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

A R T I C L E  I N F O
Article history:
Received 16 February 2015
Received in revised form 22 March 2016
Accepted 22 March 2016
Available online 20 April 2016

Keywords:
Abstract state machines
Multi-level transactions
Partial updates
Serialisability

A B S T R A C T
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 behaviour. 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.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

This paper is about the use of generalised multi-level transactions as a means to control the consistency of concurrent access of programs to shared locations, 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 [6,35,36] in distributed databases [7,31]. A commonly accepted consistency criterion is that the joint behaviour of all transactions (i.e., programs running under transactional control) with respect to

* 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 (bmvtt), Austrian Ministry for Science, Research and Economy (bmwf), and the Province of Upper Austria in the frame of the COMET center SCCH [844597]. 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.
* Corresponding author.

E-mail addresses: boerger@di.unipi.it (E. Börger), klaus-dietrich.schewe@scch.at (K.-D. Schewe), qing.wang@anu.edu.au (Q. Wang).

http://dx.doi.org/10.1016/j.scico.2016.03.008
0167-6423/© 2016 Elsevier B.V. All rights reserved.
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 behaviour 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 behaviour of concurrent programs (i.e., components $M$ of any given asynchronous system $M$) into a transactional one. This involves to also define an operator $TA\langle \cdot, TaCtl \rangle$ that transforms a program $M$ into a new one $TA(M, TaCtl)$, 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 behaviour). 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 generalise the strictly hierarchical view of multiple levels by using the partial update concept for ASMs developed in [24] and further investigated in [27] and [34]. 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 on complex database values are dealt with in terms of compatibility of appropriate value changing operators (see also [28]). 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) generalising the flat transaction model to multi-level transactions which increase the concurrency in transactions and b) including an Abort mechanism. Accordingly, the serialisable proof is a refinement of the proof in [10], as the refined model is a conservative extension of the model for flat transactions.\footnote{For a detailed illustration of combined model and proof refinement we refer the reader to the Java compiler correctness verification in [5].}

We concentrate on transaction controllers that employ locking strategies such as the common two-phase locking protocol (2PL) [32]. 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, 28, 33] 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 [22] 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 referred 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, 35, 36] 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.

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 [33,35,36] 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 [34]) 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 [34]: a location \( l' \) subsumes a location \( l \) iff in all states \( S \) the value of \( l \), i.e. \( \text{eval}(l, S) \), uniquely determines the value of \( l' \), i.e. \( \text{eval}(l', S) \). 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 realisation 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 \( op(\text{eval}(l, S), v) \). 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 \( T A(M, \text{TACtL}) \) 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 \( T A(M, \text{TACtL}) \) 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 \( T A(M, \text{TACtL}) \) 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 serialisability 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 [33] 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 [35,33] 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 realised 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) of the commit or abortion of single machines—we use Abstract 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.
The transaction operator \( TA(\bullet, \text{TaCtl}) \)

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

\[
TA(M, \text{TaCtl}) = (TA(M_i, \text{TaCtl})_{i \in I}, \text{TaCtl})
\]

It remains to explain the definition of \( TA(M, \text{TaCtl}) \) below. TaCtl keeps a dynamic set \( TransAct \) of those machines \( M \), whose run it currently has to supervise. This is to guarantee that \( M \) operates in a transactional manner, until it has \textit{Terminated} its transactional behaviour (so that it can \textit{Commit} it). To turn the behaviour 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 \textit{Transactions} 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 \textit{history(M)} is needed, which for each step of \( M \) keeps track of the newly requested locks and of the recovery updates needed to \textit{Restore} the values of the locations \( M \) changed in this step. When \( M \) enters the set \textit{TransAct}, the \textit{history(M)} has to be initialised (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 \textit{LockedBy(M)} for this purpose; \( M \) tries to obtain such locks by calling the LockHandler. In case no \textit{newLocks} are needed by \( M \) in its \textit{currState} or the LockHandler \textit{GrantedLocksTo(M)}, \( M \) can \textit{Go} to try to perform its next step: if it cannot fire (due to an inconsistency of the set \textit{aggregatedUpdSet} \( \overset{5}{\text{of updates computed by M from the assignment and the partial update instructions of M, see below}} \) it calls the \textit{Abort} component. If \textit{CanFire}(M) holds, we require \textit{Fire}(M) to perform the \textit{M-step} together with one step of all \textit{Partner}(M)-machines, i.e., of machines \( N \) that \textit{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). This means to \textit{Aggregate} the (below called genuine) updates \( M \) yields in its \textit{currState(M)} together with the partial updates of \( M \) together with the genuine updates and partial updates of all \textit{Partner(M)}-machines. In addition a \textit{RecoveryRecord} component has to \textit{Record} for each of these machines \( N \) in its \textit{history} the obtained \textit{newLocks} together with the \textit{recoveryUpdates} needed should it become necessary to \textit{Undo} the updates contributed by \( N \) to this \textit{Aggregate}-step. Then \( M \) continues its transactional behaviour until it is \textit{Terminated}.

In case the LockHandler \textit{RefusedLocksTo(M)}, namely because another machine \( N \) in \textit{TransAct} has some of these locks, \( M \) has to \textit{Wait} for \( N \); in fact it continues its transactional behaviour by calling again the LockHandler for the needed \textit{newLocks}—until the needed locked locations are unlocked, when \( N \)'s transactional behaviour is \textit{Commited}, whereafter a new request for these locks \textit{GrantedLocksTo(M)} may become true.

As a consequence deadlock may occur, namely when a cycle occurs in the transitive closure \textit{Wait} of the \textit{Wait} relation. To resolve such deadlocks the DeadlockHandler component of TaCtl chooses some machines as \textit{Victims} for a recovery. After a victimised machine \( M \) is \textit{Recovered} by the \textit{Recovery} component of TaCtl it can exit its \textit{waitForRecovery} mode and continue its transactional behaviour.

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 [12, Ch. 2.2.6]). The macros which appear in Fig. 1 are defined in the rest of this section.

---

2 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 component machines and TaCtl. From the context it is always clear which \( TA \) we are talking about.

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

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

5 We borrow the name from CoreASM [16].

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

7 A refinement (in fact a desirable optimisation) 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 \( M \) \textit{simply stay in waitForLocks} until the \textit{newLocks CanBeGranted} and refining LockHandler to only choose pairs \( (M, L) \in \text{LockRequest} \) where it can \textit{canGrantedRequestedLocks(M, L)} and doing nothing otherwise (i.e. defining \textit{RefuseRequestedLocks(M, L) = skip}). See Sect. 4.

8 To simplify the serialisability proof in Sect. 4 and without loss of generality we define a reaction of machines \( M \) to their victimisation only when they are in \textit{ctl_state}(M) = \textit{TA-ctl} (not in \textit{ctl_state}(M) = \textit{waitForLocks}). This is to guarantee that no locks are \textit{Granted} to a machine as long as it does \textit{waitForRecovery}.

9 The components for the recovery feature are highlighted in the flowchart by a different colouring.
3.2. The \( TA(M, \text{TaCtl}) \) macros

The predicate \( \text{NewLocksNeededBy}(M) \) 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 \( \text{LockedBy}(M) \) for reading or writing, or \( M \) writes some shared or output location which is not yet \( \text{LockedBy}(M) \) for the requested write operation. We compute the set of such needed, but not yet \( \text{Locked} \) locations by a function \( \text{newLocks} \) (whose arguments we omit for layout reasons in Fig. 1).

\[
\text{NewLocksNeededBy}(M) = \text{newLocks}(M, \text{currState}(M)) \neq \emptyset
\]

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

\[
s \leftarrow t
\]

resulting for \( s = f(t_1, \ldots, t_n) \) in any given state \( S \) in an update of the location \( l = (f, (\text{eval}(t_1, S), \ldots, \text{eval}(t_n, S))) \) by the value \( \text{eval}(t, S) \) [12, pg. 29]. Here \( \text{eval}(t, S) \) denotes the evaluation of \( t \) in state \( S \) (under a given interpretation \( I \) of free variables). We call such updates \( (l, \text{val}) \text{ genuine} \) (in [34] they are called exclusive) to distinguish them from partial updates. The reader who is not knowledgeable about ASMs may interpret locations \( (f, \text{args}) \) correctly as array variables with variable name \( f \) and index \( \text{args} \).

Analogously, we denote write operations that involve partial updates via an operation \( \text{op} \) by *update instructions* of form

\[
s \leftarrow_{\text{op}} t
\]

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

\[
x \leftarrow \text{parCount} \times 7
\]

in a state \( S \) are aggregated into a genuine update \( (x, \text{eval}(x + 7, S)) \). Other examples are tree manipulation by simultaneous updates of independent subtrees or more generally term rewriting by simultaneous independent subterm updates, etc. see [33,34]. \text{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 \( \text{LockedBy}(M) \) for reading \((\text{Locked}(l, M, \text{Read}))\) or for writing \((\text{Locked}(l, M, \text{Write}))\) via a genuine update or for a partial update using operation \( \text{op} \) \((\text{Locked}(l, M, \text{op}))\). We also use \( \text{Locked}(l, M, \text{temp}) \) 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 \( \text{Locked}(l, M, \text{op}) \) temporarily \((\text{case} \ \text{op} = \text{temp})\) or for a partial update operation \((\text{case} \ \text{op} \neq \text{Read, Write})\) 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.

\[
\text{newLocks}(M, \text{currState}(M)) =
\begin{align*}
&\{(l, \text{Read}) \mid l \in R-\text{Loc} \text{ and } \\
&\quad \text{not Locked}(l, M, \text{Read}) \text{ and not Locked}(l, M, \text{Write})
\} \\
&\cup \{(l, o) \mid l \in W-\text{Loc} \text{ and } o \in \{\text{Write}\} \cup \text{Opn} \text{ and not Locked}(l, M, o)
\} \\
&\cup \{(l, \text{temp}) \mid 3\exists ! l' \in W-\text{Loc} \land l \text{ does not subsume } l'\}
\end{align*}
\]

\[\text{where}\]
\[
R-\text{Loc} = \text{ReadLoc}(M, \text{currState}(M)) \cap (\text{SharedLoc}(M) \cup \text{MonitoredLoc}(M))
\]
\[
W-\text{Loc} = \text{WriteLoc}(M, \text{currState}(M)) \cap (\text{SharedLoc}(M) \cup \text{OutputLoc}(M))
\]

\[
\text{LockedBy}(M) =
\begin{align*}
&\{l \mid \text{Locked}(l, M, \text{Read}) \text{ or Locked}(l, M, \text{Write}) \text{ or Locked}(l, M, \text{temp}) \text{ or } \\
&\quad \text{Locked}(l, M, \text{op}) \text{ forsome } o \in \text{Opn}\}
\end{align*}
\]

To \( \text{CALLLOCKHANDLER} \) for the \( \text{newLocks} \) requested by \( M \) in its \( \text{currState}(M) \) means to \( \text{INSERT} \) into the \( \text{LOCKHANDLER} \)'s set of to be handled \( \text{LockRequests} \). Similarly we let \( \text{CALLCOMMIT}(M) \) resp. \( \text{CALLABORT}(M) \) stand for insertion of \( M \) into a set \( \text{CommitRequest} \) resp. \( \text{AbortRequest} \) of the \( \text{COMMIT} \) resp. \( \text{ABORT} \) component.

\[
\text{CALLLOCKHANDLER}(M) = \text{INSERT}(M, \text{LockRequest})
\]
\[
\text{CALLCOMMIT}(M) = \text{INSERT}(M, \text{CommitRequest})
\]
\[
\text{CALLABORT}(M) = \text{INSERT}(M, \text{AbortRequest})
\]

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

\[
\text{CanFire}(M) = \text{Consistent}(\text{aggregatedUpdSet}(M, \text{currState}(M))).
\]

Here \( \text{aggregatedUpdSet}(M, S) \) is defined as the set of updates \( M \) yields\(^{11}\) in state \( S \), once the resulting genuine updates have been computed for all partial updates to be performed by \( M \) in \( S \).\(^{12}\) If this update set is consistent, to \( \text{Fire}(M) \) \( \text{AGGREGATE} \) performs not only the (genuine and partial) updates of \( M \), but also those of any other \( \text{Partner}(M) \)-machine \( N \) which shares some to-be-updated location with \( M \) and \( \text{CanFire}(N) \) simultaneously with \( M \).

\[
\text{Fire}(M) = \begin{align*}
&\text{forall } N \in \text{Partner}(M) \text{ do } N \\
&\text{RELEASETEMPLOCKS}(M)
\end{align*}
\]

\[\text{where}\]
\[
\text{Partner}(M) = \{N \mid \text{ShareUpdLocs}(M, N) \text{ and } \text{mode}(N) = \text{canFire}\}
\]
\[
\text{RELEASETEMPLOCKS}(M) =
\begin{align*}
&\text{forall } l \text{ Locked}(l, M, \text{temp}) := \text{false}
\end{align*}
\]

The constraints defined in the next section for \( \text{GrantedLocksTo}(M) \) and the consistency condition for \( \text{aggregatedUpdSets} \) guarantee that \( \text{Fire}(M) \) computes and performs a consistent update set.

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 \( \text{ReadLoc, WriteLocs} \)) and relations (like subsumption) can be

---

\(^{10}\) By the second argument \( \text{currState}(M) \) of \( \text{newLocks} \) 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 \( \text{ReadLoc}(M) \) and \( \text{WriteLoc}(M) \); see Sect. 5 for the detailed definition.

\(^{11}\) See the definition in \([12, \text{Table 2.2 pg. 74}]\).

\(^{12}\) In CoreASM [16] this computation is done by corresponding plug-ins.
mathematically defined and used in rules. In accordance with usual linguistic reflection notation we also quantify over such elements, e.g. in \texttt{forall }\texttt{N }\in \texttt{SetOfAsm} \texttt{do }\texttt{N}, meaning that \texttt{do }\texttt{N} stands for an execution of (a step of) the ASM denoted by the logical variable \texttt{N}.

The \texttt{RECOVERY RECORD}(\texttt{M}) component has to \texttt{RECORD} for each Partner(\texttt{M})-machine its \texttt{recovery Updates} (defined below where we need the details for the RECOVERY machine) and the obtained \texttt{newLocks}.

\[
\texttt{RECOVERY RECORD}(\texttt{M}) = \texttt{forall }\texttt{N }\in \texttt{Partner}(\texttt{M}) \texttt{record}(\texttt{recovery Update}(\texttt{N}, \texttt{currState}(\texttt{N})), \texttt{newLocks}(\texttt{N}, \texttt{currState}(\texttt{N})), \texttt{N}) \texttt{record}(\texttt{recUpdateSet}, \texttt{lockSet}, \texttt{N}) = \texttt{APPEND}(\texttt{recUpdateSet}, \texttt{lockSet}, \texttt{history}(\texttt{N}))
\]

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

\[M = \texttt{choose }x \texttt{with }\alpha \texttt{in }r(x)\]

where \texttt{r} has to be an ASM rule. Nondeterminism can also be modelled ‘from outside’ by using choice functions, say \texttt{select}, in machines of form

\[N = r(\texttt{select}(\alpha))\]

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

\section{4. The transaction controller components}

\subsection{4.1. The Commit component}

A \texttt{CallCommit} (\texttt{M}) by machine \texttt{M} enables the \texttt{Commit} component, which handles \texttt{Commit Requests} one at a time. For this we use the \texttt{choose} operator, so we can leave the order in which the \texttt{Commit Requests} are handled refinable by different instantiations of \texttt{TaCtl}.

Committing \texttt{M} means to \texttt{Unlock} all locations \texttt{l} that are \texttt{LockedBy}(\texttt{M}).\footnote{\texttt{Unlock} is called only in states where \texttt{M} has no \texttt{temporary lock}.} Note that each lock obtained by \texttt{M} remains with \texttt{M} until the end of \texttt{M}’s transactional behaviour. Since \texttt{M} performs a \texttt{CallCommit} (\texttt{M}) when it has \texttt{Terminated} its transactional computation, nothing more has to be done to \texttt{Commit} \texttt{M} besides deleting \texttt{M} from the sets of \texttt{Commit Requests} and still to be handled \texttt{TransActions}.

\[
\texttt{commit} =
\begin{array}{l}
\texttt{if }\texttt{Commit Request} \neq \emptyset \texttt{then }\\
\texttt{choose }\texttt{M }\in \texttt{Commit Request }\texttt{ Commit} (\texttt{M})\\
\texttt{where }\\
\texttt{Commit} (\texttt{M}) =\\
\texttt{forall }l \in \texttt{LockedBy}(\texttt{M}) \texttt{ Unlock} (l, \texttt{M})\\
\texttt{DELETE} (\texttt{M}, \texttt{Commit Request})\\
\texttt{DELETE} (\texttt{M}, \texttt{TransAct})\\
\texttt{Unlock} (l, \texttt{M}) = \texttt{forall }o \in \{\texttt{Read}, \texttt{Write}\} \cup \texttt{Opn}\\
\texttt{Locked} (l, \texttt{M}, o) := \texttt{false}
\end{array}
\]

The locations (\texttt{Locked} (\texttt{l}, \texttt{M}, \texttt{o})) are shared by the \texttt{Commit}, \texttt{LockHandler} and \texttt{RECOVERY} components, but these components never have the same \texttt{M} simultaneously in their request or \texttt{Victim} set, respectively: When \texttt{M} has performed a \texttt{CallCommit} (\texttt{M}), it has \texttt{Terminated} its transactional computation and does not participate any more in any \texttt{Lock Request} or \texttt{Victimisation}. Furthermore, by definition no \texttt{M} can at the same time issue a \texttt{Lock Request} (possibly triggering the \texttt{LockHandler} component) and be a \texttt{Victim} (possibly triggering the \texttt{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 LockRequests 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 WriteLocation 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 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 [34]:

- 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 serialisability 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 [34, Def. 2.1]:

\[ l' \text{ subsumes } l = \text{forall } S \text{ eval}(l', S) \text{ uniquely determines eval}(l, S) \]

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 \( M \).

\[
\text{LockHandler} = \\
\text{if LockRequest } \neq \emptyset \text{ then } \\
\text{choose } M \in \text{LockRequest HandleLockREQUEST}(M) \\
\text{where} \\
\text{HANDLELOCKREQUEST}(M) = \\
\text{let } L = \text{newLocks}(M, \text{currState}(M)) \\
\text{if CannotBeGranted}(M, L) \\
\text{then RefuseRequestedLocks}(M, L) \\
\text{else GrantRequestedLocks}(M, L) \\
\text{DELETE}((M, L), \text{LockRequest}) \\
\text{CannotBeGranted}(M, L) = \\
\text{forall } (l, o) \in L \text{ CannotBeGranted}(l, M, o) \\
\text{CannotBeGranted}(l, M, o) = \\
\text{forall } N \in \text{TransAct} \setminus \{M\} \text{ Blocks } (N, l, o) \\
\text{Blocks } (N, l, o) = \text{forall } o' \\
\text{Locked}(l, N, o') \text{ and not Compatible}(o, o', l) \\
\text{or forall } l' \text{ Locked}(l', N, o') 	ext{ and } l' \text{ subsumes } l \\
\text{RefuseRequestedLocks}(M, L) = \text{(RefusedLocksTo}(M) := \text{true}) \\
\text{GrantRequestedLocks}(M, L) = \\
\text{forall } (l, o) \in L \text{ Locked}(l, M, o) := \text{true} \\
\text{GrantedLocksTo}(M) := \text{true}
\]

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 Victimised) machine \( M \) the pair \((M, M)\) is in the transitive (not reflexive) closure \( \text{Wait}^* \) of Wait. In this case the DeadlockHandler selects for recovery a (typically minimal) subset of Deadlocked transactions toResolve—they are Victimised 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.

\[
\text{DeadlockHandler} =
\begin{array}{l}
\text{if Deadlocked} \cap \overline{\text{Victim}} \neq \emptyset \text{ then } // \text{ there is a Wait cycle}
\end{array}
\]

choose toResolve \subseteq \text{Deadlocked} \cap \overline{\text{Victim}}

forall M \in \text{toResolve} \text{ Victim(M) := true}

where

Deadlocked = \{ M \mid (M, M) \in \text{Wait}^* \}

\text{Wait} = \text{TransitiveClosure} (\text{Wait})

\text{Wait}(M, N) = \text{for some} (l, o) \text{ Wait}(M, (l, o), N)

\text{Wait}(M, (l, o), N) =

(l, o) \in \text{newLocks}(M, \text{currState}(M)) \text{ and } N \in \text{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 Victims 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.

\[
\text{Recovery} =
\begin{array}{l}
\text{if Victim} \neq \emptyset \text{ then }
\end{array}
\]

choose M \in \text{Victim} \text{ TryToRecover(M)}

where

TryToRecover(M) =

if M \neq \text{Deadlocked} \text{ then } \text{Victim(M) := false}

else \text{ Undo(M)}

Recovered =

\{ M \mid \text{ctl-state}(M) = \text{waitForRecovery and } M \notin \text{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 genuineRecoveryUpdates by which one can restore the overwritten values in GenuineWriteLocations (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 \( s :=_{op} t \in \text{UpdInstr}(M, S) \) 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 [34] and adopted here one has to postulate that such operations and values \( (op', v') \) which are inverse to partial update operations \( (op, v) \) (where \( v = \text{eval}(t, \text{currState}(M)) \)) are defined and satisfy the following constraint for partial update instructions:

\[
\text{Inverse Operation Postulate}
\]

forall \( s :=_{op} t \in \text{UpdInstr}(M) \) forall \( (op, v) \) there is \( (op', v') \) with

forall \( w \) \( op'(op(w, v), v') = w \)

This postulate can be justified by the requirement that any transaction should be recoverable [33]. 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 foreseen that it may be undone, even if the issuing 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 [34]) has to be complemented by the inverse operation postulate to ensure recoverability. Inverse operators are claimed in the MLR recovery system [30], 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 partialRecoveryUpdates have to be aggregated with the genuineRecoveryUpdates by RESTORE.

\[
\text{recoveryUpd}(M, S) =
\text{genuineRecoveryUpd}(M, S), \text{partialRecoveryUpd}(M, S)
\]

\[
\text{genuineRecoveryUpd}(M, S) = \{(f, \text{args}, \text{val}) | (f, \text{args}) \in \text{GenuineWriteLoc}(M, S) \land \text{val} = \text{eval}(f(\text{args}), S)\}
\]

\[
\text{partialRecoveryUpd}(M, S) = \{(l, (\text{op}', \text{v}')') | \text{some } f(t_1, \ldots, t_n) := \text{op} f \in \text{UpdInstr}(M, S) \land l = (f, \text{eval}(t_1, S), \ldots, \text{eval}(t_n, S)) \land (\text{op}', \text{v}') = \text{inverse}(\text{op}, \text{eval}(t, S))\}
\]

\[
\text{Undo}(M) = \text{let } (\text{Ups}, \text{Locks}) = \text{youngest}(\text{history}(M)) \land \text{Restore}(\text{Ups}, M) \land \text{Release}(\text{Locks}, M) \land \text{Delete}((\text{Ups}, \text{Locks}), \text{history}(M))
\]

\[
\text{where }
\text{Restore}(G, P, M) = \text{AGGREGATE}(G \cup // \text{NB. } P \text{ is a multiset }
\text{[]}((f, \text{args}), \text{op}' \text{eval}(f(\text{args}), \text{currState}(M)), \text{v}') |
\{(f, \text{args}), (\text{op}', \text{v}') \in P \}) // \text{NB. multiset notation } \text{[]} \text{[]}
\]

\[
\text{Release}(L, M) = \text{forall } l \in L. \text{Unlock}(l, M)
\]

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 \((\text{op}_1', \text{v}_1')\) be inverse for \((\text{op}_1, \text{v}_1)\) for \(i = 1, 2\), such that both operations \(\text{op}_i\) are compatible and both inverse operations \(\text{op}_i'\) 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 \(\text{op}_1'(\text{op}_2'(v, \text{v}_2), \text{v}_1') = \text{op}_2'(\text{op}_1'(v, \text{v}_1'), \text{v}_2')\). As these are Undo operations, we can assume that \((\text{op}_i, \text{v}_i)\) for \(i = 1, 2\) have been executed on some previous value of location \(l\). Thus, due to commutativity we must have \(v = \text{op}_1(\text{op}_2(\text{v'}, \text{v}_2), \text{v}_1) = \text{op}_2(\text{op}_1(\text{v'}, \text{v}_1), \text{v}_2)\) for some value \(\text{v'}\). From this we get

\[
\text{op}_1'(\text{op}_2'(v, \text{v}_2), \text{v}_1') = \text{op}_1'(\text{op}_2'(\text{op}_1(\text{v'}, \text{v}_1), \text{v}_2), \text{v}_1') = \text{op}_1'(\text{op}_1(\text{v'}, \text{v}_1), \text{v}_1') = \text{v'} = \text{op}_2'(\text{op}_2(\text{v'}, \text{v}_2), \text{v}_2') = \text{op}_2'(\text{op}_1(\text{op}_2(\text{v'}, \text{v}_2), \text{v}_1), \text{v}_1') = \text{op}_2'(\text{op}_1(\text{v'}, \text{v}_1), \text{v}_1')
\]

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]:

\[
\text{Abort} = \text{forall } M \in \text{AbortRequest}
\text{iterate } \text{Undo}(M) \text{ until } \text{history}(M) = \emptyset
\text{Delete}(M, \text{TransAct})
\]

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
Each run of $T(\mathcal{M}, \text{TaCtl})$ is equivalent to a serialisation of the terminating $M_i$-runs, namely the $M_{i-1}$-run followed by the $M_{i-2}$-run etc., where $M_i$ is the $j$-th machine of $\mathcal{M}$ which performs a commit in the $T(\mathcal{M}, \text{TaCtl})$ 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 $\text{TransAct}$ of machines which currently run under the control of $\text{TaCtl}$.

First of all we have to make precise what a serial multi-agent ASM run is and what equivalence of $T(\mathcal{M}, \text{TaCtl})$ 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 $T(\mathcal{M}, \text{TaCtl})$. In general we may assume that $\text{TaCtl}$ runs forever, whereas each machine $M \in \mathcal{M}$ running as transaction will be Terminated or Aborted at some time—once Committed $M$ will only change values of non-shared and non-output locations. 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 $\Delta_i, \Gamma_i$ 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 $T(\mathcal{M}, \text{TaCtl})$ each $\Delta_i, \Gamma_i$ is the union of the corresponding sets resp. multisets of the agents executing $M \in \mathcal{M}$ resp. $\text{TaCtl}$:

$$\Delta_i = \bigcup_{M \in \mathcal{M}} \Delta_i(M) \cup \Delta_i(\text{TaCtl}) \quad \Gamma_i = \bigcup_{M \in \mathcal{M}} \Gamma_i(M) \cup \Gamma_i(\text{TaCtl}).$$

$\Delta_i(M)$ contains the genuine and $\Gamma_i(M)$ the partial updates defined by the machine $T(\mathcal{M}, \text{TaCtl})$ in state $S_i$, $\Delta_i(\text{TaCtl})$ resp. $\Gamma_i(\text{TaCtl})$ 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 transactional 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 $\text{Undo}(M)$ 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 $\text{ctlState}(M) = \text{Ta-ctl}$ does CallLockHandler($M$) and in its next step moves from $\text{waitForLocks}$ back to control state TA-ctl because the LockHandler RefusedLocksTo($M$).

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 $\text{waitForRecovery}$ 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) $\Delta_j(M), \Gamma_j(M)$ and $\Delta_j'(M), \Gamma_j'(M),$
- (b) the {(Victim, $M$), true} update from the corresponding $\Delta_i(\text{TaCtl})$ ($t < j$) which in state $S_j$ triggered the $M$-RecoveryEntry,
- (c) $\text{TryToRecover}(M)$-updates in any $\Delta_{i+k}(\text{TaCtl}), \Gamma_{i+k}(\text{TaCtl})$ between the considered $M$-RecoveryEntry and $M$-RecoveryExit step (for $i$ as below with $i < j < i + k < j'$).

---

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

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

[17] We indicate multiset operations by an upper index $\cup$.

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

[19] By eliminating this $\text{CallLockHandler}(M)$ step also the corresponding $\text{LockHandler}$ step $\text{HandleLockRequest}(M)$ disappears in the run.
(d) each $\Delta_{\ell}(M), \Gamma_{\ell}(M)$ 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 $\Delta_{\ell}(M)$ contain genuine updates $(\ell, v)$ with $\ell = (f, (eval(t_1, S_1), \ldots, eval(t_n, S_1)))$ where the corresponding UNDO updates are

$$\ell, eval(f(t_1, \ldots, t_n), S_i)) \in \Delta_{i+k}(TA\text{CTL})$$

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

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

for update instructions $f(t_1, t_2, \ldots) :=_\text{op} t$ of $M$ in $S_{i'}$ whose effect is UNDONE when RECOVERY AGGREGATES the $M$-specific partial update $(I, (op', v'))$ with the inverse operation $(op', v')$ to $(op, eval(t, S_{i'}))$ on $I$.

(e) the $HANDLE\text{LockRequest}(M)$-updates in $\Delta_{p}(TA\text{CTL})$ corresponding to $M$'s $\text{CallLockHandler}$ step (if any: in case newLocks are needed for the proper $M$-step in $S_j$) in state $S_l$ ($l < l' < i$).

The sequence $\Delta_{p}(M), \Gamma_{p}(M), \Delta_{s}(M), \Gamma_{s}(M), \ldots$ 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 $TA(M, TA\text{CTL})$ 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 $TA\text{CTL}$ are non-deterministic:

- In case several machines $M, M' \in M$ request conflicting locks at the same time, the $\text{LockHandler}$ can only grant the requested locks for one of these machines.
- Commit requests are executed in random order by the $\text{Commit}$ submachine.
- The submachine $\text{DeadLockHandler}$ chooses a set of victims, and this selection has been deliberately left abstract.
- The $\text{Recovery}$ submachine chooses in each step a victim $M$, for which the last step is UNDO together with releasing corresponding locks.

Second, the specification of $TA(M, TA\text{CTL})$ leaves deliberately open, when a machine $M \in M$ will be started, i.e., register as a transaction in $TransAct$ to be controlled by $TA\text{ctl}$. This is in line with the common view that transactions $M \in M$ can register at any time to the transaction controller $TA\text{ctl}$ 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 $TA(M, TA\text{CTL})$ are equivalent iff for each $M \in M$ the cleansed schedules

$$\Delta_{i}(M), \Gamma_{i}(M), \Delta_{s}(M), \Gamma_{s}(M), \ldots$$

and

$$\Delta'_{i}(M), \Gamma'_{i}(M), \Delta'_{s}(M), \Gamma'_{s}(M), \ldots$$

for the two runs are the same and the read locations and the values read by $M$ in $S_i$ and $S'_i$ are the same.

That is, we consider runs to be equivalent, if all transactions $M \in M$ 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 serialisability

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.\textsuperscript{20} As stated above Anorced 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. $\text{ReadLoc}(M, S)$ and $\text{WriteLoc}(M, S)$ as used in the definition of $\text{newLocks}(S, M)$.

We define $\text{ReadLoc}(M, S) = \text{ReadLoc}(r, S)$ and analogously $\text{WriteLoc}(M, S) = \text{WriteLoc}(r, S)$, 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 formulæ. 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 $\text{ReadLoc}(M, S), \text{WriteLoc}(M, S)$ instead of $\text{ReadLoc}(M, S, I), \text{ReadLoc}(M, S, I)$ respectively.

\textsuperscript{20} 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.
5.2.1. Read/write locations of terms and formulae

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

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 \textbf{let}, \textbf{forall}, \textbf{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 \( \text{WriteLoc}(\alpha, S) = \emptyset \) for every formula \( \alpha \) because formulae are not locations one could write into. \( \text{ReadLoc}(\alpha, S) \) 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 \( \text{domain}(S) \) denotes the superuniverse of \( S \) minus the Reserve set \([12, \text{Ch. 2.4.4}]\) and \( I^d_x \) the extension (or modification) of \( I \) where \( x \) is interpreted by a domain element \( d \).

\[
\text{ReadLoc}(P(t_1, \ldots, t_n), S) = \\
\{(P, \text{eval}(t_1), \ldots, \text{eval}(t_n), S))\} \cup \bigcup_{1 \leq i \leq n} \text{ReadLoc}(t_i, S) \\
\text{ReadLoc}(\neg \alpha) = \text{ReadLoc}(\alpha) \\
\text{ReadLoc}(\alpha_1 \land \alpha_2) = \text{ReadLoc}(\alpha_1) \cup \text{ReadLoc}(\alpha_2) \\
\text{ReadLoc}(\forall x \alpha, S, I) = \bigcup_{d \in \text{domain}(S)} \text{ReadLoc}(\alpha, S, I^d_x)
\]

The values of the logical variables are not read from a location but from the modified state environment function \( I^d_x \).

5.2.2. Read/write locations of ASM rules

\[ \text{ReadLoc}(\text{skip}, S) = \text{WriteLoc}(\text{skip}, S) = \emptyset \]
\[
\text{ReadLoc}(t_1 := t_2, S) = \text{ReadLoc}(t_1 :=_{\text{op}} t_2, S) = \\
\text{ReadLoc}(t_1, S) \cup \text{ReadLoc}(t_2, S) \\
\text{WriteLoc}(t_1 := t_2, S) = \text{WriteLoc}(t_1 :=_{\text{op}} t_2, S) = \text{WriteLoc}(t_1, S) \\
\text{ReadLoc}(\text{if } \alpha \text{ then } r_1 \text{ else } r_2, S) = \\
\text{ReadLoc}(\alpha, S) \cup \\
\begin{cases} 
\text{ReadLoc}(r_1, S) \text{ if eval}(\alpha, S) = \text{true} \\
\text{ReadLoc}(r_2, S) \text{ else}
\end{cases}
\]
\[
\text{WriteLoc}(\text{if } \alpha \text{ then } r_1 \text{ else } r_2, S) = \\
\begin{cases} 
\text{WriteLoc}(r_1, S) \text{ if eval}(\alpha, S) = \text{true} \\
\text{WriteLoc}(r_2, S) \text{ else}
\end{cases}
\]
\[
\text{ReadLoc}(\text{let } x = t \text{ in } r, S, I) = \text{ReadLoc}(r, S, I^\text{eval}(t, S)) \\
\text{WriteLoc}(\text{let } x = t \text{ in } r, S, I) = \text{WriteLoc}(r, S, I^\text{eval}(t, S)) \text{ call by value} \\
\text{ReadLoc}(\text{forall } x \text{ with } \alpha \text{ do } r, S, I) = \\
\text{ReadLoc}(\forall x \alpha, S, I) \cup \bigcup_{d \in \text{range}((x, \alpha), S)} \text{ReadLoc}(r, S, I^d_x) \\
\begin{cases} 
\text{where range}(x, \alpha, S, I) = \{d \in \text{domain}(S) \mid \text{eval}(\alpha, S, I^d_x) = \text{true}\} \\
\text{WriteLoc}(\text{forall } x \text{ with } \alpha \text{ do } r, S, I) = \bigcup_{d \in \text{range}((x, \alpha), S)} \text{WriteLoc}(r, S, I^d_x)
\end{cases}
\]

In the following cases the same scheme applies to read and write locations:\footnote{In yields(r1, S, I, U) \( U \) denotes the update set produced by rule \( r_1 \) in state \( S \) under \( I \).}

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

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

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

**Definition 5.2.** A run of \( T_A(M, \text{TaCtrl}) \) is serial iff there is a total order \( < \) on \( M \) 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 \( \Delta_j(M) \), \( \Gamma_j(M) \), \( \Delta_j(M') \), \( \Gamma_j(M') \), \ldots of M' satisfies \( i < j \).

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 \( TA(\mathcal{M}, \text{TaCtl}) \) is serialisable iff it is equivalent to a serial run of \( TA(\mathcal{M}, \text{TaCtl}) \).

5.3. Serialisability proof

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

**Proof.** Let \( S_0, S_1, S_2, \ldots \) be a run of \( TA(\mathcal{M}, \text{TaCtl}) \). To construct an equivalent serial run let \( M_1 \in \mathcal{M} \) be a machine that commits first in this run, i.e. Commited(\( M, S_i \)) holds for some i and whenever Commited(\( M, S_j \)) 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 \( TA(\{M_1\}, \text{TaCtl}) \) 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 Deleted from \( \text{TransAct} \) and none of the \( \text{TaCtl} \) components is triggered any more: neither \( \text{CommitRequest} \) resp. \( \text{LockRequest} \) remain empty; not \( \text{LockHandler} \) because \( \text{Deadlock} \) remains false since \( M_1 \) never \( \text{Waits} \) for any machine; not \( \text{Recovery} \) because \( \text{Victim} \) remains empty. Note that in this run the schedule for \( M_1 \) is already cleansed.

We now define a run \( S_0'', S_1'', S_2'', \ldots \) (of \( TA(\mathcal{M} - \{M_1\}, \text{TaCtl}) \), as has to be shown) which starts in the final state \( S_n'' = S_0'' \) of the \( TA(\{M_1\}, \text{TaCtl}) \) 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 \( \text{TaCtl} \) steps which concern \( M_1 \) (i.e. which are related to a lock or commit request by \( M_1 \) or a victimisation of \( M_1 \) or a \( \text{TryToRecover}(M_1) \) step). Let

\[
\Delta_i'' = \bigcup_{M \in \mathcal{M} - \{M_1\}} \Delta_i(M) \cup \{(\ell, v) \in \Delta_i(\text{TaCtl}) | (\ell, v) \text{ does not concern } M_1\},
\]

\[
\Gamma_i'' = \bigcup_{M \in \mathcal{M} - \{M_1\}} \Gamma_i(M) \cup \{(\ell, v) \in \Gamma_i(\text{TaCtl}) | (\ell, v) \text{ does not concern } M_1\}.
\]

That is, in \( \Delta_i'', \Gamma_i'' \) 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 \( \text{LockHandler} \) updates involving a \( \text{LockRequest}(M_1) \) or are \( \text{Victim}(M_1) := \text{true} \) updates of the \( \text{DeadlockHandler} \) or are updates involving a \( \text{TryToRecover}(M_1) \) step or are done by a step involving a \( \text{Commit}(M_1) \).

**Lemma 5.2.** \( S_0'', S_1'', S_2'', \ldots \) is a run of \( TA(\mathcal{M} - \{M_1\}, \text{TaCtl}) \).

**Lemma 5.3.** The run \( S_0, S_1', S_2', \ldots, S_n', S_1'', S_2'', \ldots \) of \( TA(\mathcal{M}, \text{TaCtl}) \) 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 \( M \in \mathcal{M} - \{M_1\} \). \( \square \)

**Proof of Lemma 5.2.** Omitting in \( \Delta_i'', \Gamma_i'' \) from \( \Delta_i(\text{TaCtl}), \Gamma_i(\text{TaCtl}) \) every update which concerns \( M_1 \) leaves updates by \( \text{TaCtl} \) in \( S'' \) concerning \( M \neq M_1 \).

It remains to show that every \( \text{Fire}(M) \)-step defined by \( \Delta_i(M), \Gamma_i(M) \) is a possible \( \text{Fire}(M) \)-step via \( \Delta_i'', \Gamma_i'' \) in a \( TA(\mathcal{M} - \{M_1\}, \text{TaCtl}) \) 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 \( S'' \) (together with its preceding lock request step, if any).

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

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

Case 1.2. The \( \text{newLocks} \) 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 \( \text{newLocks} \) for locations but none of them is \( \text{LockedBy}(M_1) \). Then this \( M \)-step can be shifted like in Case 1.2.

---

22 Modulo the fact that ASM steps permit simultaneous updates of multiple locations, for ASMs with only genuine updates this definition of serialisability is equivalent to Lamport's sequential consistency concept [29].
Case 3. \( M \) for its step needs no \textit{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.  

\textbf{Proof of 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 \( S_0, S'_1, S'_2, \ldots, S'_n, S'_i, S'_i, \ldots, S'_i \) all these empty update sets disappear. Thus this cleansed schedule is the same as the cleansed schedule of \( M \) for the run \( S'_0, S'_1, S'_2, \ldots \) 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 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  
\[
\Delta_i''(M_1), \Gamma_i''(M_1), \Delta_i''(M_1), \Gamma_i''(M_1), \ldots
\]
We proceed by induction on the cleansed schedule steps of \( M_1 \). When \( M_1 \) makes its first step using the updates in \( \Delta_i''(M_1), \Gamma_i''(M_1) \), this can only be a \textit{Fire}(M) step together with the corresponding RecoveryRecord updates (or a lock request directly preceding such a \( \Delta_i''(M_1), \Gamma_i''(M_1) \)-step) because in the computation with cleansed schedule each lock request of \( M_1 \) is granted and \( M_1 \) is not \textit{Victimised}. 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 Unlock 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 \( \Delta_i''(M_1), \Gamma_i''(M_1) \) (resp. also \( \Delta_i''(M_1), \Gamma_i''(M_1) \)) 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 \textit{newLocks} resp. from \( S_0 \) to \( S'_1 \)) (otherwise). The shared update of \( M_1 \) are the same in both runs by definition. The same argument applies in the inductive step which establishes the claim.  

\textbf{6. Conclusion}  

In this article we specified in terms of Abstract State Machines a multi-level transaction controller TaCtLt and a multi-level transaction operator which turns the behaviour of a set of concurrent programs into a transactional one under the control of TaCtLt. 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 behaviour 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 generalisation 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 subsumption criterion. Note however that the update instruction set concept in CoreASM realises the concept of partial updates as used here and defined in [34]. 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 [28] and to other approaches to transaction handling, e.g. [14,21,28,33]. Last but not least we would like to see further details 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 \( TA(M, TaCtLt) \) (as suggested in [18]).
Acknowledgements

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 modelling 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 [25]. 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 [23]. 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 [25]. To develop a systematic 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 labelled ordered trees as exemplified in [26]. This limitation led to the formalisation of applicative algebras as a general solution to partial updates [27]. It was shown that the problem of partial updates over sequences and labelled ordered trees can be solved in this more general algebraic framework, and the approach in [25] was a special kind of an applicative algebra.

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

1. \(p(\bot) = \bot\) for each particle \(p\), and \(\lambda(x) = \bot\) 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 [34]:

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

**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 partial update \((l, v, op)\), we obtain a new state \(S'\) by applying the partial update \((l, v, op)\) over \(S\) and \(\text{eval}(l, S') = op(\text{eval}(l, S), v)\).

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 \(P_l = \{(l, v_i, op_i) \mid 1 \leq i \leq k\}\) 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 \((\sigma_1, \ldots, \sigma_k)\) and \((\pi_1, \ldots, \pi_k)\) of \(\{1, \ldots, k\}\), we have the following for all \(x\):

\[ op_{\sigma_1}(\ldots op_{\sigma_2}(op_{\sigma_1}(x, v_{\sigma_1}), v_{\sigma_2}), \ldots, v_{\sigma_k}) = op_{\pi_1}(\ldots op_{\pi_2}(op_{\pi_1}(x, v_{\pi_1}), v_{\pi_2}), \ldots, v_{\pi_k}) \]

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 \( P_1 \) is operator-compatible, then an order-independent sequential composition \( \Theta \) of the partial update operations in \( P_1 \) (written as \( \Theta P_1 \)) is equivalent to applying all the partial updates in \( P_1 \) sequentially in any order. That is, \( \Theta P_1(x) = op_{\sigma_1}(...op_{\sigma_2}(op_{\sigma_1}(x, v_{\sigma_1}), v_{\sigma_2}), ..., v_{\sigma_{|P_1|}}) \) for any permutation \( (\sigma_1, ..., \sigma_{|P_1|}) \) of \( \{1, ..., |P_1|\} \).

Therefore, if an update multiset \( P_1 \) is consistent, then all the partial updates in \( P_1 \) can be aggregated into one genuine update on the same location \( l \). A state \( S' \) can be obtained from \( S \) by applying the multiset \( P_1 \) of partial updates sequentially, and we have \( \text{eval}(l, S') = \Theta P_1(\text{eval}(l, S)) \).

References