Multiagent Transition Systems with Safety and Liveness Faults: A Compositional Foundation for Fault-Resilient Distributed Computing

Ehud Shapiro
Weizmann Institute of Science, Rehovot, Israel
ehlerud.shapiro@weizmann.ac.il

Abstract. We present a novel mathematical framework for the specification and analysis of distributed computing systems and their implementations, with the following components:
1. Transition systems that allow the specification and analysis of computations with safety and liveness faults and their fault resilience.
2. Notions of safe, live and complete implementations among transition systems and their composition, with which the correctness (safety and liveness) and completeness of a protocol stack as a whole follows from each protocol implementing correctly and completely the protocol above it in the stack.
3. Applying the notion of monotonicity, pertinent to histories of distributed computing systems, to ease the specification and proof of correctness of implementations among distributed computing systems.
4. Multiagent transition systems, further characterized as centralized/distributed and synchronous/asynchronous; safety and liveness fault-resilience of implementations among them and their composition.
5. An algebraic and operational characterization of a protocol (family of distributed multiagent transition systems) being grassroots, which means that the protocol may be deployed independently at multiple locations and over time, and that such deployments can subsequently interoperate once interconnected; sufficient conditions for a protocol to be grassroots; and the notion of a grassroots implementation among protocols.

The framework is being employed in the specification of a grassroots ordering consensus protocol stack [33], with sovereign cryptocurrencies [34], an NFT trade protocol [34], and an efficient Byzantine atomic broadcast protocol [20] as the first applications.

Keywords: Distributed Computing · Multiagent Transition Systems · Fault Resilience · Protocol Stack

1 Introduction and Related Work

This paper presents a mathematical framework for specifying and proving in a compositional way the correctness and fault-resilience of a distributed protocol
stack. Different aspects of this problem have been addressed for almost half a century.

Process calculi have been proposed for the compositional specification and proof of concurrent systems \cite{17,20,27}, mostly focusing on synchronous communication, although variants for asynchronous distributed computing have been investigated \cite{3,11}, including their resilience to fail-stop failures \cite{12}.

Transition systems are a standard way of specifying computing systems without committing to a specific syntax. The use of transition systems for the specification of concurrent and distributed systems has been investigated extensively \cite{16,1,25}, including the notion of implementations among transition systems and their composition \cite{2,25,18}. The composition of implementations has been investigated in the context of multi-phase compilation \cite{23,29}, where the correctness of the compiler as a whole following from the correctness of each phase in the compilation. Due to the deterministic and centralized nature of compilation, this task did not require addressing questions of liveness, completeness, and fault tolerance. Transition systems have been also employed to specify and prove the fault-resilience of distributed systems \cite{36}.

Fault-resilient distributed computing, especially the problems of Byzantine Agreement \cite{35}, Byzantine Reliable Broadcast \cite{4,14,7}, Byzantine Atomic Broadcast (ordering consensus) \cite{37,19,13}, and blockchain consensus \cite{28}, have been investigated extensively. Methods for reasoning about distributed systems have been developed \cite{21,32,24}, including their fault resilience \cite{36}, and formal frameworks for the specification and proof of distributed systems were developed \cite{25,22,36}. However, the reality is that novel protocols and their proofs, e.g., \cite{6,37,19,13,8}, are typically presented outside any formal framework, probably due to the sheer complexity of the protocols and their proofs.

To the best of our knowledge, a mathematical framework for specifying and proving in a compositional way the correctness and fault-resilience of distributed protocol stack, in which each protocol implements the protocol above it and serves as a specification for the protocol below it, is novel. We developed the framework with the goal of specifying and proving the correctness and fault-resilience of a particular protocol stack: One that commences with with an open dissemination protocol that can support the grassroots formation of a peer-to-peer social network; continues with a protocol for equivocation exclusion that can support sovereign cryptocurrencies and an equivocation-resilient NFT trade protocol \cite{34}; and culminates in a group consensus protocol for ordering transactions despite Byzantine faults, namely Byzantine Atomic Broadcast \cite{20}.

Here, we present, prove correct, and analyze the fault-resilience of two abstract protocol stacks, depicted in Figure 1, as example applications of the mathematical framework. A more concrete, complex, and practical protocol stack based on the blocklace (a partially-ordered generalization of the blockchain) is presented and analyzed elsewhere \cite{33}, using the mathematical framework developed here.

A key objective of this work is the development grassroots protocols that can be deployed independently at different locations and over times, initially with
disjoint communities operating the protocol independently, and over time—once connected—forming an ever-growing interacting networked community. Here we characterize the notion of a protocol being grassroots algebraically and operationally, analyze whether protocols in the abstract protocol stack are grassroots, and discuss whether client-server protocols (e.g., all major digital platforms), consensus protocols (e.g. reliable broadcast, Byzantine agreement), majoritarian decision making protocols (e.g. democratic voting), and protocols that employ a non-composable data structure (e.g., blockchain), are grassroots, and if not, then whether and how can they be made so.

**Fig. 1.** Example Transition Systems-Based Protocol Stacks. A. Generic (Example 1), Single-Chain (Ex. 2), and Longest Chain (Ex. 3) transition systems and their implementations \(\sigma_1\) (Proposition 2) and \(\sigma_2\) (Def. 11 and Prop. 12). B. Generic Shared-Memory (Ex. 4), Single-Chain Consensus (Ex. 5), Longest-Chain Consensus (Ex. 6), and Asynchronous Block Dissemination (Ex. 7) multiagent transition systems and their implementations \(\sigma_{1m}\) (Prop. 7), \(\sigma_{2m}\) (Prop. 8), and \(\sigma_3\) (Prop. 11).

Our approach is different from that of universal composability [5], devised for the analysis of cryptographic protocols, in at least two respects: First, it does not assume, from the outset, a specific notion of communication. Second, its notion of composition is different: Universal composability uses function composition as is common in the practice of protocol design (e.g. [19,7,31]). Here, we do not compose protocols, but compose implementations among protocols, resulting in a new single implementation that realizes the high-level protocol using the primitives of the low-level protocol. For example, it seems that the universality results of Sections 2 and 3 cannot be expressed in the model of universal composability.

Grassroots composition is an instance of parallel composition, a notion that has been explored extensively [17,20,27,22,25]: Grassroots composition takes components that have the capacity to operate independently of each other, and ensures that they can still operate independently even when composed,
but also interact in novel ways after composition. While grassroots composition leaves open the possibility that components may interact once composed, it does not specify how they might interact; such interactions are a consequence of the specifics of the composed transition systems.

In the rest of the paper Section 2 presents transition systems, implementations among them, and the composition of such implementations, and includes the example protocol stack of Figure 1A. It also introduces the notion of monotonicity of transition systems [30,15], and shows that it can ease the proof of correctness of an implementation. Section 3 presents multiagent transition systems, further characterized as centralized or distributed, with the latter being synchronous or asynchronous, and includes the example multiagent protocol stack of Figure 1B. Section 4 introduces safety faults and liveness faults, implementations that are resilient to such faults, and their composition. Section 5 introduces the notion of a family of multiagent transition systems, aka protocol, introduces grassroots protocols, their characterization and implementation. Section 6 concludes the paper. Proof are relegated to Appendix A.

2 Transition Systems, Implementations and their Composition

Here, we introduce the notions of transition systems, implementations among them, and their composition, together with the examples of Figure 1A.

2.1 Transition Systems and Their Implementation

Given a set $S$, $S^*$ denotes the set of sequences over $S$, $S^+$ the set of nonempty sequences over $S$, and $\Lambda$ the empty sequence. Given $x, y \in S^*$, $x \cdot y$ denotes the concatenation of $x$ and $y$, and $x \preceq y$ denotes that $x$ is a prefix of $y$. Two sequences $x, y \in S^*$ are consistent if $x \preceq y$ or $y \preceq x$, inconsistent otherwise.

**Definition 1 (Transition System, Computation, Run).** Given a set $S$, referred to as states, the transitions over $S$ are all pairs $(s, s') \in S^2$, also written $s \rightarrow s'$. A transition system $TS = (S, s_0, T, \lambda)$ consists of a set of states $S$, an initial state $s_0 \in S$, a set of correct transitions $T \subseteq S^2$, and a liveness condition $\lambda$ which is a set of sets correct transitions; when $\lambda$ is omitted the default liveness condition is $\lambda = \{T\}$. A computation of $TS$ is a sequence of transitions $r = s \rightarrow s' \rightarrow \cdots \subseteq S^2$. A run of $TS$ is a computation that starts from $s_0$.

Recall that safety requires that bad things don’t happen, and liveness that good things do happen, eventually. For example, “a transition that is enabled infinitely often is eventually taken”. Heraclitus said that you cannot step into the same river twice. Similarly, in a transition system you cannot take the same transition in different states as, by definition, it is a different transition. Hence, a liveness requirement is on a set of transitions, rather than a single transition. For
example, the set of all transitions in which ‘p receives message m from q’, even if the state of p or of other agents changes. In multiagent transition systems, defined below, liveness may require each agent to act every so often. In such a case we consider all transitions by the same agent for the liveness requirement. To specify the liveness condition, λ considers sets of correct transitions with a liveness requirement placed on each set.

**Definition 2 (Safe, Live and Correct Run).** Given a transition system \( TS = (S, s_0, T, \lambda) \), a computation \( r \) is **safe**, also \( r \subseteq T \), if every transition of \( r \) is correct, and \( s \xrightarrow{} s' \subseteq T \) denotes the existence of a safe computation (empty if \( s = s' \)) from \( s \) to \( s' \).

A transition \( s' \xrightarrow{} s'' \in S^2 \) is **enabled on** \( s \) if \( s = s' \). A run is **live wrt** \( L \in \lambda \) if either \( r \) has a nonempty suffix in which no transition in \( L \) is enabled, or every suffix of \( r \) includes an \( L \) transition. A run \( r \) is **live** if it is live wrt every \( L \in \lambda \). A run \( r \) is **correct** if it is safe and live.

**Observation 1 (Final State)** A state is **final** if no correct transition is enabled on it. A live computation is finite only if its last state is final.

The following is an example of a generic transition system over a given set of states. Here and in the other examples in this section the liveness condition \( \lambda \) is omitted and a computation is live if it is live wrt the correct transitions.

**Example 1 (G: Generic).** Given a set of states \( S \) with a designated initial state \( s_0 \in S \), a generic transition system over \( S \) is \( G = (S, s_0, TG) \) for some \( TG \subseteq S^2 \).

**Definition 3 (Specification; Safe, Live, Correct and Complete Implementation).** Given two transition systems \( TS = (S, s_0, T, \lambda) \) (the specification) and \( TS' = (S', s'_0, T', \lambda') \), an **implementation of TS by TS'** is a function \( \sigma : S' \to S \) where \( \sigma(s'_0) = s_0 \), in which case the pair \( (TS', \sigma) \) is referred to as an **implementation of TS**. Given a computation \( r' = s'_1 \xrightarrow{} s'_2 \xrightarrow{} \ldots \) of \( TS' \), \( \sigma(r') \) is the (possibly empty) computation \( \sigma(s'_1) \xrightarrow{} \sigma(s'_2) \xrightarrow{} \ldots \), with **stutter transitions** in which \( \sigma(s'_i) = \sigma(s'_{i+1}) \) removed. The implementation \( (TS', \sigma) \) of \( TS \) is **safe/live/correct** if \( \sigma \) maps every safe/live/correct \( TS' \) run \( r' \) to a safe/live/correct \( TS \) run \( \sigma(r') \), respectively, and is **complete** if every correct run \( r \) of \( TS \) has a correct run \( r' \) of \( TS' \) such that \( \sigma(r') = r \).

**Definition 4 (s: Locally Safe, Productive, Locally Complete).** Given two transition systems \( TS = (S, s_0, T, \lambda) \) and \( TS' = (S', s'_0, T', \lambda') \) and an implementation \( \sigma : S' \to S \). Then \( \sigma \) is:

1. **Locally Safe** if \( s'_0 \xrightarrow{} y_1 \xrightarrow{} y_2 \subseteq T' \) implies that \( s_0 \xrightarrow{} x_1 \xrightarrow{} x_2 \subseteq T \) for \( x_1 = \sigma(x'_1) \) and \( x_2 = \sigma(x'_2) \) in \( S \). If \( x_1 = x_2 \) then the \( T' \) transition \( x'_1 \xrightarrow{} x'_2 \) **stutters** \( T \).

2. **Productive** if for every \( L \in \lambda \) and every correct run \( r' \) of \( TS' \), either \( r' \) has a nonempty suffix \( r'' \) such that \( L \) is not enabled in \( \sigma(r'') \), or every suffix \( r'' \) of \( r' \) **activates** \( L \), namely \( \sigma(r'') \) has an \( L \)-transition.

3. **Locally Complete** if \( s'_0 \xrightarrow{} x_1 \xrightarrow{} x_2 \subseteq T' \) implies that \( s'_0 \xrightarrow{} x'_1 \xrightarrow{} x'_2 \subseteq T' \) for some \( x'_1, x'_2 \in S' \) such that \( x_1 = \sigma(x'_1) \) and \( x_2 = \sigma(x'_2) \).
Proposition 1 (σ Correct). If an implementation \( \sigma \) is locally safe and productive then it is correct, and if in addition it is locally complete then it is complete.

Intuitively, in an implementation \((TS', \sigma)\) of \(TS, TS'\) can be thought of as the ‘virtual hardware’ (e.g. the instruction set of a virtual machine or the machine language of an actual machine) and \(\sigma\) as specifying a ‘compiler’, that compiles programs in the high-level language \(TS\) into machine-language programs in \(TS'\). The mapping \(\sigma\) from \(TS'\) to \(TS\) is in inverse direction to that of a compiler; it thus specifies the intended behavior of compiled programs in terms of the behavior of their source programs, and in doing so can serve as the basis for proving a compiler correct. Note, though, that transition systems have no formal syntax, and can be thought of as specifying the operational semantics of existing or hypothetical programming languages.

Preparing an example implementation, we present the universal single-chain transition system \(SC\), and then show how it can implement any generic transition system \(G\), justifying the title ‘universal’.

Example 2 (SC: Single-Chain). Given a set \(S\) with a designated initial state \(s_0 \in S\), the single-chain transition system over \(S\) is \(SC = (S^+, s_0, TSC)\), where \(TSC\) includes every transition \(x \rightarrow x \cdot s\) for every \(x \in S^*\) and \(s \in S\).

Namely, an SC run can generate any sequence over \(S\).

From a programming-language perspective, some transition systems we will be concerned with are best viewed as providing the operational semantics for a set of programs over a given domain. With this view, in the current abstract setting, the programming of a transition system, namely choosing a program from this potentially-infinite set of programs, is akin to identifying a (computable) subset of the transition system. In our example, for the universal single-chain transition system \(SC\) to implement a specific instance of the generic transition system \(G\), a subset of \(SC\) has to be identified that corresponds to the transitions of \(G\), as shown next. But first we define the notion of a transition system subset.

Definition 5 (Transition System Subset). Given a transition system \(TS = (S, s_0, T, \lambda)\), a transition system \(TS' = (S', s'_0, T', \lambda')\) is a subset of \(TS\), \(TS' \subseteq TS\), if \(s'_0 = s_0\), \(S' \subseteq S\), \(T' \subseteq T\), and \(\lambda'\) is \(\lambda\) restricted to \(T'\).

The definition suggests at least two specific ways to construct a subset: Choosing a subset of the states and restricting the transitions to be only among these states; or choosing a subset of the transitions. Specifically, (i) Choose some \(S' \subseteq S\) and define \(T' := T/S'\), namely \(T' := \{ s \rightarrow s' \in T : s, s' \in S' \}\). (ii) Choose some \(T' \subseteq T\). We note that in practice there must be restrictions on the choice of a subset; to begin with, \(S'\) and \(T'\) should be computable.

We want to show that the universal single-chain transition system can implement any generic transition system. Hence the following definition:

Definition 6 (Can Implement). Given transition systems \(TS = (S, s_0, T, \lambda)\), \(TS' = (S', s'_0, T', \lambda')\), \(TS'\) can implement \(TS\) if there is a subset \(TS'' = (S'', s''_0, T'', \lambda'')\), \(TS'' \subseteq TS'\) and a correct and complete implementation \(\sigma : S'' \rightarrow S\) of \(TS\) by \(TS''\).
Next we demonstrate the application of the definitions above:

**Proposition 2.** The single-chain transition system \( SC \) over \( S \) can implement any generic transition system \( G \) over \( S \).

### 2.2 Composing Implementations

The key property of correct and complete implementations is their transitivity:

**Proposition 3 (Transitivity of Correct & Complete Implementations).** The composition of safe/live/correct/complete implementations is safe/live/correct/complete, respectively.

Our next example is the longest-chain transition system, which can be viewed as an abstraction of the longest-chain consensus protocols (e.g. Nakamoto [28]), since its consistency requirement entails that only the longest chain may be freely extended; other chains are bound to copy their next sequence element from a longer chain till they catch up, if ever, and only then may contribute a new element to the chain.

**Example 3 (LC: Longest-Chain).** Given a set \( S \) and \( n > 0 \), the LC longest-chain transition system over \( S \), \( LC = ((S^*)^n, c0, TLC) \), has sets of \( n \) sequences over \( S \) as states, referred to as \( n \)-chain configurations over \( S \), initial state \( c0 = \Lambda^n \), and as transitions \( TLC \) every \( c \rightarrow c' \) where \( c' \) is obtained from \( c \) by extending one sequence \( x \in c \) to \( x \cdot s \), \( s \in S \), provided that either \( x \) is a longest sequence in \( c \) or \( x \cdot s \) is a prefix of some \( y \in c \).

We wish to prove that the longest-chain transition system \( LC \) can implement the single-chain transition system \( SC \), and by transitivity of correct implementations, also implement any generic transition systems \( G \). The mathematical machinery developed next will assist in achieving this.

### 2.3 Monotonic Transition Systems for Distributed Computing

Unlike shared-memory systems, distributed systems have a state that increases in some natural sense as the computation progresses, e.g. through accumulating messages and extending the history of local states. This notion of monotonicity, once formalized, allows a simpler and more powerful mathematical treatment of transition systems for distributed computing.

So far we have used \( \leq \) to denote the prefix relation, which is a specific partial order. In the following we also use \( \preceq \) to denote a general partial order; the use should be clear from the context.

**Definition 7 (Partial Order).** A reflexive partial order on a set \( S \) is denoted by \( \leq_S \) (with \( S \) omitted if clear from the context), \( s \preceq s' \) stands for \( s \preceq s' \) & \( s' \preceq s \), and \( s \simeq s' \) for \( s \preceq s' \) & \( s' \preceq s \). The partial order is strict if \( s \simeq s' \) implies \( s = s' \) and unbounded if for every \( s \in S \) there is an \( s' \in S \) such that \( s \prec s' \). We say that \( s, s' \in S \) are consistent wrt \( \preceq \) if \( s \preceq s' \) or \( s' \preceq s \) (or both).
It is often possible to associate a partial order with a distributed system, wrt which the local state of each agent only increases. Therefore we focus on the following type of transition systems:

**Definition 8 (Monotonic & Monotonically-Complete Transition System).** Given a partial order \( \preceq \) on \( S \), a transition system \( TS = (S, s_0, T, \lambda) \) is **monotonic** with respect to \( \preceq \) if \( s \rightarrow s' \in T \) implies \( s \preceq s' \). It is **monotonically-complete wrt** \( \preceq \) if, in addition, \( s_0 \xrightarrow{\sigma} s \subseteq T \) and \( s \preceq s' \) implies that \( s \xrightarrow{\lambda} s' \subseteq T \).

Namely, computations of a monotonically-complete transition system not only ascend in the partial order, but may also reach, from any state, any larger state in the partial order. Note that since the partial order is unbounded, a monotonically-complete transition system has no final states. Many applications of this framework, including the examples herein, require proving that a transition system is monotonically-complete. The following approach is often helpful:

**Definition 9 (\( \epsilon \)-Monotonic Completeness).** A transition system \( TS = (S, s_0, T, \lambda) \), monotonic wrt a partial order \( \preceq \) on \( S \), is **\( \epsilon \)-monotonically-complete wrt** \( \preceq \) if infinite ascending chains in \( \preceq \) are unbounded and \( s \prec s'' \) implies that there is a transition \( s \rightarrow s' \in T \) such that \( s \prec s' \) and \( s' \preceq s'' \).

**Proposition 4 (\( \epsilon \)-Monotonic Completeness).** A transition system that is \( \epsilon \)-monotonically-complete is monotonically-complete.

**Proof (of Proposition 4).** Assume a transition system \( TS = (S, s_0, T, \lambda) \) that is \( \epsilon \)-monotonically-complete wrt a partial order \( \preceq \) on \( S \), and let \( s \preceq s'' \) for \( s, s'' \in S \).

We construct a computation \( s \xrightarrow{\sigma} s'' \in T \) iteratively as follows: Given \( (s, s'') \), if \( s = s'' \) we are done, else let the next transition of the computation be \( s \rightarrow s' \in T \) for some \( s' \) for which \( s \prec s', s' \preceq s'' \), which exists by assumption, and iterate with \( (s', s'') \). The constructed sequence is an ascending chain bounded by \( s'' \), which is finite by assumption, hence the iterative construction terminates with \( (s'', s'') \).

When transition systems are monotonically-complete wrt a partial order, the following Definition 10 and Theorem 1 can be a powerful tool in proving that one can correctly implement the other.

**Definition 10 (Order-Preserving Implementation).** Let transition systems \( TS = (S, s_0, T, \lambda) \) and \( TS' = (S', s'_0, T', \lambda') \) be monotonic wrt the partial orders \( \preceq \) and \( \preceq' \), respectively. Then an implementation \( \sigma : S' \rightarrow S \) of \( TS \) by \( TS' \) is **order-preserving wrt** \( \preceq \) and \( \preceq' \) if:

1. **Up condition:** \( y_1 \preceq' y_2 \) implies that \( \sigma(y_1) \preceq \sigma(y_2) \)
2. **Down condition:** \( s_0 \xrightarrow{\sigma} x_1 \subseteq T, x_1 \preceq x_2 \) implies that there are \( y_1, y_2 \in S' \) such that \( x_1 = \sigma(y_1), x_2 = \sigma(y_2), s'_0 \xrightarrow{\lambda'} y_1 \subseteq T' \) and \( y_1 \preceq' y_2 \).

Note that if \( \preceq' \) is induced by \( \sigma \) and \( \preceq \), namely defined by \( y_1 \preceq' y_2 \) if \( \sigma(y_1) \preceq \sigma(y_2) \), then the Up condition holds trivially. The following Theorem is the linchpin of the proofs of protocol stack theorems here and in other distributed computing applications of the framework.
Theorem 1 (Correct & Complete Implementation Among Monotonically-Complete Transition Systems). Assume two transition systems \( TS = (S, s_0, T, \lambda) \) and \( TS' = (S', s'_0, T', \lambda') \), monotonically-complete wrt the unbounded partial orders \( \preceq \) and \( \preceq' \), respectively, and an implementation \( \sigma : S' \to S \) of \( TS \) by \( TS' \). If \( \sigma \) is order-preserving and productive then it is correct and complete.

If all transition systems in a protocol stack are monotonically-complete, then Theorem 1 makes it sufficient to establish that an implementation of one protocol by the next is order-preserving and productive to prove it correct. A key challenge in showing that Theorem 1 applies is proving that the implementation satisfies the Down condition (Def. 10), which can be addressed by finding an ‘inverse’ to \( \sigma \) as follows:

Observation 2 (Representative Implementation State) Assume \( TS \) and \( TS' \) as in Theorem 1 and an implementation \( \sigma : S' \to S \) that satisfies the Up condition of Definition 10. If there is a function \( \hat{\sigma} : S \to S' \) such that \( x = \sigma(\hat{\sigma}(x)) \) for every \( x \in S \), and \( x_1 \preceq x_2 \) implies that \( \hat{\sigma}(x_1) \preceq' \hat{\sigma}(x_2) \), then \( \sigma \) also satisfies the Down condition.

Proposition 5. LC can implement SC.

Proof (outline of Proposition 5). We show that both SC and LC are monotonically-complete wrt the prefix relation \( \preceq \) (Observations 3, 4) and that the implementation \( \sigma_2 \) of SC by LC is order preserving and productive (Proposition 12). Hence, according to Theorem 1, \( \sigma_2 \) is correct and complete. \( \square \)

Definition 11 (\( \sigma_2 \)). The implementation \( \sigma_2 \) maps every \( n \)-chain configuration \( c \) to the longest chain in \( c \) if it is unique, and is undefined otherwise.

In our example, the longest-chain transition system LC implements the single-chain transition system SC. But SC does not implement the generic transition system G – a subset of it, SC1, does. So, in order to prove that LC can implement G, solely based on the implementation of SC by LC, without creating a custom subset of LC for the task, the following Proposition is useful.

Proposition 6 (Restricting a Correct Implementation to a Subset). Let \( \sigma : C2 \to S1 \) be an order-preserving implementation of \( TS1 = (S1, s1, T1, \lambda1) \) by \( TS2 = (C2, s2, T2, \lambda2) \), monotonically-complete respectively with \( \preceq_1 \) and \( \preceq_2 \). Let \( TS1' = (S1', s1, T1', \lambda1') \subseteq TS1 \) and \( TS2' = (C2', s2, T2', \lambda2') \subseteq TS2 \) defined by \( C2' := \{ s \in C2 : \sigma(s) \in S1' \} \), with \( T2' := T2/C2' \), and assume that both subsets are also monotonically-complete wrt \( \preceq_1 \) and \( \preceq_2 \), respectively. If \( y1 \to y2 \in T2' \) & \( \sigma(y1) \in S1' \) implies that \( \sigma(y2) \in S1' \) then the restriction of \( \sigma \) to \( C2' \) is a correct and complete implementation of \( TS1' \) by \( TS2' \).

Corollary 1. The longest-chain transition system LC is universal for generic transition systems.
More generally, Proposition 6 is useful in the following scenario. Assume that protocols are specified via transition systems, as elaborated below. Then in a protocol stack of, say, three protocols P1, P2, P3, each implementing its predecessor, it may be the case that for the middle protocol P2 to implement the full top protocol P1, a subset $P_2'$ of P2 is needed. But, it may be desirable for P3 to implement the full protocol P2, not just its subset $P_2'$, as P2 may have additional applications beyond just implementing P1. In particular, there are often application for which an implementation by a middle protocol in the stack is more efficient than an implementation by the full protocol stack. The following proposition enables that, see Figure 2. Note that, as shown in the figure, the implementing transition system $TS_2$ that implements $TS_1$ could in turn be a subset of a broader unnamed transition system.

3 Multiagent Transition Systems: Centralized, Distributed, Synchronous and Asynchronous

3.1 Multiagent Transition Systems

We assume a domain $\Pi$ of agents. While the set $\Pi$ may be infinite, here we only consider finite subsets of $\Pi$. In the following, we use $a \neq b \in X$ as a shorthand for $a \in X \land b \in X \land a \neq b$.

In the context of multiagent transition systems, the state of the system is referred to as configuration, so as not to confuse it with the local states of agents in a distributed multiagent transition system, defined next.

**Definition 12 (Multiagent Transition System).** Given agents $P \subseteq \Pi$, a transition system $TS = (C, c_0, T, \lambda)$, with configurations $C$, initial configuration $c_0$, correct transitions $T \subseteq C^2$, and a liveness condition $\lambda$ on $T$, is **multiagent over $P$** if there is a multiagent partition $C^2 = \bigcup_{p \in P} CC_p$ of $C^2$ into disjoint sets $CC_p$ indexed by $P$, $CC_p \cap CC_q = \emptyset$ for every $p \neq q \in P$. A transition $t = s \rightarrow s' \in CC_p$ is referred to as a $p$-transition, and the set of correct $p$-transitions $T_p$ is defined by $T_p := T \cap CC_p$, for every $p \in P$.

Note that $CC_p$ includes all possible behaviors of agent $p$, both correct and faulty, and $T_p$ includes only the agent’s correct behaviors.

**Definition 13 (Safe, Live & Correct Agents).** Given a multiagent transition system $TS = (C, c_0, T, \lambda)$ over $P$ and a run $r$ of $TS$, an agent $p$ is **safe in $r$** if $r$ includes only correct $p$-transitions; is **live in $r$** if for every $L \in \lambda$ for which $L \subseteq T_p$, $r$ is live wrt $L$; and is **correct in $r$** if $p$ is safe and live in $r$.

Note that if $\lambda = \{T_p : p \in P\}$, namely the liveness condition is the partition of correct transitions to agents, then an agent $p$ is live if it is live wrt its correct $p$-transitions $T_p$.

Next, the generic transition system (Example 1) is modified to be multiagent. In the generic shared-memory multiagent transition system GS defined next, all
agents operate on the same shared global state. Yet, the transitions of different agents are made disjoint by capturing abstractly the reality of shared-memory multiprocessor systems: Each configuration incorporates, in addition to a shared global state \( s \in S \), also a unique program counter for each agent. The program counter of agent \( p \) is advanced when a \( p \)-transition is taken.

**Example 4 (GS: Generic Shared Memory).** Given a set of agents \( P \subseteq \Pi \) and states \( S \) with a designated initial state \( s_0 \), a generic shared-memory multiagent transition system over \( P \) and \( S \), \( GS = (C, c_0, TGS) \), has configurations \( C = S \times \mathbb{N}^P \) that include a shared global state in \( S \) and a program counter for each agent \( p \in P \), initial state \( c_0 = (s_0, \{0\}^P) \), and transitions \( TGS = \bigcup_{p \in P} TGS_p \subseteq C^2 \), where each \( p \)-transition \( (s, i) \to (s', i') \in TGS_p \) satisfies \( i'_p = i_p + 1 \) and \( i'_q = i_q \) for every \( q \neq p \in P \).

Note that \( TGS \) is arbitrary, and different agents may or may not be able to change the shared global state in the same way. But each transition identifies the agent \( p \) making the change by advancing \( p \)'s program counter.

Next, the single-chain transition system \( SC \) (Example 2) is modified to the multiagent transition system for single-chain consensus \( SCC \). As \( SCC \) is monotonic, program counters are not needed; it is sufficient to identify the agent contributing the next element to the shared global chain to make transitions by different agents disjoint.

**Example 5 (SCC: Single-Chain Consensus).** Given a set of agents \( P \subseteq \Pi \) and a set \( S \), the single-chain consensus multiagent transition system over \( P \) and \( S \) is \( SCC = ((S \times P)^*, \Lambda, TSCC) \), with each configuration being a sequence of agent-identified states \( (s, p) \) of a state \( s \in S \) and an agent \( p \in P \), and \( TSCC \) includes every transition \( x \to x \cdot (s, p) \) for every \( x \in (S \times P)^* \), \( s \in S \) and \( p \in P \).

Namely, an SCC run can generate any sequence of agent-identified elements of \( S \), where any agent may contribute any element to any position in the sequence.

Next, we show that SCC can implement GS, making single-chain consensus universal for shared-memory multiagent transition systems.

**Proposition 7.** SCC over \( P \subseteq \Pi \) and \( S \) can implement any generic shared-memory multiagent transition system \( GS \) over \( P \) and \( S \).

### 3.2 Centralized and Distributed Multiagent Transition Systems

Having introduced centralized/shared-memory multiagent transition systems, and before introducing distributed ones, we formalize the two notions:

**Definition 14 (Centralized and Distributed Multiagent Transition System).** A multiagent transition system \( TS = (C, c_0, T, \lambda) \) over \( P \) with multiagent partition \( C^2 = \bigcup_{p \in P} CC_p \) is **distributed** if:

1. \( C = S^P \) for some set \( S \), referred to as **local states**, namely each configuration \( c \in C \) consists of a set of local states in \( S \) indexed by \( P \), in which case we use \( c_p \in S \) to denote the local state of \( p \in P \) in configuration \( c \in C \), and
2. Any $p$-transition $c \to c'$ in $CC_p$ satisfies that $c'_p \neq c_p$ and $c'_q = c_q$ for every $q \neq p \in P$.

Else $TS$ is **centralized**.

Namely, in a distributed transition system a $p$-transition (correct or faulty) can only change the local state of $p$. In other words, even a faulty agent cannot affect the local states of other agents. As a shorthand, we will omit 'multiagent' from distributed multiagent transition systems, and instead of presenting a distributed multiagent transition system over $P$ and $S$ as $TS = (S^P, c_0, T, \lambda)$, we will refer to it as the distributed transition system $TS = (P, S, c_0, T, \lambda)$.

Next, we modify the longest-chain transition system $LC$ (Example 3) to become the distributed transition system for longest-chain consensus $LCC$, in which each agent has a chain as its local state.

**Example 6 (LCC: Longest-Chain Consensus).** Given a set of agents $P \subseteq \Pi$ and states $S$, the LCC distributed longest-chain transition system, $LCC = (P, (S \times P)^*, c_0, T_{LCC}, \lambda)$, has sequences over $S \times P$ as local states, an empty sequence as the initial local state $c_0 = \{\Lambda\}^P$, and as $p$-transitions $T_{LCC}$ every $c \to c'$ where $c'$ is obtained from $c$ by only extending $c_p$, $c'_p = c_p \cdot (s, p')$, $s \in S$, $p' \in P$, and $c'_q = c_q$ for every $q \neq p \in P$, provided that either $p = p'$ and $c_p$ is a longest sequence in $c$, or $p' \neq p$ and $c_p \cdot (s, p')$ is a prefix of $c_q$ in $c$ for some $q \neq p \in P$. The liveness condition $\lambda = \{T_p : p \in P\}$ is the multiagent partition over correct transitions.

Note that the transition system, while distributed, is synchronous (a notion defined formally below), as an agent’s ability to extend its local chain by a certain element depends on the present local states of other agents. Next, we show that $LCC$ can implement SCC, making the longest-chain consensus distributed transition system $LCC$ universal for shared-memory multiagent transition systems.

**Proposition 8.** $LCC$ can implement SCC.

We noted informally why we consider $LCC$ synchronous. Next, we define the notions of synchronous and asynchronous distributed transition systems, prove that $LCC$ is synchronous and investigate an asynchronous distributed transition system and its implementation of the LCC.

### 3.3 Synchronous and Asynchronous Distributed Multiagent Transition Systems

A partial order $\preceq$ over a set of local states $S$ naturally extends to configurations $C = S^P$ over $P \subseteq \Pi$ and $S$ by $c \preceq c'$ for $c, c' \in C$ if $c_p \preceq c'_p$ for every $p \in P$.

**Definition 15 (Distributed Transition System; Synchronous and Asynchronous).** Given agents $P \subseteq \Pi$, local states $S$, and a distributed transition system $TS = (P, S, c_0, T, \lambda)$, then $TS$ is **asynchronous** wrt a partial order $\preceq$ on $S$ if:
1. TS is monotonic wrt \( \preceq \), and
2. for every \( p \)-transition \( c \rightarrow c' \in T \), \( T \) also includes the \( p \)-transition \( d \rightarrow d' \) for every \( d, d' \in C \) that satisfy the following asynchrony condition:

\[
    c \preceq d, (c_p \rightarrow c'_p) = (d_p \rightarrow d'_p), \text{ and } d'_q = d_q \text{ for every } q \neq p \in P.
\]

If no such partial order on \( S \) exists, then TS is synchronous.

With this definition, we note that the distributed longest-chain transition system LCC is not asynchronous wrt the prefix relation, as an enabled transition to extend the local chain can become disabled if some other chain extends and becomes longer. We argue that this is the case wrt any partial order.

**Proposition 9.** LCC is synchronous.

Next we devise an asynchronous block dissemination transition system ABD, and prove its universality by using it to implement the synchronous LCC.

**Definition 16 (Block).** Given agents \( P \subseteq \Pi \) and states \( S \), a block over \( P \) and \( S \) is a triple \((p,i,s)\) such a block is referred to as an \( i \)-indexed \( p \)-block with payload \( s \).

**Example 7 (ABD: Asynchronous Distributed Block Dissemination).** Given a set of agents \( P \subseteq \Pi \) and states \( S \) that do not include the undefined element \( \perp \notin S \), the asynchronous distributed block dissemination transition system, ABD = \((P,B,c_0,TABD,\lambda)\), has local states \( B \) being all finite sets of blocks over \( P \) and \( S \cup \{\perp\} \), an empty set as the initial local state \( c_0 = \{\emptyset\} \), and \( TABD \) has every \( p \)-transition \( c \rightarrow c' \) for every \( p \in P \), where \( c' \) is obtained from \( c \) by adding a block \( b = (p',i,s) \) to \( c_p \), \( c'_p = c_p \cup \{b\} \), \( p' \in P \), \( i \in \mathbb{N} \), \( s \in S \cup \{\perp\} \), and either:

1. \( p \)-Creates-\( b \): \( p' = p \), \( i = i' + 1 \), where \( i' := \max \{j : (p,j,s) \in c_p\} \), or
2. \( p \)-Receives-\( b \): \( p' \neq p \), \( (p',i,s) \in c_q \setminus c_p \) for some \( q \neq p \in P \).

The liveness condition \( \lambda \) places two transitions in the same set if they have identical labels, \( p \)-Creates-\( b \) or \( p \)-Receives-\( b \) as the case may be.

In other words, every agent \( p \) can either add a consecutively-indexed \( p \)-block to its local state, possibly with \( \perp \) as payload, or obtain a block it does not have from some other agent. Next, we explore some properties of ABD: Fault-resilient dissemination and equivocation detection. We use ‘\( p \) knows \( b \)’ in a run \( r \) to mean that \( b \in c_p \) for some \( c \in r \).

While in ABD agents do not explicitly disseminate blocks they know to other agents, only receive blocks that they do not know from other agents, faulty agents may cause partial dissemination by deleting a block from their local state after only some of the agents have received it. The following proposition states that faulty agents cannot prevent correct agents from eventually sharing all the blocks that they know, including blocks created and partially disseminated by faulty agents.

**Proposition 10 (ABD Block Liveness).** In an ABD run, if a correct agent knows a block \( b \) then eventually all correct agents know \( b \).
Definition 17 (Equivocation). An equivocation by agent \( p \) consists of two \( p \)-blocks \( b = (p, i, s) \), \( b' = (p, i', s') \) where \( i = i' \) but \( s \neq s' \). An agent \( p \) is an equivocator in \( B \) if \( B \) includes an equivocation by \( p \). A set of blocks \( B \) is equivocation-free if it does not include an equivocation.

The following corollary states that if an agent \( p \) tries to mislead (e.g., double spend) correct agents by disseminating to different agents equivocating blocks, then eventually all correct agents will know that \( p \) is an equivocator.

Corollary 2 (ABD Equivocation Detection). In an ABD run, if two blocks \( b, b' \) of an equivocation by agent \( p \) are each known by a different correct agent, then eventually all correct agents know that \( p \) is an equivocator.

Next, we prove that asynchronous distributed block dissemination ABD can implement the synchronous distributed longest-chain LCC. In fact, this implementation offers a naive distributed asynchronous ordering consensus protocol. Its lack of resilience to equivocation and to fail-stop agents, implied by the FLP theorem [10], is discussed in the next section. This limitation reflects on the implementation presented here and not on ABD: The Cordial Miners family of protocols [20] employs a more concrete and practical (blocklace-based [33]) variant of asynchronous block dissemination to construct Byzantine fault-resilient order consensus protocols (aka Byzantine Atomic Broadcast) for the models of asynchrony and eventual synchrony.

Proposition 11. ABD can implement LCC.

4 Safety Faults, Liveness Faults, and their Resilience

A safety fault is a subset (or all) of the incorrect transitions, and a liveness fault is a subset of the liveness condition. A computation performs a safety fault \( F \) if it includes an \( F \) transition. It performs a liveness fault \( \lambda' \subseteq \lambda \) if it is not live wrt a set \( L \in \lambda' \). Formally:

Definition 18 (Safety and Liveness Faults). Given a transition system \( TS = (S, s_0, T, \lambda) \), a safety fault is a set of incorrect transitions \( F \subseteq S^2 \setminus T \). A computation performs a safety fault \( F \) if it includes a transition from \( F \). A liveness fault is a a subset \( \lambda' \subseteq \lambda \) of the liveness condition \( \lambda \). An infinite run performs a liveness fault \( \lambda' \) if it is not live wrt \( L \) for some \( L \in \lambda' \).

Note that any safety fault can be modelled with the notion thus defined, by enlarging \( S \) and thus expanding the set of available incorrect transitions \( S^2 \). Similarly, any liveness fault can be modeled by revising \( \lambda \) accordingly.

Definition 19 (Safety-Fault Resilience). Given transition systems \( TS = (S, s_0, T, \lambda) \), \( TS' = (S', s'_0, T', \lambda') \) and a safety fault \( F \subseteq S^2 \setminus T' \), a correct implementation \( \sigma : S' \to S \) is \( F \)-resilient if for any live \( TS' \) run \( r' \subseteq T \cup F \), the run \( \sigma(r') \) is correct.
In other words, a safety-fault-resilient implementation does not produce incorrect transitions of the specification even if the implementation performs safety faults, and it produces a live run if the implementation run is live.

Next we compare the resilience of single-chain consensus SCC and longest-chain consensus LCC to the safety fault in which an agent trashes the chain by adding junk to it. We show that the implementation of the generic shared-memory GS by LCC is more resilient to such faults than the implementation by SCC: In SCC such a faulty transition terminates the run, violating liveness; in LCC it does not, as long as there is at least one non-faulty agent.

The following Theorem addresses the composition of safety-fault-resilient implementations. See Figure 3.

**Theorem 2 (Composing Safety-Fault-Resilient Implementations).** Assume transition systems $TS_1 = (S_1, s_1, T_1, \lambda_1)$, $TS_2 = (S_2, s_2, T_2, \lambda_2)$, $TS_3 = (S_3, s_3, T_3, \lambda_3)$, correct implementations $\sigma_{s_1} : S_2 \rightarrow S_1$ and $\sigma_{s_2} : S_3 \rightarrow S_2$, and let $\sigma_{s_1} \equiv \sigma_{s_2} \circ \sigma_{s_3}$. Then:

1. If $\sigma_{s_3}$ is resilient to $F_3 \subseteq S_3^2 \setminus T_3$, then $\sigma_{s_1}$ is resilient to $F_3$.
2. If $\sigma_{s_1}$ is resilient to $F_2 \subseteq S_2^2 \setminus T_2$, and $F_3 \subseteq S_3^2 \setminus T_3$ satisfies $\sigma_{s_3}(F_3) \subseteq F_2$, then $\sigma_{s_1}$ is resilient to $F_3$.
3. These two types of safety-fault resilience can be combined for greater resilience: If $\sigma_{s_1}$ is $F_2$-resilient, $\sigma_{s_2}$ is $F_3$-resilient, $F_3' \subseteq S_2^2 \setminus T_3$, and $\sigma_{s_3}(F_3') \subseteq F_2$, then $\sigma_{s_1}$ is resilient to $F_3 \cup F_3'$.

**Example 8 (Resilience to Safety Faults in Implementations by SCC and LCC).** For SCC, consider the safety fault $F_1$ to be the faulty $q$-transitions $c \rightarrow c \cdot 0$ for every configuration $c$ and some $q \in P$. For LCC, consider the safety fault $F_2$ to be the faulty $q$-transitions $c_q \rightarrow c_q \cdot 0$ for every configuration $c$ and some agent $q \in P$. Then a faulty SCC run $r$ with an $F_1$ transition cannot be continued, and hence $\sigma_2(r)$ is not live and hence incorrect. On the other hand, in a faulty LCC run $r$ with $F_2$ transitions, the faulty transitions are mapped by $\sigma_2m$ to stutter, the run can continue and the implementation is live as long as at least one agent is not faulty. Note that this holds for the implementation of SCC by LCC, as well as for the composed implementation of GS by LCC, as stated by the following Theorem 2 (the $F_3$ case).

Next we consider the implementation of longest-chain consensus LCC by asynchronous block dissemination ABD, and it non-resilience to the safety fault of equivocation.

**Example 9 (Non-Resilience to Equivocation of the implementation of LCC by ABD).** Consider the implementation $\sigma_3$ of LCC $\equiv (P, (S \times P)^*, c_0, TLCC)$ by ABD $\equiv (P, B, c_0, TABD, \lambda)$, and let $F \subseteq (B_P)^2$ include equivocations by a certain agent $p \in P$ for every configuration, namely for every configuration $c \in B_P$, in which $c_p$ includes a $p$-block $b = (p, i, s)$, $F$ includes the $p$-transition $c_p \rightarrow c_p \cup \{b'\}$ for $b' = (p, i, s')$ for some $s' \neq s \in S$. A run $r$ with such an equivocating transition by $p$ may include subsequently a $q$-Receives-$b$ and $q'$-Receives-$b'$ transitions, following which, say in configuration $c'$, the chain computed by $\sigma_3(c')$ for $q$ and for $q'$ would not be consistent, indicating $\sigma_3(r)$ to be faulty (not safe).
Definition 20 (Can Implement with Safety-Fault Resilience). Given transition systems $TS = (S, s_0, T, \lambda)$, $TS' = (S', s'_0, T', \lambda')$ and $F \subseteq S'^2 \setminus T'$, $TS'$ can implement $TS$ with $F$-resilience if there is a subset $TS'' = (S'', s''_0, T'', \lambda'') \subseteq TS'$, $F \subset S'' \times S''$, and an $F$-resilient implementation $\sigma : S'' \to S$ of $TS$ by $TS''$.

The requirement $F \subset S'' \times S''$ ensures that the subset $TS''$ does not simply ‘define away’ the faulty transitions $F$.

Definition 21 (Can Implement with Liveness-Fault Resilience). Given transition systems $TS = (S, s_0, T, \lambda)$, $TS' = (S', s'_0, T', \lambda')$, then $TS'$ can implement $TS$ with $\bar{\lambda}$-resilience, $\bar{\lambda} \subseteq \lambda'$, if there is a subset $TS'' = (S'', s''_0, T'', \lambda'') \subseteq TS'$, and an implementation $\sigma : S'' \to S$ of $TS$ by $TS''$, resilient to $\bar{\lambda}$ restricted to $\lambda''$.

As an example of resilience to a liveness fault, consider the following:

Example 10 (Resilience to fail-stop agents of the implementation of SCC and GC by LCC). Consider $LCC = (P, (S \times P)^*, c_0, TLCC, \lambda)$, and recall that the liveness condition $\lambda = \{T_p : p \in P\}$ is the multiagent partition over correct transitions. An LCC run $r$ with a liveness fault $\lambda'$ may have all agents $p$ for which $T_p \in \lambda'$ fail-stop after some prefix of $r$. Still, at least one live agent remains by the assumption that $\lambda'$ is a strict subset of $\lambda$, and hence $\sigma_2(r)$ is a live (and hence correct) LCC run. Thus $\sigma_3$ is resilient to any liveness fault of LCC provided at least one agent remains live. Next, consider the implementation of GC by LCC. First, we defined a subset SCC1 of SCC to implement GC. Then we defined LCC1 a subset of LCC to implement SCC1. Such a composed implementation is resilient to fail-stop agents ($\bar{\lambda}$ in the example above), where their transitions are restricted to LCC1 ($\lambda''$ in the definition above).

5 Grassroots Protocols and their Implementation

In the following we assume that the set of local states $S$ is a function of the set of participating agents $P \subseteq \Pi$. Intuitively, the local states could be sets of signed and/or encrypted messages sent by members of $P$ to members of $P$; NFTs created by and transferred among members of $P$; or blocks signed by members of $P$, with hash pointers to blocks by other members of $P$. With this notion, we define a family of transition systems to have for each set of agents $P \subseteq \Pi$ one transition system that specifies all possible behaviors $T(P)$ of $P$ over $S(P)$ and a liveness condition $\lambda(P)$ over $T(P)$.

Definition 22 (Family of Multiagent Transition Systems; Protocol; Asynchronous; Subset). Assume a function $S$ that maps each set of agent $P \subseteq \Pi$ into a set of local states $S(P)$. A family $F$ of multiagent transition systems over $S$ is a set of transition systems such that for each set of agents $P \subseteq \Pi$ there is one transition system $TS(P) = (C(P), c_0(P), T(P), \lambda(P)) \in F$ with
configurations $C(P)$ over $P$ and $S(P)$, transitions $T(P)$ over $C(P)$ and a liveness condition $\lambda(P)$. The states of $\mathcal{F}$ are $S(\mathcal{F}) := \bigcup_{P \subseteq \Pi} S(P)$. If a family of multiagent transition systems is distributed then we refer to it as a protocol, which is asynchronous if every transition system in $\mathcal{F}$ is asynchronous, and is a subset of protocol $\mathcal{F}'$ if for every $P \subseteq \Pi$, $TS(P)$ is a subset of $TS'(P)$.

For simplicity and to avoid notational clutter, we often assume a given set of agents $P$ and refer to the representative member of $\mathcal{F}$ over $P$, rather than to the entire family $\mathcal{F}$. Furthermore, when the family $\mathcal{F}$ and the set of agents $P$ are given we sometimes refer to the protocol $TS(P) = (P, S(P), c0(P), T(P), \lambda(P)) \in \mathcal{F}$ simply as $TS = (P, S, c0, T, \lambda)$. Next, we define the notion of a grassroots protocol. Intuitively, in a grassroots protocol, the behavior of agents in a small community that runs the protocol is not constrained by its context, for example by this community being composed with another community or, equivalently, being embedded within a larger community. Yet, the protocol allows agents in two communities placed together to behave in new ways not possible when the two communities operate the protocol in isolation. In particular, agents may interact with each other across community boundaries. This supports the grassroots deployment of a distributed system – multiple independent disjoint deployments at different locations and over time, which may subsequently interoperate once interconnected.

We define the notion of a grassroots protocol with two auxiliary notions: the projection of a configuration and the union of distributed transition systems, defined next.

**Definition 23 (Projection of a Configuration).** Let $P' \subset P \subset \Pi$ and let $TS = (P, S, c0, T)$ be a distributed transition system. The projection of a configuration $c \in S^P$ on $P'$, $c/P'$, is the configuration $c'$ over $P'$ satisfying $c'_p = c_p$ for all $p \in P'$.

The union of two distributed transition systems over disjoint sets of agents includes all transitions in which one component makes its own transition and the other component stands still, implying that the runs of the union include exactly all interleavings of the runs of its components.

**Definition 24 (Union of Distributed Transition Systems).** Let $TS1 = (P1, S1, c01, T1)$, $TS2 = (P2, S2, c02, T2)$ be two distributed transition systems, $P1 \cap P2 = \emptyset$. Then the union of $TS1$ and $TS2$, $TS := TS1 \cup TS2$, is the multiagent transition systems $TS = (P1 \cup P2, S1 \cup S2, c0, T)$ with initial state $c0$ satisfying $c0/P1 = c01, c0/P2 = c02$, and all $p$-transitions $c \rightarrow c' \in T$, $p \in P$, satisfying $p \in P1 \land (c/P1 \rightarrow c'/P1) \in T1 \land c/P2 = c'/P2$ or $p \in P2 \land (c/P2 \rightarrow c'/P2) \in T2 \land c/P1 = c'/P1$.

The notion of a grassroots composition requires the composed system to include all computations each component can do on its own, and then some. In other words, put together, each component of the composed transition system can still behave independently, as if it is on its own; but the composed system also has additional behaviors.
Definition 25 (Grassroots). A protocol $\mathcal{F}$ supports grassroots composition, or is grassroots, if for every $\emptyset \subset P_1, P_2 \subset \Pi$ such that $P_1 \cap P_2 = \emptyset$, the following holds:

$$TS(P_1) \cup TS(P_2) \subset TS(P_1 \cup P_2)$$

Example 11 (Synchronous LCC is not grassroots; Asynchronous ABD is). We note that the union of two longest-chain consensus LCC transition systems over disjoint sets of agents is not a subset of an LCC transition system. The reason is that since the component transition systems operate independently, the chains they create—in particular their longest chains—can be inconsistent. Hence their union $LCC(P_1) \cup LCC(P_1)$ may have computations that reach inconsistent states, computations that are not in $LCC(P_1 \cup P_2)$. Hence LCC is not grassroots. On the other hand, ABD is: In two disjoint components $ABD(P_1)$ and $ABD(P_2)$ agents create and receive blocks among themselves, behaviors that are also available in $ABD(P_1 \cup P_2)$. But they also include new ones, with $p$-Receives-$b$ transitions for blocks created by agents in the other component.

Next, we describe a sufficient condition for a protocol to be grassroots:

Definition 26 (Monotonic and Asynchronous Protocols). Let $\mathcal{F}$ be a protocol. A partial order $\preceq$ over $S(\mathcal{F})$ is preserved under projection if for every $P_2 \subset P_1 \subseteq \Pi$ and every two configurations $c, c'$ over $P_1$ and $S(P_1)$, $c \preceq c'$ implies that $c/P_2 \preceq c'/P_2$. A protocol $\mathcal{F}$ is monotonic wrt a partial order $\preceq$ over $S(\mathcal{F})$ if $\preceq$ is preserved under projection and every member of $\mathcal{F}$ is monotonic wrt $\preceq$; it is asynchronous wrt $\preceq$ if, in addition, every member of $\mathcal{F}$ is asynchronous wrt $\preceq$.

A protocol is interactive if two sets of agents can perform together computations they cannot perform in isolation, and is non-interfering if a transition that can be carried out by a group of agents can still be carried out if there are additional agents that observe it from their initial state. Formally:

Definition 27 (Interactive & Non-Interfering Protocol). A protocol $\mathcal{F}$ is interactive if for every $\emptyset \subset P_1, P_2 \subset \Pi$ such that $P_1 \cap P_2 = \emptyset$, the following holds: $TS(P_1 \cup P_2) \not\subseteq TS(P_1) \cup TS(P_2)$. It is non-interfering if for every $P' \subset P \subset \Pi$, with transition systems $TS = (P, S(P), c_0, T), TS' = (P', S(P'), c_0', T') \in \mathcal{F}$, and every transition $c_1 \rightarrow c_2 \in T$, $T$ includes the transition $c_1' \rightarrow c_2' \in T'$ for which $c_1' = c_1/P'$, $c_2' = c_2/P'$, and $c_{1p} = c_{2p} = c_0_p$ for every $p \in P \setminus P'$.

Theorem 3 (Grassroots Protocol). An asynchronous, interactive, and non-interfering protocol is grassroots.

Example 12 (ABD is grassroots, again). Examining ABD, it can be verified that it is an asynchronous, interactive and non-interfering protocol. We have argued above that ABD is interactive. It is non-interfering as agents in their initial state do not interfere with other agents performing $p$-Creates-$b$ or $p$-Receives-$b$ transitions amongst themselves. And we have already concluded (Example 7) that ABD is asynchronous.
We note that client-server/cloud systems, blockchain protocols with hardcoded seed miners, e.g. Bitcoin \cite{9}, and permissioned consensus protocols, e.g. Byzantine Atomic Broadcast, with a predetermined set of participants \cite{35,20}, are all not grassroots. However, Theorem 3 and the examples above do not imply that ordering consensus protocols cannot be grassroots. In fact, they can be, provided that the participants (seed or all) in an instance of an ordering consensus protocol are not provided \textit{a priori}, as in LCC and in standard descriptions of consensus protocols, but are determined by the agents themselves in a grassroots fashion according to the protocol. A blocklace-based grassroots consensus protocol stack that demonstrates this is presented elsewhere \cite{33}.

6 Conclusions

Multiagent transition systems come equipped with powerful tools for specifying distributed protocols and for proving the correctness and fault-resilience of implementations among them. The tools are best applied if the transition systems are monotonically-complete wrt a partial order, as is often the case in distributed protocols and algorithms. Employing this framework in the specification of a grassroots ordering consensus protocol stack has commenced \cite{33}, with sovereign cryptocurrencies \cite{34} and an efficient Byzantine atomic broadcast protocol \cite{20} as the first applications.
References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
2. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 15(1), 73–132 (1993)
3. Boudol, G.: Asynchrony and the pi-calculus. Ph.D. thesis, INRIA (1992)
4. Bracha, G.: Asynchronous byzantine agreement protocols. Information and Computation 75(2), 130–143 (1987)
5. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001), https://eprint.iacr.org/2000/067.pdf, revised 2020
6. Cristian, F., Aghili, H., Strong, R., Dolev, D.: Atomic broadcast: From simple message diffusion to byzantine agreement. Information and Computation 118(1), 158–179 (1995)
7. Das, S., Xiang, Z., Ren, L.: Asynchronous data dissemination and its applications. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2721 (2021)
8. Das, S., Xiang, Z., Ren, L.: Asynchronous data dissemination and its applications. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 2705–2721. CCS ’21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484808 https://doi.org/10.1145/3460120.3484808
9. Developer, B.: P2p network (Retrieved 20222). https://developer.bitcoin.org/devguide/p2p_network.html
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32(2), 374–382 (1985)
11. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 372–385 (1996)
12. Francalanza, A., Hennessy, M.: A theory for observational fault tolerance. The Journal of Logic and Algebraic Programming 73(1-2), 22–50 (2007)
13. Giridharan, N., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Bullshark: Dag bft protocols made practical. arXiv preprint arXiv:2201.05677 (2022)
14. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlović, M., Seredinschi, D.A.: The consensus number of a cryptocurrency. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 307–316 (2019)
15. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. Communications of the ACM 60(7), 83–92 (2017)
16. Hesselink, W.H.: Deadlock and fairness in morphisms of transition systems. Theoretical computer science 59(3), 235–257 (1988)
17. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
18. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and kripke logical relations. ACM SIGPLAN Notices 47(1), 59–72 (2012)
19. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is dag. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. p. 165–175. PODC’21, Association for Computing Machinery, New York,
20. Keidar, I., Naor, O., Shapiro, E.: Cordial miners: A family of simple, efficient and self-contained consensus protocols for every eventualty. arXiv preprint arXiv:2205.09174 (2022)

21. Krogh-Jespersen, M., Timany, A., Ohlenbusch, M.E., Gregersen, S.O., Birkedal, L.: Aneris: A mechanised logic for modular reasoning about distributed systems. In: ESOP. pp. 336–365 (2020)

22. Lamport, L.: Specifying concurrent systems with tla+. NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES 173, 183–250 (1999)

23. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)

24. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. ACM SIGPLAN Notices 51(1), 357–370 (2016)

25. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. Laboratory for Computer Science, Massachusetts Institute of Technology (1988)

26. Milner, R.: A calculus of communicating systems. Springer (1980)

27. Milner, R.: Communicating and mobile systems: the pi calculus. Cambridge university press (1999)

28. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf

29. Paraskevopoulou, Z., Li, J.M., Appel, A.W.: Compositional optimizations for cer-ticoq. Proceedings of the ACM on Programming Languages 5(1ICFP), 1–30 (2021)

30. Pilikiewicz, A., Potier, F.: The essence of monotonic state. In: Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation. pp. 73–86 (2011)

31. Princehouse, L., Chenchu, R., Jiang, Z., Birman, K.P., Foster, N., Soulé, R.: Mica: A compositional architecture for gossip protocols. In: European Conference on Object-Oriented Programming. pp. 644–669. Springer (2014)

32. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proceedings of the ACM on Programming Languages 2(POPL), 1–30 (2017)

33. Shapiro, E.: The blocklace: A partially-ordered generalization of the blockchain and its grassroots consensus protocol stack. To appear. Also accessible as Section 4 of an earlier version of this paper: https://arxiv.org/abs/2112.13650v8 (2022)

34. Shapiro, E.: Sovereign cryptocurrencies: Foundation for grassroots cryptoeconomy. arXiv preprint arXiv:2202.05619 (2022)

35. Shostak, R., Pease, M., Lamport, L.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)

36. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 357–368 (2015)

37. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356 (2019)
A Proofs

Proof (of Observation [1]). Assume by way of contradiction that the live computation \( r \) is finite and its last state \( s \) is not final. Hence there is a correct transition \( t \) enabled on \( s \), and \( r \) violates both liveness requirements: First, that \( r \) has a nonempty suffix in which no correct transition is enabled, since \( t \) is enabled on every nonempty suffix of \( r \). Second, that every suffix of \( r \) includes an correct transition, since the suffix that include only \( s \) does not. Hence \( r \) is not live. A contradiction.

Proof (of Proposition [2]). We prove the proposition by way of contradiction. Assume that \( \sigma \) is locally safe but not safe. Hence, there is a computation \( r' \subseteq T' \) with an incorrect transition \( t \in \sigma(r') \setminus T \). Consider a prefix \( r'' \) of \( r' \) for which \( t \in \sigma(r'') \). This prefix violates local safety. A contradiction.

Assume that \( \sigma \) is productive but not live. Then there is a set of transitions \( L \in \lambda \) and a computation \( r' \subseteq T' \) for which \( \sigma(r') \) is not live wrt \( L \). This means that in every nonempty suffix of \( \sigma(r') \) \( L \) is enabled, and there is a suffix of \( \sigma(r') \) that does not include an \( L \) transition. This violates both alternative conditions for \( \sigma \) being productive: that \( r' \) has a nonempty suffix \( r'' \) such that \( L \) is not enabled in \( \sigma(r'') \), and that every suffix \( r'' \) of \( r' \) activates \( L \). A contradiction.

Assume that \( \sigma \) is locally complete but not complete. Then there is a run \( r \subseteq T \) for which there is no run \( r' \subseteq T' \) such that \( \sigma(r') = r \). Then there must be a prefix \( \bar{r} \prec r \) of \( r \) for which for no run \( r' \subseteq T' \), \( \bar{r} \prec \sigma(r') \). Thus \( \bar{r} \) violates local completeness, a contradiction. This completes the proof.

Proof (of Proposition [3]). Given a generic transition system \( G = (S, s_0, TG) \) over \( S \), we define a subset \( SC_1 \) of \( SC \) and a mapping \( \sigma_1 \) from \( SC_1 \) to \( G \) that together implement \( G \). The transition system \( SC_1 = (S^+, s_0, TSC_1) \) has the transition \( x \cdot s \rightarrow x \cdot s \cdot s' \in TSC_1 \) for every \( x \in S^+ \) and every transition \( s \rightarrow s' \in TG \). The mapping \( \sigma_1 : S^+ \rightarrow S \) takes the last element of its input sequence, namely \( \sigma_1(x \cdot s) := s \).

To prove that \( \sigma_1 \) is correct we have to show that \( \sigma_1 \) is:

1. **Locally Safe**: \( s_0 \xrightarrow{*} y \rightarrow y' \subseteq TSC_1 \) implies that \( s_0 \xrightarrow{*} x \xrightarrow{*} x' \subseteq TG \) for \( x = \sigma_1(y) \) and \( x' = \sigma_1(y') \) in \( S \).

   Let \( y = s_0 \cdot s_1 \cdot \ldots \cdot s_k, y' = s_0 \cdot s_1 \cdot \ldots \cdot s_i \cdot s_{i+1}, i \leq k \). For each transition \( s_0 \cdot s_1 \cdot \ldots \cdot s_i \rightarrow s_0 \cdot s_1 \cdot \ldots \cdot s_{i+1} \), the transition \( s_i \rightarrow s_{i+1} \in TG \) by definition of \( TSC_1 \). Hence \( s_0 \xrightarrow{*} x \xrightarrow{*} x' \subseteq TG \), satisfying the safety condition.

2. **Productive**: \( TSC_1 \) is the only set in the liveness condition, and any \( TSC_1 \) transition from any state of \( SC_1 \) activates \( TG \).

3. **Locally Complete**: \( s_0 \xrightarrow{*} x \rightarrow x' \subseteq TG \) implies that there are \( y, y' \in S^+ \) such that \( x = \sigma_1(y), x' = \sigma_1(y') \), and \( s_0 \xrightarrow{*} y \xrightarrow{*} y' \subseteq TSC_1 \).

   Let \( x = s_k, x' = s_{k+1}, k \geq 1 \), and \( s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow s_{k+1} \in TG \). Then \( y = s_0 \cdot s_1 \cdot \ldots \cdot s_k \) and \( y' = y \cdot s_{k+1} \) satisfy the completeness condition.

This completes the proof.

Proof (of Proposition [3]). Assume transition systems \( TS_1 = (S_1, s_1, T_1, \lambda_1) \), \( TS_2 = (S_2, s_2, T_2, \lambda_2) \), \( TS_3 = (S_3, s_3, T_3, \lambda_3) \) and implementations \( \sigma_{21} : S_2 \rightarrow S_1 \) and \( \sigma_{32} : S_3 \rightarrow S_2 \), and let \( \sigma_{31} := \sigma_{21} \circ \sigma_{32} \).
Assume that $\sigma_{32}$ and $\sigma_{21}$ are safe. Let $r \subseteq T3$ be a safe $TS3$ run. Then $\sigma_{32}(r)$ is a safe $TS2$ run by the safety of $\sigma_{32}$, and hence $\sigma_{21}(\sigma_{32}(r))$ is a safe run by the safety of $\sigma_{21}$. Hence $\sigma_{31}$ is safe.

Assume that $\sigma_{32}$ and $\sigma_{21}$ are live. Let $r \subseteq T3$ be a live $TS3$ run. Then $\sigma_{32}(r)$ is a live $TS2$ run by the liveness of $\sigma_{32}$, and hence $\sigma_{21}(\sigma_{32}(r))$ is a live run by the liveness of $\sigma_{21}$. Hence $\sigma_{31}$ is live.

A safe and live run is correct, hence if $\sigma_{32}$ and $\sigma_{21}$ are correct then so is $\sigma_{31}$.

Assume that $\sigma_{32}$ and $\sigma_{21}$ are complete. Let $r1 \subseteq T1$ be a correct $TS1$ run. By completeness of $\sigma_{21}$ there is a correct $TS2$ run $r2 \subseteq T2$ such that $\sigma_{21}(r2) = r1$. By completeness of $\sigma_{32}$ there is a correct $TS3$ run $r3 \subseteq T3$ such that $\sigma_{32}(r3) = r2$. Hence $\sigma_{31}(r3) = r1$, establishing the completeness of $\sigma_{31}$.

This completes the proof.

Proof (of Theorem 1). According to Proposition 1, to show that a productive $\sigma$ is correct and complete it is sufficient to show that $\sigma$ is:

1. **Locally Safe**: $s_0 \xrightarrow{'} y \rightarrow y' \subseteq T'$ implies that $s_0 \rightarrow x \rightarrow x' \subseteq T$ for $x = \sigma(y)$ and $x' = \sigma(y')$ in $S$.

   By monotonicity of $TS'$ it follows that $s_0 \leq y \prec y'$; by the Up condition on $\sigma$, it follows that $s_0 \leq \sigma(y) \leq \sigma(y')$; by assumption that $TS$ is monotonically-complete it follows that $s_0 \xrightarrow{'} x \rightarrow x' \subseteq T$ for $x = \sigma(y)$ and $x' = \sigma(y')$ in $S$. Hence $\sigma$ is safe.

2. **Locally Complete**: $s_0 \xrightarrow{'} x \rightarrow x' \subseteq T$ implies $s_0' \xrightarrow{'} y \rightarrow y' \subseteq T'$ for some $y, y' \in S'$ such that $x = \sigma(y)$ and $x' = \sigma(y')$.

   Let $s_0 \rightarrow x \rightarrow x' \subseteq T$. By monotonicity of $TS$, $s_0 \leq x \leq x'$; by the Down condition on $\sigma$, there are $y, y' \in S'$ such that $x = \sigma(y), x' = \sigma(y')$, and $y \leq y'$; by assumption that $TS'$ is monotonically-complete, $s_0' \xrightarrow{'} y \rightarrow y' \subseteq T'$. Hence $\sigma$ is complete.

This completes the proof of correctness and completeness of $\sigma$.

Proof (of Observation 2). As $TS'$ is monotonically-complete, it has a computation $\hat{\sigma}(x) \xrightarrow{'} \hat{\sigma}(x') \subseteq T'$ that satisfies the Down condition.

**Observation 3** $SC$ is monotonically-complete wrt $\preceq$.

Proof (of Observation 3). SC is monotonic wrt $\preceq$ since every transition increases its sequence. Given two sequences $x, x' \in S^*$ such that $x \sim x'$, let $x' = x \cdot s_1 \ldots s_k$, for some $k \geq 1$. Then $x \rightarrow x'$ via the sequence of transitions $x \rightarrow x \cdot s_1 \rightarrow \ldots \rightarrow x \cdot s_1 \ldots s_k$. Hence SC is monotonically-complete.

**Observation 4** $LC$ is monotonically-complete wrt $\preceq$.

The proof is similar to the proof of Observation 3.

**Observation 5 (LC Configurations are Consistent)** An $n$-chain configuration $c$ is consistent if every two chains in $c$ are consistent. Let $r$ be a run of $LC$ and $c \in r$ a configuration. Then $c$ is consistent.
Proof (of Observation 3). The proof is by induction on the index \( k \) of a configuration in \( r \). All empty sequences of the initial configuration of \( r \) are pairwise consistent. Assume the \( k^{th} \) configuration \( c \) of \( r \) is consistent and consider the next \( r \) transition \( c \rightarrow c' \in \text{TLC} \). The transition adds an element \( s \) to one sequence \( x \in c \) that either is a longest sequence, or \( x \cdot s \) is consistent with another longer sequence \( x' \in c \). As all sequences in \( c \) are pairwise consistent by assumption, then they are also consistent with \( x \cdot s \) by construction. Hence all sequences of \( c' \) are pairwise consistent and hence \( c' \) is consistent.

Hence the following implementation of \( SC \) by \( LC \) is well-defined.

**Proposition 12.** \( \sigma_2 \) is order-preserving wrt the prefix relation \( \preceq \) over consistent n-chain configurations and is productive.

**Proof (of Proposition 12).** To show that \( \sigma_2 \) is order-preserving it is sufficient to show (Proposition 1) that:

1. **Up condition:** \( y \preceq y' \) for \( y, y' \in S1 \) implies that \( \sigma_2(y) \preceq \sigma_2(y') \) and \( y \prec y' \) for \( y, y' \in S1 \) implies that \( \sigma_2(y) \prec \sigma_2(y') \)

2. **Down condition:** \( s_0 \xrightarrow{\sigma} x \in T0, x \preceq x' \) implies that there are \( y, y' \in S1 \) such that \( x = \sigma_2(y), x' = \sigma_2(y') \), \( c_0 \xrightarrow{\sigma} y \subseteq T1 \) and \( y \preceq y' \).

Regarding the Up condition, assume that \( y \preceq y' \) are consistent and that \( y_p \) is the unique longest chain in \( y' \). Then \( \sigma_2(y) = y_p \preceq y' = \sigma_2(y') \), and if \( y \prec y' \) \( \sigma_2(y) = y_p \prec y' = \sigma_2(y') \).

Regarding the Down condition, define \( y_p := x, y'_p := x' \), and \( y_q := y'_q := A \) for every \( q \neq p \in P \). Then \( x = y_p = \sigma_2(y), x' = y'_p = \sigma_2(y'), c_0 \xrightarrow{\sigma} y \subseteq T1 \) by the same transitions that lead from \( s0 \) to \( x \), and \( y \preceq y' \) by construction.

To see that \( \sigma_2 \) is productive, note that every \( LC \) transition extends one of the chains in a configuration. Hence, after a finite number of transitions, the next \( LC \) chain will extend the longest chain in the configuration, and activate \( SC \).

**Proof (of Proposition 3).** Assume \( TS1, TS2, TS1', TS2' \) and \( \sigma \) as in the Proposition and that \( y \rightarrow y' \subseteq T2 \& \sigma(y) \in S1' \) implies that \( \sigma(y') \in S1' \). Define \( \sigma' : C2' \rightarrow S1' \) to be the restriction of \( \sigma \) to \( C2' \). We have to show that \( \sigma' \) is correct. To do that, it is sufficient to show that \( \sigma' \) is:

1. **Locally Safe:** \( s2 \xrightarrow{\sigma'} y \rightarrow y' \subseteq T2' \) implies that \( s1 \xrightarrow{\sigma} x \rightarrow x' \subseteq T1' \) for \( x = \sigma'(y) \) and \( x' = \sigma'(y') \) in \( S1 \).

   This follows from the safety of \( \sigma \), \( S1' \subseteq S1 \) and the assumption that \( y \rightarrow y' \subseteq T2' \& \sigma(y) \in S1' \) implies that \( \sigma(y') \in S1' \).

2. **Productive:** if any suffix of any infinite correct computation of \( TS2' \) activates \( T1' \).

   By monotonicity of \( TS2' \), any infinite correct computation \( r \) of \( T2' \) from \( x' \) has a transition \( t \) that is strictly increasing, and hence by \( \sigma \) satisfying the Up condition, the transition \( t \) activates \( T1' \).

3. **Locally Complete:** \( s1 \xrightarrow{\sigma} x \rightarrow x' \subseteq T1' \), implies that there are \( y, y' \in C2' \) such that \( x = \sigma'(y), x' = \sigma'(y') \), and \( s2 \xrightarrow{\sigma'} y \rightarrow y' \subseteq T2' \).
Fig. 2. Some Steps in the Proof of Proposition 6 (with an example in yellow): While $TS_2$ (a subset of Messaging) implements $TS_1$ (Dissemination), which in turn implements $TS_0$ (Reliable Broadcast), $TS'_1$ (a subset of Dissemination) is sufficient to implement $TS_0$. Hence, it may be more efficient to employ the subset $TS'_2$ (of Messaging) instead of the full $TS_2$ for the composed implementation of $TS_0$. Still, $TS_1$ (Dissemination) may have other applications (e.g. grassroots social network, sovereign cryptocurrencies [34]), hence it would be useful to implement the entire $TS_1$, but then use only the subset $TS'_2$ of $TS_2$ in the composed implementation of $TS_0$. Proposition 6 provides conditions that enable that.

By completeness of $\sigma$, there are $y, y' \in C_2$ such that $x = \sigma(y)$, $x' = \sigma(y')$, and $s_2 \xrightarrow{\sigma} y \xrightarrow{\sigma} y' \subseteq T_2$. By definition of $C_2'$ as the domain of $\sigma$, $y, y' \in C_2'$. As $y \xrightarrow{\sigma_2} y' \subseteq T_2$, then $y \preceq_2 y'$. By assumption that $TS'_2$ is monotonically-complete, there is a computation $s_2 \xrightarrow{\sigma_2} y \xrightarrow{\sigma_2} y' \subseteq T_2'$.

This completes the proof. □

Proof (of Corollary 7). Given a generic transition system $G$ over $S$, a correct implementation $\sigma_1$ of $G$ by SC exists according to Proposition 2. The implementation $\sigma_2$ of SC by LC is correct according to Proposition 5. Then, Propositions 3 and 5 ensure that even though a subset SC1 of SC was used in implementing $G$, the result of the composition $\sigma_{21} := \sigma_2 \circ \sigma_1$ is a correct implementation of $G$ by LC.

Proof (outline of Proposition 7). The proof is similar to that of Proposition 2. Given a generic shared-memory multiagent transition system $GS = ((S \times
Given $LCC = (P, (S \times P)^*, c_0, TLCC)$ and $ABD = (P, B, c_0, TABD, \lambda)$, show that $ABD$ and $LCC$ are monotonically-complete wrt $\preceq$ and $\succeq$ (Propositions ??, ??), respectively. Define $\sigma_3$ for each configuration $c \in C$ by $\sigma_3(c)_p := \sigma_3'(c_p)$, where $\sigma_3'$ is defined as follows. Given a set of blocks $B$, let $\text{sort}(B)$ be the sequence obtained by sorting $B$ lexicographically, removing $\bot$ blocks and then possibly truncating the output sequence, where blocks $(p, i, s)$ are sorted first according to the index of the block $i \in \mathbb{N}$ and then
according to the agent \( p \in P \), and truncation occurs at the first gap if there is one, namely at the first index \( i \) for which the next agent in order is \( p \) but there is no block \((p, i, s) \in B \) for any \( s \in S \cup \{\perp\} \). Proposition 13 argues that \( \sigma_3 \) is order-preserving, which allows the application of Theorem 1 and completes the proof. \( \square \)

Namely \( \sigma_3 \) performs for each agent \( p \) a ‘round robin’ complete total ordering of the set of block of its local state \( c_p \), removing undefined elements along the way, until some next block missing from \( c_p \) prevents the completion of the total order.

First, we observe that for every configuration \( c \in r \) in an ABD run \( r \), the sequences in \( \sigma_3(c) \) are consistent. Note that if \( x \preceq y \) and \( x' \preceq y \) then \( x \) and \( x' \) are consistent.

**Observation 6 (Consistency of \( \sigma_3 \))** Let \( r \) be a correct run of ABD. Then for every configuration \( c \in r \), the chains of \( \sigma(c) \) are mutually consistent.

**Proof (of Observation 6).** First, note that in a correct run \( r \), every configuration \( c \in r \) is equivocation free. Also note that \( \sigma'_3 \) is monotonic wrt \( \subseteq \) and \( \preceq \), namely if \( B \subseteq B' \) and both \( B, B' \) are equivocation free, then \( \sigma'_3(B) \preceq \sigma'_3(B') \). For a configuration \( c \in r \), \( c_p \subseteq B(c) \) for every \( p \in P \) and hence \( \sigma'_3(c_p) \preceq \sigma'_3(B(c)) \), and therefore every two sequences \( \sigma'_3(c_p), \sigma'_3(c_q) \) are consistent. \( \square \)

Next, we show that \( \sigma_3 \) is order preserving.

**Proposition 13.** \( \sigma_3 \) is order preserving wrt \( \subseteq \) and \( \preceq \).

**Proof (of Proposition 13).** According to definition 10, we have to prove two conditions. For the Up condition, we it is easy to see from the definition of \( \sigma_3 \) that \( c'_1 \subseteq c'_2 \) for \( c'_1, c'_2 \in B' \) implies that \( \sigma(c'_1) \preceq \sigma(c'_2) \), as the output sequence of the sort procedure can only increase if its input set increases.

For the Down condition, we construct an ABD representative configuration for a LCC configuration so that if the \( i \)th element of the LCC longest chain is \((p, s)\), then the ABD configuration has the block \((p, i, s)\), as well as the blocks \((q, i, \perp)\) for every other agents \( q \neq p \). Specifically, given a LCC configuration \( c \) with a longest chain \( c_1 = (s_0, p_0), (s_1, p_1), \ldots, (s_k, p_k) \) for some \( l \in P \), we define the representative ABD configuration \( c' \) as follows. First, let \( B \) be the following set of blocks \( B \). For each \( i \in [k] \) \( B \) has the block \((p_i, i, s_i)\) and the blocks \((q, i, \perp)\) for every \( q \neq p \). Clearly \( \sigma_3'(B) = c_l \) by construction. Let \( B' := \{(p, j, s) \in B : j \leq i\} \). It is easy to see that for each \( i \in [k] \), \( \sigma_3'(B') \) is the \( i \)-prefix of the longest chain \( c_l \). Then for each \( p \in P \), where \( |c_p| = i \), we define \( c'_p := B' \). Hence, \( \sigma_3'(c'_p) = c_p \) for every \( p \in P \) and thus \( \sigma_3(c') = \sigma(c) \). \( \square \)

**Proof (of Theorem 2).** Assume transition systems and implementations as in the theorem statement. As the composition of live implementations is live, and the assumption is that the runs with safety faults are live, we only argue for safety and conclude correctness.
Fig. 3. Some Steps in the Proof of Theorem 3. \( \sigma_{32} \) is resilient to \( F_3 \) and maps \( F_3' \) to \( F_2 \). \( \sigma_{21} \) is resilient to \( F_2 \). As a result, \( \sigma_{31} := \sigma_{21} \circ \sigma_{32} \) is resilient to \( F_3 \cup F_3' \).

1. Assume that \( \sigma_{32} \) is resilient to \( F_3 \subseteq S_3^2 \setminus T_3 \). We argue that \( \sigma_{31} \) is resilient to \( F_3 \). Then for any \( TS_3 \) run \( r \subseteq T_3 \cup F_3 \), the run \( \sigma_{31}(r) \in TS_1 \) is correct, namely \( \sigma_{31}(r) \in T_1 \), since \( \sigma_{32} \) is \( F_3 \)-resilient by assumption, and hence \( r' = \sigma_{32}(r) \) is correct, and \( \sigma_{21} \) is correct by assumption, and hence \( \sigma_{21}(r) \) is correct, namely \( \sigma_{31}(r) = \sigma_{21} \circ \sigma_{32}(r) \in T_1 \).

2. Assume \( \sigma_{21} \) is resilient to \( F_2 \subseteq S_2^2 \setminus T_2 \), and \( F_3 \subseteq S_3^2 \setminus T_3 \) satisfies \( \sigma_{32}(F_3) \subseteq F_2 \). We argue that \( \sigma_{31} \) is resilient to \( F_3 \). For any \( TS_3 \) run \( r \subseteq T_3 \cup F_3 \), the run \( \sigma_{32}(r) \in TS_2 \cup F_2 \) by assumption. As \( \sigma_{21} \) is \( F_2 \)-resilient by assumption, the run \( \sigma_{21} \circ \sigma_{32}(r) = \sigma_{31}(r) \) is correct.

3. Assume that \( \sigma_{21} \) is \( F_2 \)-resilient, \( \sigma_{32} \) is \( F_3 \)-resilient, \( F_3' \subseteq S_3^2 \setminus T_3 \), and \( \sigma_{32}(F_3') \subseteq F_2 \). We argue that \( \sigma_{31} \) is resilient to \( F_3 \cup F_3' \). For any \( TS_3 \) run \( r \subseteq T_3 \cup F_3 \cup F_3' \), the run \( \sigma_{32}(r) \in TS_2 \cup F_2 \) by assumption. As \( \sigma_{21} \) is \( F_2 \)-resilient by assumption, the run \( \sigma_{21} \circ \sigma_{32}(r) = \sigma_{31}(r) \) is correct. \( \square \)

Proof (of Proposition 10). If in configuration \( c \) there is a block \( b \) known by \( q \) but not by \( p \), both correct, then this holds in every subsequent configuration unless \( p \) receives \( b \). Hence, due to liveness of \( p \)-Receives-\( b \), either the \( p \)-Receives-\( b \) from \( q \) transition is eventually taken, or \( p \) receives \( b \) through a \( p \)-Receives transition from another agent. In either case, \( p \) eventually receives \( b \). \( \square \)

Next, we provide an operational characterization of grassroots via the notion of interleaving.

**Definition 28 (Joint Configuration, Interleaving).** Let \( F \) be a family of distributed transition systems, \( P_1, P_2 \subseteq \Pi \), \( P_1 \cap P_2 = \emptyset \). Then a joint config-
Multiagent Transition Systems with Faults

**Fig. 4.** Some Steps in the Proof of Theorem 5

**uration** $c$ of $c_1 \in C(P1)$ and $c_2 \in C(P2)$, denoted by $(c_1, c_2)$, is the configuration over $P1 \cup P2$ and $S(P1) \cup S(P2)$ satisfying $c_1 = c/P1$ and $c_2 = c/P2$. Let $r1 = c_{01} \rightarrow c_{11} \rightarrow c_{12} \rightarrow \ldots$ be a run of $TS(P1)$, $r2 = c_{02} \rightarrow c_{21} \rightarrow c_{22} \rightarrow \ldots$ a run of $TS(P2)$. Then an **interleaving** $r$ of $r1$ and $r2$ is a sequence of joint configurations $r = c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow \ldots$ over $P1 \cup P2$ and $S(P1) \cup S(P2)$ satisfying:

1. $c_0 = (c_{01}, c_{02})$
2. For all $i \geq 0$, if $c_i = (c_{1j}, c_{2k})$ then $c_{i+1} = (c_{1j'}, c_{2k'})$ where either $j' = j + 1$ and $k' = k$ or $j' = j$ and $k' = k + 1$.

Note that for $c_i = (c_{1j}, c_{2k})$, $i = j + k$ for all $i \geq 0$.

**Theorem 4 (Operational Characterization of Grassroots).** Let $F$ be a family of distributed transition systems with no redundant configurations. Then $F$ is grassroots iff for every $\emptyset \subset P1, P2 \subset \Pi$ such that $P1 \cap P2 = \emptyset$, the following holds:

- For every two runs $r1$ of $TS(P1) \in F$ and $r2$ of $TS(P2) \in F$, every interleaving $r = c_0 \rightarrow c_1 \rightarrow \ldots$ of $r1$ and $r2$ is a run of $TS(P1 \cup P1) \in F$, and

- There is a run $r$ of $TS(P1 \cup P1) \in F$ that is not an interleaving of any two runs $r1$ of $TS(P1) \in F$ and $r2$ of $TS(P2) \in F$.

Definition [30] of interactivity is algebraic. The following Proposition 14 provides an alternative operational characterization of interactivity via interleavings of runs of its component transition systems, defined next.

**Proposition 14 (Operational Characterization of Interactivity).** Let $F$ be a family of distributed transition systems. Then $F$ is interactive iff for every
Since assume that the condition holds and that Proof (of Proposition 14).

To prove the ‘if’ direction by way of contradiction, there are two runs \( r \in TS(P1) \in F \) and any run of \( TS(P2) \in F \).

Proof (outline of Theorem 4). The Theorem follows from the following Propositions 14 and 15 and from Observation ??.

Definition 29 (Subsidiary). A family of distributed transition systems \( F \) upholds subsidiarity if for every \( \emptyset \subset P1, P2 \subset \Pi \) such that \( P1 \cap P2 = \emptyset \), the following holds:

\[
TS(P1) \cup TS(P2) \subseteq TS(P1 \cup P2)
\]

Definition 30 (Interactive). A family of distributed transition systems \( F \) is interactive if for every \( \emptyset \subset P1, P2 \subset \Pi \) such that \( P1 \cap P2 = \emptyset \), the following holds:

\[
TS(P1 \cup P2) \not\subseteq TS(P1) \cup TS(P2)
\]

We say that a configuration in a transition system is redundant if it is not reachable by a run of the transition system.

Proposition 15 (Operational Characterization of Subsidiary). Let \( F \) be a family of distributed transition systems with no redundant configurations. Then \( F \) upholds subsidiarity if for every \( P1, P2 \subset \Pi \) such that \( P1 \cap P2 = \emptyset \), and every two runs \( r1 \) of \( TS(P1) \in F \) and \( r2 \) of \( TS(P2) \in F \), every interleaving \( r = c_0 \rightarrow c_1 \rightarrow \ldots \) of \( r1 \) and \( r2 \) is a run of \( TS(P1 \cup P1) \in F \).

Proof (of Proposition 15). To prove the ‘if’ direction by way of contradiction, assume that the condition holds and that \( F \) is not interactive. Then there are \( P1, P2 \) as stated for which every member of \( TS(P1) \cup TS(P2) \) is a member of \( TS(P1 \cup P2) \).

For every run \( r \) of \( TS(P1 \cup P1) \in F \) there are two runs \( r1 \) of \( TS(P1) \in F \) and \( r2 \) of \( TS(P2) \in F \) for which \( r \) is their interleaving. By Definition 28, \( TS(P1 \cup P1) \) includes all \( p \)-transitions \( c \rightarrow c' \in T \), \( p \in P \), satisfying \( p \in P1 \land (c \rightarrow c'/P1) \in T1 \land c/P2 = c'/P2 \) or \( p \in P2 \land (c/P2 \rightarrow c'/P2) \in T2 \land c/P1 = c'/P1 \).

This satisfies the condition of Definition 24 of union, namely \( TS(P1) \cup TS(P1) \subseteq TS(P1 \cup P2) \). A contradiction.

To prove the ‘only if’ direction by way of contradiction, assume that \( F \) is interactive but that the condition does not hold. Hence for every run \( r \) of \( TS(P1 \cup P1) \in F \) there are two runs \( r1 \) of \( TS(P1) \in F \) and \( r2 \) of \( TS(P2) \in F \) for which \( r \) is their interleaving. By Definition 24 of union, this implies that \( TS(P1) \cup TS(P2) \subseteq TS(P1 \cup P2) \), a contradiction. This completes the proof.

Proof (of Proposition 15). To prove the ‘if’ direction by way of contradiction, assume that the condition holds and that \( F \) does not uphold subsidiarity. Then there are \( P1, P2 \) satisfying the condition, and two transitions \( c1 \rightarrow c1' \in T(P1) \), \( c2 \rightarrow c2' \in T(P2) \), such that the \( p \)-transition \( (c1, c2) \rightarrow (c1', c2') \notin T(P1 \cup P2) \).

Since \( F \) has no redundant configurations, there are runs \( r1 = c01 \rightarrow c1' \in T(P1) \) and \( r2 = c02 \rightarrow c2' \in T(P1) \), and consider their interleaving \( r \), in which the last transition is \( (c1, c2) \rightarrow (c1', c2') \). By construction, the last transition of the run \( r \) is \( (c1, c2) \rightarrow (c1', c2') \), implying that it is in \( T(P1 \cup P2) \), a contradiction.
To prove the ‘only if’ direction by way of contradiction, assume that $F$ upholds subsidiarity but that the condition does not hold. Hence there are runs $r_1$ and $r_2$ of $TS(P_1)$ and $TS(P_2)$, with an interleaving $r$ that is not a run of $TS(P_1 \cup P_2)$. Consider the maximal prefix $\hat{r} = (c_{10}, c_{20}) \xrightarrow{r} (c_1, c_2)$ of $r$ that is a run of $TS(P_1 \cup P_2)$. Consider the $r$-transition $(c_1, c_2) \rightarrow (c_1', c_2')$ that extends $\hat{r}$. By definition of union of transition systems, the transition $(c_1, c_2) \rightarrow (c_1', c_2')$ is a transition of $TS(P_1) \cup TS(P_1)$. By the assumption that $F$ is grassroots, it follows that $(c_1, c_2) \rightarrow (c_1', c_2') \in TS(P_1 \cup P_2)$, and therefore $\hat{r}$ is not maximal as constructed, a contradiction. $\square$

**Fig. 5.** Some Steps in the Proof of Theorem 3

Proof (of Theorem 3). Let $F$ be a non-interfering family of distributed transition systems that is monotonic and asynchronous wrt a partial order $\preceq$, $P_1, P_2 \subseteq \Pi$ such that $P_1 \cap P_2 = \emptyset$, $r_1$ a run of $TS(P_1) \in F$, $r_2$ a run of $TS(P_2) \in F$, $r = c_0 \rightarrow c_1 \rightarrow \ldots$ an interleaving of $r_1$ and $r_2$. We argue that $r$ is a run of $TS(P_1 \cup P_1)$ (See Figure 5). Consider any $p$-transition $(c_1, c_2) \rightarrow (c_1', c_2') \in r$. Wlog assume that $p \in P_1$ (else $p \in P_2$ and the symmetric argument applies) and let $\hat{c}, \hat{c}'$ be the $TS(P_1 \cup P_1)$ configurations for which $\hat{c}/P_1 = c/P_1, \hat{c}'/P_1 = c'/P_1,$ and $\hat{c}/P_2 = \hat{c}'/P_2 = c_0/P_2$. Since $(c_1 \rightarrow c_1') \in T(P_1)$, $F$ is non-interfering, and in $\hat{c}, \hat{c}'$ members of $P_2$ stay in their initial state, then by Definition 27 it follows that the $p$-transition $\hat{c} \rightarrow \hat{c}' \in T(P_1 \cup P_2)$. Since $c_p = c_p$ by construction, $c_0(P_2) \preceq c_2$ by monotonicity of $TS(P_2)$, $c_0(P_1 \cup P_2) \preceq c$ by monotonicity of $TS(P_1 \cup P_2)$ and the assumption that $\preceq$ is preserved under projection, it follows that $c \rightarrow c' \in T(P_1 \cup P_2)$ by the assumption that it is asynchronous.
wrt $\preceq$ (Definition 12). As $c \to c'$ is a generic transition of $r$, it follows that $r \subseteq T(P_1 \cup P_2)$, satisfying the condition of Proposition 15, implying that $F$ upholds subsidiarity. Together with the assumption that $F$ is interactive, we conclude that $F$ is grassroots.

**Definition 31 (Correct and Local Implementation among Protocols).**
Let $F, F'$ be protocols over $S, S'$, respectively. A function $\sigma$ that provides a mapping $\sigma : S'^P \to S^P$ for every $P \subseteq \Pi$ is a correct implementation of $F$ by $F'$ if for every $P \subseteq \Pi$, $\sigma$ is a correct implementation of $TS(P)$ by $TS'(P)$, and it is local if $\sigma(c)_p = \sigma(c_p)$ for every configuration $c \in S'^P$, $P \subseteq \Pi$.

Namely, $\sigma$ is local if it is defined for each local state independently of other local states.

**Example 13 (The implementation of LCC by ABD is not local).** The reason why $\sigma_3$ that implements LCC by ABD is not local is that it depends on knowledge of the set of agents $P$ – the sort procedure cannot proceed beyond index $i$ if an $i$-indexed $p$-block is missing, for some $p \in P$.

In line with the example above, the following theorem shatters the hope of a non-grassroots protocol to have a correct local implementation by a grassroots, and can be used to show that a protocol is grassroots by presenting a correct local implementation of it by another grassroots protocol.

**Theorem 5 (Grassroots Implementation).** A protocol that has a correct local implementation by a grassroots protocol is grassroots.

**Proof (of Theorem 5).** We apply Theorem 4 in both directions. See Figure 4 roman numerals (i)-(v) in the proof refer to the Figure. Let $F, F'$ be families of multiagent transition systems over $S, S'$, respectively, $\sigma$ a correct implementation of $F$ by $F'$ and assume that $F'$ is grassroots. Let $P_1, P_2 \subseteq \Pi$ such that $P_1 \cap P_2 = \emptyset$, $r_1 \subseteq T(P_1)$ a run of $TS(P_1) \in F$, $r_2 \in T(P_2)$ a run of $TS(P_2) \in F$, (i) and $r = c_0 \to c_1 \to \ldots$ an interleaving of $r_1$ and $r_2$ (Def. 28). Since $\sigma$ is a local and correct implementation of $TS(P_1)$ by $TS'(P_1)$, there is a run $r_1' \in T'(P_1)$ of $TS'(P_1)$, such that $\sigma(r_1') = r_1$; the same holds for $P_2$, $r_2'$ and $r_2$. (ii) Let $r'$ be the interleaving of $r_1'$ and $r_2'$ for which $\sigma(r') = r$; such an interleaving can be constructed iteratively, with each $p$-transition of $TS(P_1)$ realized by the implementing computation of $TS'(P_1)$ for $p \in P_1$, and similarly for $p \in P_2$. (iii) Since $F'$ is grassroots by assumption, then by Proposition 4, the 'only if' direction, $r' \in T'(P_1 \cup P_1)$ is a run of $TS'(P_1 \cup P_1) \in F$. (iv) By assumption, $\sigma$ is a correct implementation of $TS(P_1 \cup P_2)$ by $TS'(P_1 \cup P_2)$. Hence $\sigma(r') = r \in T(P_1 \cup P_1)$ is a correct computation of $TS'(P_1 \cup P_1)$, (v) satisfying the conditions for the 'if' direction of Proposition 4 and concluding the $F$ is grassroots. □