do}\ r, S, I) = \textstyle\bigcup_{a \in \mathit{range}(x, \alpha, S, I)} \mathit{WriteLoc}(r, S, I_x^a)
\end{aligned}$$

In the following cases the same scheme applies to read and write locations:[22]

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

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

Due to the assumption that for component machines $M \in \mathit{TransAct}$ nondeterminism is expressed by choice functions no further clause is needed to define *ReadLoc* and *WriteLocs* for machines of form **choose** x **with** α **do** r.

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

Definition 5.2. A run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ 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 $\Delta_{j_1}(M'), \Gamma_{j_1}(M'), \Delta_{j_2}(M'), \Gamma_{j_2}(M'), \ldots$ 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 have not yet committed.

Definition 5.3. A run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is *serialisable* iff it is equivalent to a serial run of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$.[23]

5.3. Serializability Proof

Theorem 5.1. *Each run of* $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ *is serialisable.*

Proof. Let $S_0, S_1, S_2, \ldots$ 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 $S_0, S'_1, S'_2, \ldots, S'_n$. As M_1 commits, this run is finite. M_1 has been $\textsc{Delete}$d from *TransAct* and none of the $\textsc{TaCtl}$ components is triggered any more: neither $\textsc{Commit}$ nor $\textsc{LockHandler}$ because *CommitRequest* resp. *LockRequest* remain empty; not $\textsc{DeadlockHandler}$ because *Deadlock* remains false since M_1 never *Wait*s for any machine; not $\textsc{Recovery}$ because *Victim* remains empty. Note that in this run the schedule for M_1 is already cleansed.

[23] Modulo the fact that ASM steps permit simultaneous updates of multiple locations, for ASMs with only genuine updates this definition of serializability is equivalent to Lamport's sequential consistency concept [28].

We now define a run $S''_0, S''_1, S''_2, \ldots$ (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 $\Delta_i(M), \Gamma_i(M)$ for the originally given run all updates made by steps of M_1 and all updates in $\textsc{TaCtl}$ steps which concern M_1 (i.e. which are related to a lock or commit request by M_1 or a victimization of M_1 or a $\textsc{TryToRecover}(M_1)$ step). Let

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

$$\Gamma''_i = \overset{+}{\bigcup_{M \in \mathcal{M} - \{M_1\}}} \Gamma_i(M) \cup^+ \{(\ell, v) \in \Gamma_i(\textsc{TaCtl}) \mid (\ell, v) \text{ does not concern } M_1\}.$$

That is, in Δ''_i, Γ''_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 $\textsc{LockHandler}$ updates involving a $\mathit{LockRequest}(M_1)$ or are $\mathit{Victim}(M_1) := \mathit{true}$ updates of the $\textsc{DeadlockHandler}$ or are updates involving a $\textsc{TryToRecover}(M_1)$ step or are done by a step involving a $\textsc{Commit}(M_1)$.

Lemma 5.2. *$S''_0, S''_1, S''_2, \ldots$ is a run of $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$.*

Lemma 5.3. *The run $S_0, S'_1, S'_2, \ldots, S'_n, S''_1, S''_2, \ldots$ of $\mathit{TA}(\mathcal{M}, \textsc{TaCtl})$ is equivalent to the original run $S_0, S_1, S_2, \ldots$.*

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

Proof.(Lemma 5.2) Omitting in Δ''_i, Γ''_i from $\Delta_i(\textsc{TaCtl}), \Gamma_i(\textsc{TaCtl})$ every update which concerns M_1 leaves updates by $\textsc{TaCtl}$ in S''_i concerning $M \neq M_1$.

It remains to show that every $\textsc{Fire}(M)$-step defined by $\Delta_i(M), \Gamma_i(M)$ is a possible $\textsc{Fire}(M)$-step via Δ''_i, Γ''_i in a $\mathit{TA}(\mathcal{M} - \{M_1\}, \textsc{TaCtl})$ run starting in S''_0. Since the considered M-schedule $\Delta_i(M), \Gamma_i(M)$ 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).

Case 1. M for its steps uses *newLocks* and some of the to-be-locked locations are also $\mathit{LockedBy}(M_1)$.

Case 1.1. Some of the *newLocks* granted to M are incompatible with some of the locks granted to M_1. Then due to cleansing the *newLocks* are requestedd by M after $\textsc{Commit}(M_1)$ so that the lock race between M and M_1 is eliminated.

Case 1.2. The *newLocks* granted to M are compatible with all locks granted to M_1. The compatibility permits to shift the considered proper M-step to after the next proper M_1-step.

Case 2. M for its step uses *newLocks* for locations but none of them is $\mathit{LockedBy}(M_1)$. Then this M-step can be shifted like in Case 1.2.

Case 3. M for its step needs no *newLocks*. Then all needed locks have been granted before and to those preceding steps the argument for Case 1 and Case 2 applies by induction. □

Proof.(Lemma 5.3) 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 $\Delta''_i(M) = \Delta_i(M)$ and $\Gamma''_i(M) = \Gamma_i(M)$ 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

$$\Delta_0(M_1), \Gamma_0(M_1), \Delta_1(M_1), \Gamma_1(M_1), \ldots$$

for the run $S_0, S_1, S_2, \ldots$ and the corresponding cleansed schedule

$$\Delta_{i_0}(M_1), \Gamma_{i_0}(M_1), \Delta_{i_1}(M_1), \Gamma_{i_1}(M_1), \ldots$$

.

We proceed by induction on the cleansed schedule steps of M_1. When M_1 makes its first step using the updates in $\Delta_{i_0}(M_1), \Gamma_{i_0}(M_1)$, this can only be a Fire(M_1)-step together with the corresponding RecoveryRecord updates (or a lock request directly preceding such a $\Delta_{i_1}(M_1), \Gamma_{i_1}(M_1)$-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 have not been affected by a preceding incompatible step of any $M \neq M_1$—otherwise M would have locked before the corresponding 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 $\Delta_{i_0}(M_1), \Gamma_{i_0}(M_1)$ (resp. also $\Delta_{i_1}(M_1), \Gamma_{i_1}(M_1)$) coincide with the corresponding location values in the first (resp. also second) step of M_1 following the cleansed schedule with the same compatible updates of partners of M 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 shared updates of M_1 are the same in both runs by definition. The same argument applies in the inductive step which establishes the claim. □

6. Conclusion

In this article we specified in terms of Abstract State Machines a multi-level transaction controller TaCtl and a multi-level transaction operator which turns the behaviour of a set of concurrent programs into a transactional one under the control of TaCtl. In this way the locations shared by the programs and possibly containing complex (hierarchically structured) values are accessed in a well-defined manner. For this we proved that all concurrent transactional runs are serialisable.

The relevance of the transaction operator is that it permits to concentrate on the specification of program behavior ignoring any problems resulting from the use of shared possibly nested locations. That is, specifications can be written in a way that shared locations, including those which contain complex values, are treated as if they were exclusively used by a single program. This is valuable for numerous applications, as shared locations (in particular in a database with complex values) are common, and random access to them is hardly ever permitted.

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 [14]. 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 in terms of the synchronous parallelism of ASMs. More importantly we included also partial updates to cope with (a generalization of) multi-level transactions. In this way we stimulate more parallelism in transactional systems. We were also able to strengthen the multi-level transaction model by adding further clarification about the dependencies between the levels—actually, we showed that a strict organisation into levels is not required, as long as subsumption dependencies are taken into consideration—and about the necessity to provide inverse operators for the partial updates that are used for higher-level operations.

Among further work we would like to be undertaken is to provide for the transaction controller and the *TA* operator specified in this paper a (proven to be correct) implementation, in particular as plug-in for the CoreASM [16, 15] or Asmeta [4] interpreter engines. This needs in particular a careful analysis of the subsumtion criterion. Note however that the update instruction set concept in CoreASM realizes the concept of partial updates as used here and defined in [33]. We would also like to see refinements or adaptations of our transaction controller model for different approaches to serialisability [21], to multi-level transaction protocols [27] and to other approaches to transaction handling, e.g. [14, 21, 27, 32]. Last but not least we would like to see further detailings of our correctness 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 [17, 20, 19] or the (Event- [3]) B [2] theorem prover for an (Event-) B transformation of $TA(\mathcal{M}, \text{TACTL})$ (as suggested in [18]).

Acknowledgement. The attempt to define a plug-in transaction controller concept one can apply to introduce sequential components into a concurrent computation under transactional constraints was partly motivated by the ATM case study presented to the Dagstuhl seminar reported in [18]. The approach of this paper has been used for the specification of the ASM modeling the ATM in [13]. We thank an anonymous referee for his insightful reading of our paper and in

particular for having pointed out a flaw in the original manuscript.

Appendix A. Partial Updates

The problem of partial updates in ASMs was first studied by Gurevich and Tillmann [24]. They observed that partial updates naturally arise in the context of synchronous parallel systems, and the problem has also manifested itself in the development of AsmL, an ASM-based specification language [22]. Although in principle partial updates can be avoided in the traditional ASM setting by explicitly formulating all intended partial updates to a structure by genuine updates, this can turn out to become rather cumbersome and in fact, AsmL required a solution that allows a programmer to freely use partial updates to modify counters, sets and maps in the main program and in submachines and in submachines of submachines, etc. without worrying how submachines will report modifications and how to integrate modifications. Therefore, Gurevich and Tillmann studied the problem of partial updates over the data types *counter*, *set* and *map* [24]. To develop a systematical approach for partial updates, they proposed an algebraic framework in which a *particle* was defined as an unary modification operation over a data type, and a parallel composition of particles as an abstraction of order-independent sequential composition. In doing so, they defined a partial update as a pair (l, p) where l is a location as in the standard ASMs, but p is a particle which is different from a value v in a genuine update (l, v) of the traditional ASMs.

Nonetheless, Gurevich and Tillmann later realised that the previous framework was too limited, for example, it failed to address partial updates over the data types *sequence* or *labeled ordered trees* as exemplified in [25]. This limitation led to the formalisation of *applicative algebras* as a general solution to partial updates [26]. It was shown that the problem of partial updates over *sequences* and *labeled ordered trees* can be solved in this more general algebraic framework, and the approach in [24] was a special kind of an applicative algebra.

Definition A.1. An *applicative algebra* consists of: (i) elements of a data type τ, which include a *trivial element* $\perp$ and at least one additional element, (ii) a monoid of particles over τ which include a *null particle* λ and the identity operation *id*, and (iii) a parallel composition Ω that, given an arbitrary finite multiset of particles, produces a particle. Each applicative algebra also needs to satisfy the following two conditions:

(1) $p(\perp) = \perp$ for each particle p, and $\lambda(x) = \perp$ for every element x.

(2) $\Omega(\{\!\!\{p\}\!\!\}) = p$, $\Omega(M + \{\!\!\{id\}\!\!\}) = \Omega M$, and $\Omega(M + \{\!\!\{\lambda\}\!\!\}) = \lambda$.

A multiset M of particles is called *consistent* iff $\Omega M \neq \lambda$.

Although applicative algebra provides a general framework for partial updates, the notion of particle is nonetheless not intuitive. Furthermore, the notion of location considered in these studies is the same as in the standard ASMs, which did not consider the subsumption relation between locations. Thus, the following definitions for partial updated were proposed in [33]:

Definition A.2. A location l_1 *subsumes* a location l_2 if, for all states S, $\mathit{eval}(l_1, S)$ uniquely determines $\mathit{eval}(l_2, S)$.

Definition A.3. A *partial update* is a triple (l, v, op) consisting of a location l, a value v, and a binary operator op. Given a state S and a single partial update (l, v, op), we obtain a new state S' by applying the partial update (l, v, op) over S and $\mathit{eval}(\ell, S') = \mathit{op}(\mathit{eval}(l, S), v)$.

In the above definition, locations may subsume one another, i.e. one location is a substructure of another location. Intuitively, for a partial update (l, v, op), the binary operator op specifies how the value v partially affects the value of l in the current state. When multiple partial updates are generated to the same location simultaneously, a multiset P_l of partial updates is obtained for the location l. The following definition of operator-compatibility ensures that partial updates to the same location are consistent in an update multiset.

Definition A.4. Let $P_l = \{\!\!\{(l, v_i, \mathit{op}_i) \mid i = 1, ..., k\}\!\!\}$ be a multiset of partial updates to the same location l. Then P_l is said to be *operator-compatible* if, for any two permutations $(\sigma_1, ..., \sigma_k)$ and $(\pi_1, ..., \pi_k)$ of $\{1, \ldots, k\}$, we have the following for all x:

$$\mathit{op}_{\sigma_k}(...\mathit{op}_{\sigma_2}(\mathit{op}_{\sigma_1}(x, v_{\sigma_1}), v_{\sigma_2}), ..., v_{\sigma_k}) = \mathit{op}_{\pi_k}(...\mathit{op}_{\pi_2}(\mathit{op}_{\pi_1}(x, v_{\pi_1}), v_{\pi_2}), ..., v_{\pi_k})$$

An update multiset P_l is *consistent* if it is operator-compatible.

Based on the above definition, the following proposition is straightforward since, for an operator-compatible update multiset to the same location, applying its partial updates in any order yields the same result.

Proposition A.1. *If an update multiset P_l is operator-compatible, then an order-independent sequential composition Θ of the partial update operations in P_l (written as ΘP_l) is equivalent to applying all the partial updates in P_l sequentially in any order. That is, $\Theta P_l(x) = \mathit{op}_{\sigma_{|P_l|}}(...\mathit{op}_{\sigma_2}(\mathit{op}_{\sigma_1}(x, v_{\sigma_1}), v_{\sigma_2}), ..., v_{\sigma_{|P_l|}})$ for any permutation $(\sigma_1, ..., \sigma_{|P_l|})$ of $\{1, \ldots, |P_l|\}$.*

Therefore, if an update multiset P_l is consistent, then all the partial updates in P_l can be aggregated into one genuine update on the same location l. A state S' can be obtained from S by applying the multiset P_l of partial updates sequentially, and we have $\mathit{eval}(l, S') = \Theta P_l(\mathit{eval}(l, S))$.

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] The Abstract State Machine Metamodel website, `http://asmeta.sourceforge.net` (2006).

[5] D. Batory, E. Börger, Modularizing theorems for software product lines: The Jbook case study, J. Universal Computer Science 14 (12) (2008) 2059–2082.

[6] C. Beeri, A. Bernstein, N.Goodman, A model for concurrency in nested transaction systems, J. ACM 36 (2) (1989) 230–269.

[7] A. Bernstein, N.Goodman, Concurrency control in distributed database systems, ACM Transactions on Computer Systems 13 (2) (1981) 121–157.

[8] E. Börger, The ASM refinement method, Formal Aspects of Computing 15 (2003) 237–257.

[9] E. Börger, I. Durdanović, D. Rosenzweig, Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters, in: U. Montanari, E. R. Olderog (eds.), Proc. PROCOMET'94 (IFIP Working Conf. on Programming Concepts, Methods and Calculi), North-Holland, 1994.

[10] E. Börger, K.-D. Schewe, Specifying transaction control to serialize concurrent program executions, in: Y. Ait-Ameur, K.-D. Schewe (eds.), Proc. ABZ 2014, vol. 8477 of LNCS, Springer, 2014.

[11] E. Börger, K.-D. Schewe, Concurrent Abstract State Machines, Acta InformaticaSubmitted. Available at the authors' websites.

[12] E. Börger, R. F. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer, 2003.

[13] E. Börger, S. Zenzaro, Modeling for change via component-based decomposition and ASM refinement, S-BPM ONE, ACM Digital Library, 2015.

[14] R. Elmasri, S. B. Navathe, Fundamentals of Database Systems, Addison Wesley, 2006.

[15] R. Farahbod, V. Gervasi, U. Glässer, CoreASM: An Extensible ASM Execution Engine, Fundamenta Informaticae XXI.

[16] R. Farahbod, et al., The CoreASM Project, `http://www.coreasm.org`.

[17] A. Gargantini, E. Riccobene, Encoding Abstract State Machines in PVS, in: Y. Gurevich, P. Kutter, M. Odersky, L. Thiele (eds.), Abstract State Machines: Theory and Applications, vol. 1912 of Lecture Notes in Computer Science, Springer-Verlag, 2000.

[18] U. Glässer, S. Hallerstede, M. Leuschel, E. Riccobene, Integration of Tools for Rigorous Software Construction and Analysis (Dagstuhl Seminar 13372), Dagstuhl Reports 3 (9) (2014) 74–105.
URL `http://drops.dagstuhl.de/opus/volltexte/2014/4358`

[19] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, W. Zimmermann, Compiler correctness and implementation verification: The verifix approach, in: P. Fritzson (ed.), Int. Conf. on Compiler Construction, Proc. Poster Session of CC'96, IDA Technical Report LiTH-IDA-R-96-12, Linköping, Sweden, 1996.

[20] G. Goos, H. von Henke, H. Langmaack, Project Verifix, `http://www.info.uni-karlsruhe.de/projects.php/id=28&lang=en`.

[21] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[22] Y. Gurevich, B. Rossman, W. Schulte, Semantic essence of AsmL, Theoretical Computer Science 343 (3) (2005) 370–412.

[23] Y. Gurevich, N. Tillmann, Partial updates: Exploration, Journal of Universal Computer Science 7 (11) (2001) 917–951.

[24] Y. Gurevich, N. Tillmann, Partial updates: Exploration, Journal of Universal Computer Science 7 (11) (2001) 917–951.
URL `http://www.jucs.org/jucs_7_11/partial_updates_exploration`

[25] Y. Gurevich, N. Tillmann, Partial updates exploration II, in: Abstract State Machines, 2003.
URL `citeseer.ist.psu.edu/gurevich03partial.html`

[26] Y. Gurevich, N. Tillmann, Partial updates, Theoretical Computer Science 336 (2-3) (2005) 311–342.

[27] M. Kirchberg, K.-D. Schewe, J. Zhao, Using Abstract State Machines for the design of multi-level transaction schedulers, in: J.-R. Abrial, U. Glässer (eds.), Rigorous Methods for Software Construction and Analysis – Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday, vol. 5115 of LNCS Festschrift, Springer, 2009, pp. 65–77.

[28] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Computers 28 (9) (1979) 690–691.

[29] D. B. Lomet, MLR: A recovery method for multi-level systems, in: M. Stonebraker (ed.), Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, ACM Press, 1992.

[30] M.Özsu, P. Valduriez, Principles of Distributed Database Systems, Prentice-Hall, 1994.

[31] C. Papadimitriou, The Theory of Database Concurrency Control, Computer Science Press, 1986.

[32] K.-D. Schewe, T. Ripke, S. Drechsler, Hybrid concurrency control and recovery for multi-level transactions, Acta Cybernetica 14 (3) (2000) 419–453.

[33] K.-D. Schewe, Q. Wang, Partial updates in complex-value databases, in: Information Modelling and Knowledge Bases XXII, 20th European-Japanese Conference on Information Modelling and Knowledge Bases (EJC 2010), Jyväskylä, Finland, 31 May - 4 June 2010, vol. 225 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2010.
URL http://dx.doi.org/10.3233/978-1-60750-690-4-37

[34] G. Weikum, Transaktionsverwaltung in Datenbanksystemen mit Schichtenarchitektur, Ph.D. thesis, TU Darmstadt (1986).

[35] G. Weikum, Principles and realization strategies of multilevel transaction management, ACM Transaction on Databbase Systems 16 (1) (1991) 132–180.