An Epistemic Perspective on Consistency of Concurrent Computations

Klaus v. Gleissenthall[1] and Andrey Rybalchenko[1,2]

[1] Technische Universität München
[2] Microsoft Research Cambridge

Abstract. Consistency properties of concurrent computations, e.g., sequential consistency, linearizability, or eventual consistency, are essential for devising correct concurrent algorithms. In this paper, we present a logical formalization of such consistency properties that is based on a standard logic of knowledge. Our formalization provides a declarative perspective on what is imposed by consistency requirements and provides some interesting unifying insight on differently looking properties.

1 Introduction

Writing correct distributed algorithms is notoriously difficult. While in the sequential case, various techniques for proving algorithms correct exist [14, 17], in the concurrent setting, due to the nondeterminism induced by scheduling decisions and transmission failures, it is not even obvious what correctness actually means. Over the years, a variety of different *consistency properties* restricting the amount of tolerated nondeterminism have been proposed [10–12, 16, 18, 19]. These properties range from simple properties like sequential consistency [16] or linearizability [10, 12] to complex conditions like eventual consistency [19], a distributed systems condition. Reasoning about these properties is a difficult, yet important task since their implications are often surprising.

Currently, the study of consistency properties and the development of reasoning tools and techniques for such properties [4, 6, 20] is done for each property individually, i.e., on a per property basis. To some extent, this trend might be traced back to the way consistency properties are formulated. Typically, they explicitly require existence of certain computation traces that are obtained by rearrangement of the trace that is to be checked for consistency, i.e., these descriptions of consistency properties do not rely on a logical formalism. While such an approach provides fruitful grounds for the design of specialized algorithms and efficient tools, it leaves open important questions such as how various properties relate to each other or whether advances in dealing with one property can be leveraged for dealing with other properties.

In contrast to the trace based definitions found in literature, we propose to study consistency conditions in terms of epistemic logic [7, 9]. Here we can rely on a distributed knowledge modality [7], which is a natural fit for describing distributed computation. In this logic, an application $D_G(\varphi)$ of the distributed

knowledge modality to a formula φ denotes the fact that a group G *knows* that a formula φ holds.

We present a logical formalization of three consistency properties: the classical sequential consistency [16] and linearizability [10, 12], as well as a recently proposed formulation [5] of eventual consistency for distributed databases [19]. Our characterizations show that moving the viewpoint from reasoning about traces (models) to reasoning about knowledge (logic) can lead to new insights. When formulated in the logic of knowledge, these differently looking properties agree on a common schematic form: $\neg D_G(\neg \mathit{correct})$. According to this schematic form, a computation satisfies a consistency property if and only if a group G of its participants, i.e., threads or distributed nodes, *do not know that* the computation violates a specification *correct* that describes computations from the *sequential* perspective, i.e., without referring to permutations thereof. For example, when formalising sequential consistency of a concurrent register *correct* only states that the first read operation returns zero and any subsequent read operation returns the value written by the latest write operation.

The common form of our characterizations exposes the differences between the consistency properties in a formal way. A key difference lies in the group of participants that provides knowledge for validating the specification *correct*. For example, a computation is sequentially consistent if it satisfies the formula $\neg D_{\textsc{Threads}}(\neg \mathit{correct})$, i.e., the group G of agents needed to validate the sequential specification comprises the group of threads Threads accessing the shared memory. Surprisingly, the same group of agents is needed to validate eventual consistency, since in our logic it is characterized by the formula $\neg D_{\textsc{Threads}}(\neg \mathit{correctEVC})$. This reveals an insight that eventual consistency is actually not an entirely new consistency condition, but rather an instance of sequential consistency that is determined by a particular choice of *correct*. In contrast to the two above properties, the threads' knowledge is not enough to validate linearizability. To capture linearizability, the set of participants G needs to go beyond the participating threads Threads and include an additional observer thread *obs* as well. The observer only acquires knowledge of the relative order between returns and calls. As logical characterization of linearizability, we obtain $\neg D_{\textsc{Threads}\cup\{obs\}}(\neg \mathit{correct})$.

We show that including the observer induces a different kind of knowledge, i.e., it weakens the modal system from *S5* to *S4* [21]. As a consequence, the agents lose certainty about their decision whether or not a trace is consistent. For sequential consistency (*seqCons*) the agents know whether or not a trace is sequentially consistent, i.e., the formula $(\mathit{seqCons} \leftrightarrow D_{\textsc{Threads}}(\mathit{seqCons})) \wedge (\neg \mathit{seqCons} \leftrightarrow D_{\mathit{Threads}}(\neg \mathit{seqCons}))$ is valid. In contrast, for linearizability (*Lin*) the threads cannot be sure whether a trace they validate as linearizable is indeed linearizable, i.e., there exists a trace that satisfies $\mathit{Lin} \wedge \neg D_{\textsc{Threads}\uplus\{obs\}}(\mathit{Lin})$.

The discovery that eventual consistency can be reduced to sequential consistency is facilitated by a generalization of classical sequential consistency that follows naturally from taking the epistemic perspective. Our formalization of *correct* for eventual consistency is given by *correctEVC* that requires nodes to

keep consistent logs, i.e., whenever a transaction is received by a distributed node, the transaction must be inserted into the node's logs in a way that is consistent with the other nodes' recordings. We allow *correctEVC* not only to refer to events that are performed by the nodes that take part in the computation, but also to auxiliary events that model the *environment* that interacts with nodes. We use the environment to model transmission of updates from one distributed node to another. Our knowledge characterization then implicitly quantifies over the order of occurrence of such events, which serves as a correctness certificate for a given trace.

Contributions In summary, our paper makes the following contributions. We provide characterizations for sequential consistency (Section 4), eventual consistency (Section 5) and linearizability (Section 6) which we prove correct wrt. their standard definitions. Our characterizations reveal a remarkable similarity between consistency properties that is not apparent in their standard formulations. Through our characterizations, we identify a natural generalization of sequential consistency that allows us to reduce eventual consistency, a complex property usually defined by the existence of two partial-ordering relations, to sequential consistency. In contrast to this reduction, we show that linearizability requires a different kind of knowledge than sequential consistency and prove a theorem (Section 7) illustrating the ramifications of this difference.

2 Examples

In this section, before providing technical details, we give an informal overview of our characterizations.

2.1 Sequential Consistency

Trace-Based Definition The most fundamental consistency condition that concurrent computations are intuitively expected to satisfy is sequential consistency [16]. Its original definition reads:

> The result of any execution is the same as if the operations of all the processors were executed in *some sequential order*, and the operations of each individual processor appear in the sequence in the *order specified by its program*.

Equivalently, this more formal version can be found in the literature (cf., [1]): For a trace E to be sequentially consistent, it needs to satisfy two conditions: (1) E must be *equivalent* to a witness trace E' and (2) trace E' needs to be *correct* with respect to some specification. To be equivalent, two traces need to be permutations that preserve the local order of events for each thread.

Example 1. Consider the following traces representing threads t_1 and t_2 storing and loading values on a shared register. For the purpose of this example, we

assume the register to be initialized with value 0. We use ":=" to abbreviate "equals by definitions".

$$E_1 := (t_2, ld(0))\ (t_2, ld(1))\ (t_1, st(1))$$
$$E_2 := (t_2, ld(0))\ (t_1, st(1))\ (t_2, ld(1))$$
$$E_3 := (t_2, ld(0))\ (t_1, st(1))\ (t_2, ld(2))\ .$$

Trace E_1 is sequentially consistent, because it is equivalent to E_2 and E_2 meets the specification of a shared register, i.e., each load returns the last value stored. In contrast, E_3 is not sequentially consistent, because no appropriate witness can be found. In no equivalent trace, t_2's load of 2 is preceded by an appropriate store operation.

Logic In this paper, in contrast to the above trace-based formulation, we investigate consistency from the perspective of *epistemic logic*. Epistemic logic is a formalism used for reasoning about the knowledge distributed nodes/threads acquire in a distributed computation. For example, in trace E_1 thread t_2 *knows* it first loaded value 0 and then value 1 while t_1 *knows* it stored 1. When we consider the knowledge acquired by the threads t_1 and t_2 together as a group, we say that the group of threads $\{t_1, t_2\}$ *jointly knows* t_2 first loaded 0 and then 1 while t_1 stored 1. We denote the fact that a group G jointly knows that a formula φ holds by $D_G(\varphi)$, which is an application of the distributed knowledge modality. According to our logical characterization of sequential consistency: $\neg D_{\text{THREADS}}(\neg \mathit{correct})$, a trace is sequentially consistent, if the group of all threads accessing the shared data-structure does not jointly know that the trace is not correct.

Example 1 (continued). This means trace E_1 is sequentially consistent. In trace E_1, the threads know that t_2 first loaded 0 and then 1 and that t_1 stored 1, however they do not know in which order these events were scheduled. This means, for all they know t_1 could have stored 1 before t_2 loaded it and after t_2 loaded 1, which would meet the specification. In contrast, E_3 is not sequentially consistent. The threads know that t_2 loaded 2, however they also know that no thread stored this value. This means E_3 cannot have met the specification.

Indistinguishability We formalize this notion of knowledge in terms of the local perspective individual threads have on the computation. We extract this perspective by a function $\downarrow$ such that $E \downarrow t$ projects trace E onto the local events of thread t. If two traces do not differ from the local perspective of thread t, we say that they are indistinguishable for t. We write $E \sim_t E'$ to denote that for thread t, trace E is indistinguishable from trace E'. Combining their abilities to distinguish traces, a group of threads can distinguish two traces whenever there is a thread in the group that can. We write $E \sim_G E'$ to denote that for for every member of group G trace E is indistinguishable from trace E'. Indistinguishability allows us to define the knowledge of a group. A group G knows a fact φ if this fact holds on all traces that the threads in G cannot distinguish from the

actual trace. We write $E \vDash \varphi$ to say that trace E satisfies formula φ. Formally (see Section 3.3): $E \vDash D_G(\varphi)$:iff for all E' s.t. $E \sim_G E'$: $E' \vDash \varphi$, where we use ":iff" to abbreviate "equals by definition".

Example 1 (continued). For trace E_1, the thread-local projections are: $E_1 \downarrow t_1 = (t_1, st(1))$ and $E_1 \downarrow t_2 = (t_2, ld(0))(t_2, ld(1))$. We get the same projections for E_2, and $E_3 \downarrow t_1 = (t_1, st(1))$ and $E_3 \downarrow t_2 = (t_2, ld(0))(t_2, ld(2))$. From these projections, we get: $E_1 \sim_{t_1} E_2 \sim_{t_1} E_3$ and $E_1 \sim_{t_2} E_2$ but $E_1 \not\sim_{t_2} E_3$ and $E_2 \not\sim_{t_2} E_3$. For groups of threads, we have: $E_1 \sim_{\{t_1,t_2\}} E_2$ but $E_2 \not\sim_{\{t_1,t_2\}} E_3$, because $E_2 \not\sim_{t_2} E_3$. We write $E \vDash \mathit{correctREG}$ to say E is correct with respect to the specification of a shared register. Then $E_1 \vDash \neg D_{\text{THREADS}}(\neg \mathit{correctREG})$, $E_2 \vDash \neg D_{\text{THREADS}}(\neg \mathit{correctREG})$ and $E_3 \vDash D_{\text{THREADS}}(\neg \mathit{correctREG})$.

Knowledge in the Trace-Based Formulation Interestingly, the notion of equivalence found in the trace-based formulation of sequential consistency precisely corresponds to $\sim_{\text{THREADS}}$, the indistinguishability relation of all threads accessing the shared data-structure. This suggests that the knowledge-based formulation of consistency lies already buried in the original definition. Similarly, the formulation "The result of any execution is the same as if ...", found in the original definition alludes to the possibility of a fact φ, which, in epistemic logic, is represented by the dual modality of knowledge $\neg D_G(\neg\varphi)$.

2.2 Eventual Consistency

Eventual consistency [19] is a correctness condition for distributed database systems, as those employed in modern geo-replicated internet services. In such systems, threads (distributed nodes) keep local working-copies (repositories) of the database which they may update by performing a commit operation. Queries and updates have revision ids, representing the current state of the local copy. Whenever a thread commits, it broadcasts local changes to its repository and receives changes made by other threads. After the commit, a new revision id is assigned. As the underlying network is unreliable, committed changes may however be delayed or lost before reaching other threads.

In this setting, weaker guarantees on consistency than in a multi-processor environment are required, as network partitions are unavoidable, causing updates to be delayed or lost. Consequently, eventual consistency is a prototypical example for what is called "weak"-consistency. We present a recent, partial-order based definition drawn from the literature [5] in Section 5.

Taking the knowledge perspective on eventual consistency reveals a remarkable insight. Eventual consistency is actually not an entirely new, weaker consistency condition, but sequential consistency – with an appropriate sequential specification.

In our logical characterization, eventual consistency is defined by the formula: $E \vDash \neg D_{\text{THREADS}}(\neg \mathit{correctEVC})$. That is, to be eventually consistent, a trace needs to be sequentially consistent with respect to a sequential specification $\mathit{correctEVC}$. Our formula for $\mathit{correctEVC}$ uses the past time modality

$\boxminus(\varphi)$ (see Section 3.3), representing the fact that so far, formula φ was true. We specify *correctEVC* by:

$$\begin{aligned} \mathit{correctEVC} \coloneqq\ & \forall t \forall q \forall r (\boxminus(\mathit{query}(t,q,r) \rightarrow \exists \mathcal{L}(\mathcal{L}\ \mathit{validLog}\ t\ \wedge \mathit{result}(q,\mathcal{L},r)))) \\ & \wedge \mathit{atomicTrans} \wedge \mathit{alive} \wedge \mathit{fwd}\ . \end{aligned}$$

This formula says that for all threads, queries and results, so far, whenever a thread t posed a query q to its local repository, producing result r, thread t must be able to present a valid log $\mathcal{L}$, such that the result of posing query q on a machine that performed only the operations logged in log $\mathcal{L}$ matches the recorded result r. The additional conjuncts *atomicTrans*, *alive* and *fwd* specify further requirements on the way updates may be propagated in the network.

In our characterization, a log $\mathcal{L}$ is a sequence of actions (i.e., queries, updates and commits). The formula *validLog* describes the conditions a log has to satisfy to be valid for a thread t:

$$\mathcal{L}\ \mathit{validLog}\ t\ \coloneqq\ \forall a(a\ \mathit{in}\ \mathcal{L} \leftrightarrow t\ k_{\mathit{log}}\ a)\ \wedge \mathit{consistent}(\mathcal{L})\ .$$

This formula requires that for all actions a, a is logged in $\mathcal{L}$ (represented by the infix-predicate *in*) if and only if thread t knows about action a. A thread knows about all the actions that it performed itself and the actions performed in revisions that were forwarded to it. The formula $\mathit{consistent}(\mathcal{L})$ ensures that all actions in the log $\mathcal{L}$ appear in an order consistent with the actual order of events.

Environment Events To make this result possible, we make a generalization that comes naturally in the knowledge setting. We allow traces to contain *environment* events that represent actions that are not controlled by the threads that participate in the computation. In our characterization, environment events are used to mark positions where updates were successfully forwarded from one client to another. By allowing *correctEVC* to refer to those events, we implicitly encode an existential quantification over all possible positions for these events. That means a trace is eventually consistent if any number of such events could have occurred such that the specification *correctEVC* is met.

Example 2. Consider the following traces of a simple database that allows clients to update and query the integer variable x:

$$\begin{aligned} E_4 \coloneqq\ & (t_1, \mathit{up}(0, x \coloneqq 0))\ (t_1, \mathit{com}(0))\ (t_1, \mathit{up}(1, x \coloneqq 1))(t_1, \mathit{com}(1)) \\ & (t_2, \mathit{qu}(0, x, 0))(t_2, \mathit{com}(0))\ (t_2, \mathit{qu}(1, x, 1)) \\ E_5 \coloneqq\ & (t_1, \mathit{up}(0, x \coloneqq 0))\ (t_1, \mathit{com}(0))\ (\mathit{env}, \mathit{fwd}(t_1, t_2, 0))(t_1, \mathit{up}(1, x \coloneqq 1)) \\ & (t_1, \mathit{com}(1))\ (t_2, \mathit{qu}(0, x, 0))(t_2, \mathit{com}(0))\ (\mathit{env}, \mathit{fwd}(t_1, t_2, 1)) \\ & (t_2, \mathit{qu}(1, x, 1))\ . \end{aligned}$$

Updates are of the form $\mathit{up}(\mathit{id}, u)$, where id is the revision-id and u the actual update. In our example, updates are variable assignments $x \coloneqq v$ meaning that

a variable x is assigned value v. Queries are of the form $qu(id, q, r)$, where id stands for the revision-id, q for the query, and r for the result. Queries in our example consist only of variables, i.e., a query returns the current value assigned. The action $com(id)$ represents the act of committing, that is, sending revision id over the network and checking for updates. Forwarding actions are performed by the environment env. The event $(env, fwd(t, t', id))$ represents the environment forwarding the changes made in revision id from thread t to thread t'.

In trace E_5, when thread t_2 queries the value of x in revision 0, thread t_2 can present the log $\mathcal{L} \coloneqq up(0, x \coloneqq 0)\ com(0)\ qu(0, x, 0)$ as an evidence of the correctness of the result 0. As by the time of t's query, only revision 0 has been forwarded from t_1 to t_2, thread t_2 only knows about t_1's first update and its own query. Querying x after the update $x \coloneqq 0$ yields 0, so $result(x, \mathcal{L}, 0)$ holds.

When thread t_2 queries x in revision 1, thread t_1's second update has been forwarded, so t_2 can present the log $\mathcal{L} \coloneqq up(0, x \coloneqq 0)\ com(0)\ up(1, x \coloneqq 1)\ com(1)\ qu(0, x, 0)\ com(0)\ qu(1, x, 1)$. Since t_2 received the t_1's revision 1 the log contains the second update $x \coloneqq 1$ and t_2's query of x returns 1. This means $E_5 \vDash correctEVC$. As a consequence, we have $E_4 \vDash \neg D_{\text{THREADS}}(\neg correctEVC)$, because $E_4 \sim_{\text{THREADS}} E_5$ and $E_5 \vDash correctEVC$. The forwarding events in E_5 mark positions where the transmission of updates through the network could have occurred to make the computation meet $correctEVC$.

2.3 Linearizability

While the threads' knowledge characterizes sequential consistency and eventual consistency, their knowledge is not strong enough to define linearizability. Linearizability extends sequential consistency by the requirement that method calls must effect all visible change of the shared data at some point between their invocation and their return. Such a point is called the *linearization points* of the method.

To characterize linearizability, we introduce another agent called *the observer* that tracks the available information on linearization points. To do this, the observer monitors the order of non-overlapping (sequential) method calls in a trace. The observer's view of a trace is the order of non-overlapping method calls. This order is represented by a set of pairs of return and invoke events, such that the return took place before the invocation. We extract this order by a projection function $obs(\cdot)$.

Example 3. Consider the following traces where method calls are split into invocation- and return events:

$$E_6 \coloneqq (t_2, inv\ ld())\ (t_2, ret\ ld(1))\ (t_1, inv\ st(1))\ (t_1, ret\ st(true))$$
$$E_7 \coloneqq (t_2, inv\ ld())\ (t_1, inv\ st(1))\ (t_2, ret\ ld(1))\ (t_1, ret\ st(true))$$
$$E_8 \coloneqq (t_1, inv\ st(1))\ (t_1, ret\ st(true))\ (t_2, inv\ ld())\ (t_2, ret\ ld(1))\ .$$

For trace E_6, the observer's projection function $obs(\cdot)$ yields: $obs(E_6) = \{(\ (t_2, ret\ ld(1)), (t_1, inv\ st(1))\ )\}$. This means the observer sees that t_2's load

returned before t_1's store was invoked. In trace E_7, the method calls overlap. Consequently, the observer knows nothing about this trace: $obs(E_7) := \varnothing$. For E_8, we get $obs(E_8) = \{(\ (t_1, ret\ st(true)), (t_2, inv\ ld())\ )\}$.

The observer's view tracks the available information on linearization points. In trace E_6, thread t_2's linearization point for the call to load must have occurred before the linearization point of t_1's call to store. This follows from the fact that $t2$'s load returned before $t1$'s call to store and that linearization point must occur somewhere between a method's invocation and its return. In trace E_7 linearization points may have occurred in any order as the method calls overlap.

To the observer, a trace E is indistinguishable from a trace E' if the order of linearization points in E is preserved in E' and maybe an order between additional linearization points is fixed (see Section 3.1): $E \preceq_{obs} E'$:iff $obs(E) \subseteq obs(E')$. A trace E is linearizable if the threads together with the observer do not know that the trace is incorrect: $E \vDash \neg D_{\text{THREADS} \uplus \{obs\}}(\neg correct)$.

Example 3 (Continued). We have $E_6 \not\preceq_{obs} E_7$, but $E_7 \preceq_{obs} E_6$. Trace E_7 is linearizable since $E_7 \sim_{\text{THREADS} \uplus \{obs\}} E_8$ and $E_8 \vDash correctREG$. However, trace E_6 is not linearizable since there is no indistinguishable trace that meets the specification. Note that the threads without the observer could not have detected this violation of the specification, i.e., $E_6 \vDash \neg D_{\text{THREADS}} \neg correctREG$.

2.4 Knowledge about Consistency

As we describe sequential consistency in a standard logic of knowledge, corresponding axioms apply (see, e.g. [21, chapter 2.2]). For example, everything a group of threads knows is also true: (T) $:= \vDash D_G(\varphi) \rightarrow \varphi$ (Truth axiom), groups of threads know what they know: (4) $:= \vDash D_G(\varphi) \rightarrow D_G(D_G(\varphi))$ (positive introspection) and groups of threads know what they do not know: (5) $:= \vDash \neg D_G(\varphi) \rightarrow D_G(\neg D_G(\varphi))$ (negative introspection). For a complete axiomatization of a similar epistemic logic with time see [3].

Interestingly, adding the observer not only strengthens the threads' ability to distinguish traces but changes the *kind* of knowledge agents acquire about a computation. Whereas $\sim_{\text{THREADS}}$ is an *equivalence relation*, $\sim_{\text{THREADS} \uplus \{obs\}}$ is only a *partial order*. As a consequence, D_{THREADS} corresponds to the modal system *S5*, whereas $D_{\text{THREADS} \uplus \{obs\}}$ corresponds to the weaker system *S4* [21]. This means, that $D_{\text{THREADS} \uplus \{obs\}}$ does not satisfy the axiom of negative introspection (5).

It seems natural to ask if the differences in the type of knowledge between sequential consistency and linearizability affect the ability to detect violations of the specification. In Section 7, we show that the difference the lack of axiom (5) makes, lies in the certainty threads have about their decision. Whereas for sequentially consistent ($seqCons := \neg D_{\text{THREADS}}(\neg correct)$), whenever the threads decide that a trace is sequentially consistent, they can be sure that the trace is indeed sequentially consistent: $(seqCons \leftrightarrow D_{\text{THREADS}}(seqCons))$ for linearizability ($Lin := \neg D_{\text{THREADS} \uplus \{obs\}}(\neg correct)$), it can occur that the threads together with the observer decide that a trace is linearizable, however, they cannot be sure that it really was: $Lin \wedge \neg D_{\text{THREADS} \uplus \{obs\}}(Lin)$.

3 Logic Of Knowledge

In this section we present a standard logic of knowledge (see [9]) that we use for our characterizations. We follow the exposition of [15]. We define the set $\mathcal{E}$ of events as $\mathcal{E} \ni e := (t, act)$, representing $t \in \textsc{Threads} \uplus \{env\}$ performing an action $act \in \mathcal{A}$. The environment env can perform synchronization events that go unseen by the threads. In our characterization of eventual consistency, the environment forwards transactions from one node to the other. We define the generic set of actions: $\mathcal{A} \ni act := \ inv(m, v) \mid ret(m, v)$. Threads can invoke or return from methods $m \in \textsc{Methods}$ with $v \in \textsc{Values}$. For our characterization of eventual consistency, we instantiate $\mathcal{A}$ with application-specific actions. These can easily be translated back into the generic form by splitting up events into separate invocation- and return-parts.

3.1 Preliminaries

We denote by $\mathcal{E}^*$ the set of finite-, and by $\mathcal{E}^\infty$ the set of infinite sequences over $\mathcal{E}$. We denote the empty sequence by ϵ. Let $\mathcal{E}^\omega := \mathcal{E}^* \uplus \mathcal{E}^\infty$ and $E \in \mathcal{E}^\omega$. Then $E \upharpoonright i$ denotes the finite prefix up to- and including i. We let $E@i$ be the element of sequence E at position i. We define $len(E)$ to be the length of E, where $len(\epsilon) = 0$, and $len(E) = \omega$, if $E \in E^\infty$. For $e \in \mathcal{E}$, we say that $pos(e, E) = j$, if $E@j = e$ and $pos(e, E) = \omega$ otherwise. Hence, we write $e \in E$ if $pos(e, E) < \omega$. We make the assumption that each event occurs only once in a trace. This is not a restriction as we could add a unique time-stamp or a sequence number to each event.

We formally define projection functions and indistinguishability relations. A thread's view of a computation trace is the part of the trace it can observe. We define this part by a projection function that extracts the respective events. We use this projection function to define an indistinguishability relation for each thread.

Thread Indistinguishability Relation For a thread $t \in \textsc{Threads}$ the indistinguishability relation $\sim_t \subseteq (\mathcal{E}^\omega \times (\mathbb{N} \uplus \{\omega\}))^2$ is defined such that: $(E, i) \sim_t (E', i')$:iff $(E \upharpoonright i) \downarrow t = (E' \upharpoonright i') \downarrow t$ where $\downarrow$: $(\mathcal{E}^\omega \times \textsc{Threads}) \to \mathcal{E}^\omega$ designates a projection function onto t's local perspective. $E \downarrow t$ is the projection on events in the set $\{(t, act) \mid act \in \mathcal{A}\}$, i.e., the sequence obtained from E by erasing all events that are not in the above set.

Observer Indistinguishability Relations The observer's view of a trace is the order of non-overlapping method calls. We let $\textsc{Inv} \ni in := (t, inv(m, v))$ and $\textsc{Ret} \ni r := (t, ret(m, v))$. The indistinguishability relation of the observer $\preceq_{\text{obs}} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$ is given by: for all $(E, i), (E', i') \in \mathcal{E}^\omega \times \mathbb{N}$: $(E, i) \preceq_{\text{obs}} (E', i')$:iff $\text{obs}(E, i) \subseteq \text{obs}(E', i')$ where obs: $(\mathcal{E}^\omega \times \mathbb{N}) \to \mathcal{P}(\mathcal{E}^2)$ designates a projection onto the observer's local view, such that: $obs(E, i) = \{(r, in) \in \textsc{Ret} \times \textsc{Inv} \mid \text{pos}(r, E) < \text{pos}(in, E) \leq i\}$. We abbreviate $obs(E) := obs(E, len(E))$.

Joint Indistinguishability Relations Joint indistinguishability relations link pairs of traces that a group of threads can distinguish if they share their knowledge. Whenever a thread in the group can tell the difference between two traces, the group can. Let $G \subseteq \textsc{Threads}$. We define the joint indistinguishability relation of group G to be $\sim_G := (\bigcap_{t \in G} \sim_t)$ and $\sim_{G \uplus \{obs\}} := \sim_G \cap \preceq_{obs}$. For any indistinguishability relation $\sim$, we write $E \sim E'$ as an abbreviation for $(E, len(E)) \sim (E, len(E'))$.

3.2 Syntax

A formula in the logic takes the form:

$$\varphi, \psi ::= \varphi \wedge \psi \mid \neg\varphi \mid \varphi S \psi \mid \varphi U \psi \mid D_G(\varphi) \mid \forall x(\varphi) \ .$$

with $G \subseteq \textsc{Threads} \uplus \{obs\}$ and $p \in \textsc{Predicates}$, which we instantiate for each of our characterizations. The logic provides the temporal modalities $\varphi S \psi$ representing the fact that since ψ occurred, φ holds and the modality $\varphi U \psi$ representing the fact that until ψ occurrs, φ holds. Additionally, it provides the *distributed knowledge* modality D_G and first order quantification. Let Φ denote the set of all formulae in the logic.

3.3 Semantics

We now define the satisfaction relation $\vDash \subseteq (\mathcal{E}^\omega \times (\mathbb{N} \uplus \omega)) \times \Phi$. We let:

$$\begin{array}{ll} (E,i) \vDash \varphi \wedge \psi & :\text{iff } (E,i) \vDash \varphi \text{ and } (E,i) \vDash \psi \\ (E,i) \vDash \neg\varphi & :\text{iff not } (E,i) \vDash \varphi \ . \end{array}$$

We define the temporal modalities by:

$$\begin{array}{lll} (E,i) \vDash \varphi S \psi & :\text{iff} & \text{there is } j \leq i \text{ s.t. } (E,j) \vDash \psi \text{ and} \\ & & \text{for all } j < k \leq i : (E,k) \vDash \varphi \\ (E,i) \vDash \varphi U \psi & :\text{iff} & \text{there is } j \leq i \text{ s.t. } (E,j) \vDash \psi \text{ and} \\ & & \text{for all } 1 \leq k < j : (E,k) \vDash \varphi \ . \end{array}$$

We define distributed knowledge as: $(E,i) \vDash D_G(\varphi)$:iff for all (E',i'): if $(E,i) \sim_G (E',i')$ then $(E',i') \vDash \varphi$, with $G \subseteq \textsc{Threads} \uplus \{obs\}$. Let D be the domain of quantification. We define first-order quantification: $(E,i) \vDash \forall x(\varphi)$:iff for all $d \in \mathrm{D} : (E,i) \vDash \varphi[d/x]$. By $\varphi[d/x]$, we denote the term φ with all occurrences of x replaced by d. We define D as the disjoint union of all quantities used in the definition of a condition. We write $E \vDash \varphi$ as an abbreviation for $(E, len(E)) \vDash \varphi$.

Additional Definitions For convenience, we define the following standard operators in terms of our above definitions: $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi := \neg\varphi \vee \psi$, $\top := (p \vee \neg p)$ for some atomic predicate p, $\Diamond\!\!\!-\varphi := \top S \varphi$ ("once φ"), $\boxminus\varphi := \neg \Diamond\!\!\!- \neg\varphi$ ("so far φ"), $\Diamond\varphi := \top U \varphi$ ("eventually φ"), $\Box\varphi := \neg \Diamond \neg\varphi$ ("always φ"), $\varphi W \psi := \varphi U \psi \vee \Box\varphi$ ("weak until"), $\exists x(\varphi) := \neg\forall x(\neg\varphi)$.

4 Sequential Consistency

We present a trace-based definition of sequential consistency (cf., [1]) and prove our logical characterization equivalent. Our definition of sequential consistency generalizes the original definition [16] by allowing non-sequential specifications.

Definition 1 (Sequential Consistency). *Let* $\textsc{Spec} \subseteq \mathcal{E}^*$ *be a specification of the shared data-structure. A trace* $E \restriction i$ *is sequentially consistent* $seqCons(E, i)$ *if and only if there is* $(E', i') \in \mathcal{E}^\omega \times \mathbb{N}$ *s.t. for all* $t \in \textsc{Threads}$*:*

$$(E \restriction i) \downarrow t = (E' \restriction i') \downarrow t \text{ and } E' \restriction i' \in \textsc{Spec} .$$

Basic Predicates For our logical characterization, we define the predicate *correct* representing the fact that a trace meets the specification:

$$(E, i) \vDash \mathit{correct} \text{ :iff } E \restriction i \in \textsc{Spec} .$$

Theorem 1 (Logical Characterization of Sequential Consistency). *A trace* $E \restriction i$ *is sequentially consistent if and only if the threads do not jointly know that it is incorrect:* $seqCons(E, i)$ *iff* $(E, i) \vDash \neg D_{\textsc{Threads}}(\neg \mathit{correct})$.

5 Eventual Consistency

We define the set of actions for eventual consistency as:

$$\mathcal{A} \ni act := \mathit{qu}(id, q, r) \mid \mathit{up}(id, u) \mid \mathit{com}(id) \mid \mathit{fwd}(t, t', id) .$$

Threads may pose a query (*qu*) $q \in \textsc{Queries}$ with result $r \in \textsc{Values}$, issue an update (*up*) $u \in \textsc{Updates}$, or commit (*com*) their local changes. Queries, updates and commits get assigned a revision-id $id \in \textsc{Identifiers}$, representing the current state of the local database copy. We assume that if a thread commits, the committed revision id matches the revision id of the previous queries and updates i.e., those performed since the last commit, and that thread-revision-id pairs (t, id) are unique. Again, this is no restriction. To fulfil the requirement, the threads can just increment their local revision id whenever they commit. As updates may get lost in the network, we represent by $\mathit{fwd}(t, t', id)$ the successful forwarding of the updates made by thread t in revision id to thread t'.

Preliminaries We let $set(E) = \{e \mid e \in E\}$, i.e., the set of events in trace E. On a fixed trace E, we define the program order $\prec_p$ as $e \prec_p e'$:iff if there is t such that $pos(e, E \downarrow t) < pos(e', E \downarrow t)$. We let "$\_$" represent irrelevant, existential quantification. Let $e \equiv_t e'$ if and only if there is $id \in \textsc{Identifiers}$ such that $e = (t, \_(id, \_))$ and $e' = (t, \_(id, \_))$, i.e., if the events belong to the same revision of thread t. A relation $\preceq$ factors over $\equiv_t$ if $x \preceq y$, $x \equiv_t x'$ and $y \equiv_t y'$ imply $x' \preceq y'$. Updates are interpreted in terms of a set of states $\textsc{States}$, i.e., we assume there is an interpretation function $u^\# : \textsc{States} \to \textsc{States}$, for each $u \in \textsc{Updates}$, and a designated initial state $s_0 \in \textsc{States}$. For each query $q \in \textsc{Queries}$, there is an interpretation function $q^\# : \textsc{States} \to \textsc{Values}$. For a finite set of events E_S, a total order $\prec$ over the events in E_S, and a state s we let $\mathit{apply}(E_s, \prec, s)$ be the result of applying all updates in E_s to s, in the order specified by $\prec$.

Definition 2 (Eventual Consistency). *We use the definition presented in [5]. A trace $E \in \mathcal{E}^\omega$ is eventually consistent ($evCons(E)$) if and only if there exist a partial order $\prec_v$ (visibility order), and a total order $\prec_a$ (arbitration order) on the events in $set(E)$ such that:*

- $\prec_v \subseteq \prec_a$ *(arbitration extends visibility).*
- $\prec_p \subseteq \prec_v$ *(visibility is compatible with program-order).*
- *for each $e_q = (t, qu(id, q, r)) \in E$, we have $r = q^\#(apply(\{e \mid e \prec_v e_q\}, \prec_a, s_0))$ (consistent query results).*
- $\prec_a$ *and* $\prec_v$ *factor over* $\equiv_t$ *(atomic revisions).*
- *if $(t, com(id)) \notin E$ and $(t, \_(id, \_)) \prec_v (t', \_)$ then $t = t'$ (uncommitted updates).*
- *if $e = (t, com(id)) \in E$ then there are only finitely many $e' := (t', com(id'))$ such that $e' \in E$ and $e \not\prec_v e'$ (eventual visibility).*

5.1 Logical Characterization

Basic Predicates We represent queries and updates by predicates $query(t, q, r, id)$ and $update(t, u, id)$, representing $t \in$ Threads, issuing $q \in$ Queries with result $r \in$ Values on revision $id \in$ Identifiers, and t performing $u \in$ Updates on revision id, respectively. As threads work on their local copies, revision ids mark the version of data the threads work with. We represent commits by the predicate $commit(t, id)$, representing t committing its state in revision id. After performing a commit, a new revision id is assigned. We define:

$$\begin{aligned}
(E, i) &\vDash query(t, q, r, id) &&:\text{iff } E@i = (t, qu(id, q, r)) \\
(E, i) &\vDash update(t, u, id) &&:\text{iff } E@i = (t, up(id, u)) \\
(E, i) &\vDash commit(t, id) &&:\text{iff } E@i = (t, com(id)) \ .
\end{aligned}$$

We let $query(t, q, r) := \exists id(query(t, q, r, id))$. Upon commit, a thread forwards all the information from its local repository to the database system and receives updates from other threads. Committed updates may however be delayed or lost by the network. By the predicate $forward(t, t', id)$ we mark the event that the environment forwarded the updates t performed in revision id to t'. We let: $(E, i) \vDash forward(t, t', id)$:iff $E@i = (env, fwd(t, t', id))$. Eventual consistency requires all threads to keep valid logs. Logs are finite sequences of actions, i.e., $\mathcal{L} \in A^*$. We let: $(E, i) \vDash a \text{ in } \mathcal{L}$:iff $a \in L$. By $t \ k_{log} \ a$ we denote the fact that t knows about action a. The predicate k_{log} represents individual knowledge, i.e., knowledge in the sense of knowing about an action in contrast to knowing that a fact is true [15]. We let:

$$\begin{aligned}
(E, i) \vDash t \ k_{log} \ a \ :\text{iff there is } j \le i : (E@j = (t, a) \text{ or} \\
((E, j) \vDash forward(t', t, id) \text{ and there is } l < j : \ (E, l) \vDash commit(t', id) \\
\text{and } (E, l) \vDash t' \ k_{log} \ a)) \ .
\end{aligned}$$

That is, threads know an action if they performed it themselves, or they received an update containing it. Upon commits, threads pass on all actions they know about. We represent log validity by the formula: $\mathcal{L}\ validLog\ t\ :=\ \forall a(t\ k_{log}\ a \leftrightarrow a\ in\ \mathcal{L}) \wedge consistent(\mathcal{L})$. That is, to be a valid log for thread t, log $\mathcal{L}$ must contain exactly the actions that t knows of and these actions must be arranged in an order consistent with respect to the other threads logs. A log $\mathcal{L}$ is consistent if the actions in the log occur in the same order as the actions in the real trace. This means the sequence of actions in $\mathcal{L}$ must be a subsequence of the actions in the real trace. A sequence $a = a_1 a_2 \ldots a_n$ is a subsequence of a sequence $b = b_1 b_2 \ldots b_m$ ($a \preceq b$), if and only if there exist $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that for all $1 \leq j \leq n : a_j = b_{i_j}$. We project a sequence of events to a sequence of actions by the function $act : \mathcal{E}^* \to \mathcal{A}^*$, such that $act((t_1, a_1)(t_2, a_2) \ldots (t_n, a_n)) = a_1 a_2 \ldots a_n$. We define: $(E, i) \vDash consistent(\mathcal{L})$:iff $\mathcal{L} \preceq act(E \downharpoonright i)$.

Query Results All queries that threads issue must return the correct result with respect to the logged operations. That is, the query's result must match the result the query would yield when issued on a database that performed all the updates in the log. We represent the fact that query q would yield result r on log $\mathcal{L}$ by the predicate result$(q, \mathcal{L}, r)$. We define the order of actions in a log $\mathcal{L}$ by the relation $<_{\mathcal{L}}$. We let $a <_{\mathcal{L}} a'$:iff $pos(a, \mathcal{L}) < pos(a', \mathcal{L}) < \omega$. We define: $(E, i) \vDash result(q, \mathcal{L}, r)$:iff $r = q^{\#}(apply(set(\mathcal{L}), <_{\mathcal{L}}, s_0))$.

Network Assumptions We pose additional requirements on the network: updates in the same revision must be sent as atomic bundles (*atomicTrans*). Only committed updates can be forwarded (*fwd*). Active threads must eventually receive all committed update (*alive*). We define the helper predicate: $rev(t, id) := \exists q \exists r(query(t, q, r, id)) \vee \exists u(update(t, u, id))$ representing the fact, that the current action belongs to revision id of thread t. We specify the requirements that updates made in the same revision must be sent bundled as indivisible transactions by the formula : $atomicTrans\ :=\ \forall t \forall id(\boxminus(rev(t, id) \rightarrow rev(t, id)\ W\ commit(t, id)))$. That is, queries and updates from revision id are only followed by other queries and updates from the same revision, or a commit. We enforce that only committed revisions can be forwarded by: $fwd\ :=\ \forall t \forall t' \forall id(\boxminus(fwd(t, t', id) \rightarrow \Box(\neg commit(t, id))))$. Threads that makes progress, i.e. that commit infinitely often must eventually receive all committed updates. We formalize this as:

$$
\begin{aligned}
alive := \ & \forall t \forall t' \forall id \\
& (\boxminus(commit(t, id) \wedge \Box \Diamond (\exists id'(commit(t', id'))) \rightarrow \\
& \Diamond forward(t, t', id))) \ .
\end{aligned}
$$

We represent *correctEVC* by the formula:

$$
\begin{aligned}
correctEVC := \ & \forall t \forall q \forall r \\
& (\boxminus(query(t, q, r) \rightarrow \exists \mathcal{L}(\mathcal{L}\ validLog\ t\ \wedge result(q, \mathcal{L}, r)))) \\
& \wedge atomicTrans \wedge alive \wedge fwd \ .
\end{aligned}
$$

Theorem 2 (Logical Characterization of Eventual Consistency). *A trace is eventually consistent if and only if the threads do not know that it violates correctEVC. For all traces* $E \in \mathcal{E}^\omega$*:*

$$evCons(E) \text{ if and only if } E \vDash \neg D_{\text{THREADS}} \neg (correctEVC) \ .$$

6 Linearizability

Linearizability refines sequential consistency by guaranteeing that each method call takes its effect at exactly one point between its invocation and its return.

For our definition of linearizability, we follow [8]. As for sequential consistency, our definition generalizes the original notion [10, 12] by allowing non-sequential specifications. We define the real-time precedence order $\preceq_{real} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$: $(E,i) \preceq_{real} (E',i')$:iff $i = i'$ and there is a bijection $\pi : \{1,\dots,i\} \to \{1,\dots,i\}$ s.t for all $j \in \mathbb{N}$ such that $j \leq i : E@j = E'@\pi(j)$, i.e., E' is a permutation of E, and for all $j,k \in \mathbb{N}$ such that $j < k \leq i :$ if $E@j \in$ RET and $E@k \in$ INV then $\pi(j) < \pi(k)$, i.e., when permuting the events in E, calls are never pulled before returns.

Definition 3 (Linearizability). *A trace* (E,i) *is linearizable (*$lin(E,i)$*) if and only if there is* $(E',i') \in \mathcal{E}^\omega \times \mathbb{N}$ *such that (1) for all* $t \in$ THREADS*:* $(E \restriction i) \downarrow t = (E' \restriction i') \downarrow t$ *(2)* $(E,i) \preceq_{real} (E',i')$ *and (3)* $E' \restriction i' \in$ SPEC*.*

Theorem 3 (Logical Characterization of Linearizability). *A trace* $E \restriction i \in \mathcal{E}^*$ *is linearizable if and only if the threads together with the observer do not know that it is incorrect:*

$$lin(E,i) \text{ iff } (E,i) \vDash \neg D_{\text{THREADS} \uplus \{obs\}} \neg correct \ .$$

7 Knowledge about consistency

We write $\vDash \varphi$ as an abbreviation for: for all $E \in \mathcal{E}^\omega$: $E \vDash \varphi$. Let $seqCons := \neg D_{\text{THREADS}}(\neg correct)$.

Theorem 4 (Detection Sequential Consistency). *Threads can decide whether a trace is sequentially consistent or not:* $\vDash (seqCons \leftrightarrow D_{Threads}(seqCons)) \wedge (\neg seqCons \leftrightarrow D_{Threads}(\neg seqCons))$.

Let $Lin := \neg D_{\text{THREADS} \uplus \{obs\}} \neg correct$.

Theorem 5 (Detection Linearizability). *There is* $E \in \mathcal{E}^\omega$ *such that* $(E,i) \vDash Lin \wedge \neg D_{Threads \uplus \{obs\}}(Lin)$*. As in sequential consistency, the threads together with the observer can spot if a trace is not linearizable:* $\vDash \neg Lin \leftrightarrow D_{Threads \uplus \{obs\}}(\neg Lin)$.

8 Related Work

The only applications of epistemic logic to concurrent computations that we are aware of are a logical characterization of wait-free computations by Hirai [13] and a knowledge based analysis of cache-coherence by Baukus et al. [2].

Acknowledgements We would like to thank Jade Alglave, Alexey Gotsman, Rose Hoberman, Simon Kramer, Corneliu Popeea, Moshe Y. Vardi, and the anonymous reviewers for helpful feedback. This research was supported in part by a Microsoft Research Scholarship and by the ERC project 308125 VeriSynth.

References

1. H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.
2. K. Baukus and R. van der Meyden. A knowledge based analysis of cache coherence. In *ICFEM*, 2004.
3. F. Belardinelli and A. Lomuscio. A complete first-order logic of knowledge and time. In *KR*, 2008.
4. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI*, 2010.
5. S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.
6. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.
7. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 2003.
8. A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*. 2011.
9. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *JACM*, 37(3):549–587, 1990.
10. Herlihy, M. P., and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
11. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
12. M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL*, 1987.
13. Y. Hirai. An intuitionistic epistemic logic for sequential consistency on shared memory. In *LPAR*, 2010.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969.
15. S. Kramer and A. Rybalchenko. A multi-modal framework for achieving accountability in multi-agent systems. In *LIS*, 2010.
16. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
17. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
18. C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4):631–653, 1979.
19. M. Shapiro and B. Kemme. Eventual consistency. In M. T. Özsu and L. Liu, editors, *Encyclopedia of Database Systems*, pages 1071—1072. Springer, 2009.
20. V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.
21. H. van Ditmarsch, W. van der Hoeck, and B. Kooi. *Dynamic Epistemic Logic*. Springer, Dordrecht, 2008.