

Extended External Validity

Ritual

April 20, 2026

Abstract

Proposers in current blockchain instantiations have extreme discretionary (and unilateral) power over the set of transactions in a block and their ordering. The set of blocks that a proposer can build is constrained in order to provide guarantees on *inclusion*, *exclusion*, and *sequencing* to applications and users. Each guarantee is enforced as a conjunct of the block validity function via extended external validity [6]. External validity, introduced by Cachin, Kursawe, and Shoup [6], is a property of SMR protocols that allows the validity condition on proposed values to reference application-specific predicates. We prove that predictable validity (the proposer can compute the post-confirmation state) is necessary for extended external validity with current-state predicates, and characterize the censorship-resistance/liveness tradeoff: each active guarantee adds evaluation work to block construction and can only shrink the set of valid blocks, and conflicting guarantees can leave no valid block.

1 Introduction

Symphony [35] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

The set of blocks that a proposer can build is constrained in order to provide guarantees on *inclusion*, *exclusion*, and *sequencing* to applications and users.

Block proposers in existing protocols exercise several distinct powers that are often treated as atomic: which transactions to include, which to exclude, how to order them, and when to propose. Recent Ethereum discussions have explored unbundling proposer or validator roles at a high level [27, 28]. However, most concrete protocol proposals still intervene on individual powers in isolation and for particular objectives, rather than offering a general framework in which these powers are explicitly separated and can be independently constrained or reassigned. For example, Proposer-Builder Separation [5] shifts block construction to a builder, but still transfers the relevant bundle largely as a unit. Protocol-Enforced Proposer Commitments (PEPC) push further in this direction by allowing proposers to make more general commitments about block contents that are enforced by the protocol, rather than merely outsourcing block construction wholesale [29]. Inclusion lists such as EIP-7547 [10] constrain omission discretion, order-fairness protocols [14] constrain sequencing through a global ordering rule, while encrypted mempools [23] reduce the proposer’s informational advantage without redistributing decision authority. Symphony generalizes this design space by transferring the decisions about block contents to on-chain applications and the users interacting with them at the protocol level.

In a leader-based SMR protocol, a proposer has discretion over which valid block to submit for a given pre-state. Most protocols impose basic constraints on block validity, e.g. that the block is well-formed, all transaction signatures are valid, and the parent hash is correct. Outside of these basic checks, the proposer has full power to determine the contents of the block. Some existing protocols further constrain the block proposer’s discretion over the block. For example, in Ethereum, all included transactions must pay at least a base fee that is burned by the protocol

(EIP-1559) for a block to be considered valid. In this way, the protocol limits the ability of the proposer to include arbitrary transactions, even if the block has space remaining.

Some other existing modifications further limit the proposer’s abilities (both in- and out-of-protocol):

Mechanism	Action
Inclusion lists [10]	Enforce transaction inclusion in the next block
Hyperliquid [1]	Enforces cancel prioritization over take orders
MEV bundles [11]	Enforce atomicity of transaction bundles

Note that these constraints differ from the baseline validity checks on blocks in that they are specifically designed to provide additional features to applications and users. In other words, enforced restrictions of the proposer’s action space can be interpreted as guarantees that applications and users find economically valuable. Symphony supports a much broader class of such guarantees by streamlining their enforcement through consensus and allowing applications and users to register according to their needs.

External validity, introduced by Cachin, Kursawe, and Shoup [6], is a property of SMR protocols that allows the validity condition on proposed values to reference application-specific predicates. In standard BFT protocols, the validity condition checks structural properties: well-formed blocks, valid signatures, correct state root. External validity extends this with predicates that reference the current state.

This paper makes the following contributions:

- Each guarantee is enforced as a conjunct of the block validity function via extended external validity [6] (Section 3.1).
- We prove that predictable validity (the proposer can compute the post-confirmation state) is necessary for extended external validity with current-state predicates, and characterize the censorship-resistance/liveness tradeoff (Sections 2.2 and 3.4.3).

Section 2 establishes the formal framework: system model, predictable validity, and the guarantee families. Section 3 defines extended external validity and characterizes the CR/liveness tradeoff. Section 4 establishes liveness. Section 5 discusses limitations. Section 6 reviews related work. Section 7 concludes.

2 Formal Framework

2.1 System Model

We consider a distributed system of n stake-weighted replicas (validators), of which at most $f < n/3$ are Byzantine. The system operates under partial synchrony [9]: there exists a Global Stabilization Time (GST) and a message delay bound Δ , both unknown to the protocol, such that after GST all messages between correct replicas are delivered within Δ . Before GST, messages may be delayed arbitrarily but are not lost.

Each replica i holds stake $w_i > 0$ with $\sum_{i=1}^n w_i = W$. A set of replicas A constitutes a *quorum* if $\sum_{i \in A} w_i > 2W/3$. Byzantine replicas control at most $f_W < W/3$ total stake.

The system runs a *state machine replication (SMR)* protocol [22]. SMR (equivalently, atomic broadcast) is the task of maintaining a totally-ordered log of state transitions such that all correct replicas agree on the same sequence of transitions and apply them in the same order. Formally: given a deterministic state transition function $\text{STF} : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$, where $\mathcal{S}$ is the state space and $\mathcal{T}$ the set of transactions, SMR guarantees that (1) all correct replicas deliver the same sequence of transactions, and (2) the resulting state is uniquely determined by applying STF to the initial state in that order. We extend the standard SMR model with additional state components defined in Section 2.4.

2.2 Predictable Validity

Extended external validity (Section 3) requires that the block proposer and all correct validators agree on the state against which validity predicates are evaluated. When a proposer’s turn comes to propose a block, they can exactly compute the state resulting from confirmation of their block—that is *predictable validity*. We formalize this as a property of the underlying SMR protocol.

Definition 2.1 (Predictable Validity). *An SMR protocol provides predictable validity if the following holds: when a correct proposer L constructs a block B at time t , there exists a unique state S (the state L is building on) such that:*

1. *S is derived from the committed prefix of the ledger from L ’s view. (Other validators may lag behind L ’s view of the committed prefix; the key is that L builds on a well-defined prefix.)*
2. *L can compute S from its local information at time t . For every correct validator V that receives B , V can verify that S is the correct pre-state (derived from the committed prefix) and that B is valid when applied to S .*
3. *For any predicate $p : \mathcal{S} \rightarrow \{0, 1\}$ over the state space, $p(S)$ is well-defined and yields the same result for L and all correct validators.*

For single-leader protocols with linear commit (see the classification below), this property is a direct consequence of total-order state machine replication [22]: when a proposer is about to propose a block, they can exactly compute the state resulting from confirmation of their block—they know the contents they are building on. This is commonly called *predictable validity* in the SMR literature.

Proposition 2.2 (Predictable Validity Requirement for Extended External Validity). *Let Π be an SMR protocol. If Π does not provide predictable validity because a correct proposer L and a correct validator V can derive different states $S_L \neq S_V$ from their respective local information, then there exist a state predicate $p : \mathcal{S} \rightarrow \{0, 1\}$ used in the block validity function and executions in which a correct proposer L constructs a block B that L considers valid (because $p(S_L) = 1$ under L ’s view of the state) but a correct validator V considers invalid (because $p(S_V) = 0$ under V ’s view), or vice versa.*

Proof. Suppose, as in the statement, that there exist a correct proposer L at time t and a correct validator V at time $t' > t$ such that the states S_L and S_V derived from their respective local information differ: $S_L \neq S_V$. Since p is an arbitrary predicate, there exists a p that distinguishes S_L and S_V (e.g., $p(S) = \mathbf{1}[S = S_L]$). For this p , $p(S_L) = 1$ but $p(S_V) = 0$. If p is a conjunct of the block validity function, the proposer constructs a block that it believes is valid (it includes transactions mandated by p), but the validator rejects the block (because p is not satisfied under S_V). Hence correct replicas disagree on the validity of a block proposed by a correct proposer. $\square$

Classification of SMR protocols. We classify existing SMR protocols by this property:

Protocol family	Predictably valid?	Staleness
Single-leader, linear (CometBFT [4], HotStuff [26])	Yes	0
Single-leader, pipelined (MonadBFT, Simplex [17])	Yes	0
Leaderless, metastable (Avalanche [20])	Probabilistic	Convergence-dep.
DAG-based (Bullshark [24], Narwhal [8])	Over committed prefix	$\leq$ commit interval

Guarantee	Typical registrant	Enforcement
Inclusion	User	Block invalid if triggered tx absent
Exclusion	Protocol / User / App	Block invalid if excluded tx present
Ordering	Application	Block invalid if order violates $(\pi_C, \{\tau_i\})$

Table 1: Families of guarantees provided by Symphony.

The key parameter is *staleness* [2]: the gap between the latest state and the predictably valid state available at proposal time. Linear protocols have staleness 0: the proposer’s predecessor state is precisely the latest committed state. DAG-based protocols have staleness bounded by the commit interval. The state corresponding to the last committed anchor round is predictably valid, but more recent state depends on future vertices and is not yet determined.

Predicates over the current state (the common case for applications: “trigger when price crosses threshold,” “include when oracle updates”) require staleness 0. Predicates that tolerate bounded staleness δ (“trigger when price crossed threshold at some point in the last δ rounds”) can operate on DAG-committed prefixes. This is not a binary classification but a parameterized one.

2.3 Proposer Power Bundle

In this paper, we instantiate three categories of guarantees (Table 1). These are grouped by the block-construction dimension they primarily restrict and are use-case driven rather than fundamental, as the families can overlap and may conflict (see e.g. [33]).

Inclusion guarantees promise that a designated transaction will be present whenever a state predicate is true. An inclusion constraint consists of a transaction T and a predicate $P : \mathcal{S} \rightarrow \{0, 1\}$, and if $P(S) = 1$, then any block B such that $T \notin B$ is invalid. Observe that inclusion guarantees are attached to a specific transaction, and are determined entirely as a function of the pre-state, with no other dependence on the content of block B . See [33] for a more in-depth treatment of inclusion guarantees.

Exclusion guarantees promise that certain transactions will not be present. They consist of a predicate $Q : \mathcal{S} \times \mathcal{T} \rightarrow \{0, 1\}$, where if $Q(S, T) = 1$ for state S and transaction T , then any block B containing T is invalid. Exclusion guarantees can be registered by different parties: Users can attach exclusion guarantees to a specific transaction to prohibit its inclusion as a function of state (e.g., a user can submit a transaction to trade with an AMM, but require that it cannot be included if the AMM price is outside a certain desirable range). Applications can apply exclusion guarantees to all transactions that interact with the application (e.g., a trading application can implement a circuit-breaker when oracle staleness exceeds a threshold). Finally, the protocol itself can use exclusion guarantees (e.g., quarantine during proof system degradation). See [33] for a more in-depth treatment of exclusion guarantees.

Ordering guarantees make promises about the relative ordering of the subsequence of transactions targeting an application in a block. A contract C declares an ordering policy $(\pi_C, \{\tau_i\})$: a finite ordered list of recognized call sequences over C ’s functions, with optional tiebreakers τ_i evaluated on finalized state for intra-priority ordering. Transactions whose call sequences match a recognized pattern must appear in the block in the order specified by π_C ; transactions with unrecognized call sequences are automatically deprioritized and placed below all recognized transactions. See [34] for a more in-depth treatment of ordering guarantees.

Each guarantee has distinct implications for the incentives of participants and the consensus mechanism. Inclusion guarantees strengthen censorship resistance at the cost of liveness (more mandatory inclusions increase the proposer’s critical-path work). Exclusion guarantees enable application-level safety controls but introduce a potential griefing vector (malicious exclusion predicates). Ordering rights constrain the proposer’s intra-application ordering for adopting contracts but do not constrain inter-application ordering (where the application’s transaction subsequence sits within the block), inclusion decisions (which transactions targeting the contract enter the block), or interleaving with other contracts’ transactions. MEV shifts from the intra-application ordering surface to the inclusion, interleaving, and composability surfaces. Each requires its own incentive analysis (akin to [21]).

2.4 Extended State Machine

The standard SMR model treats transactions as opaque: the validity condition checks structural properties (signatures, well-formedness) and execution correctness (replicated transactions are re-executed by all validators). Symphony introduces flexibility into the state and validity layers: the state includes active triggers and ordering policies; the validity function references these via extended external validity. This subsection defines the extended state.

Symphony extends the standard SMR model [22] with additional state components:

Definition 2.3 (Symphony State). *The state S consists of:*

- *Execution state S_{exec} : account balances, contract storage (standard).*
- *Trigger pools: a set of active inclusion triggers $\mathcal{I} = \{(P_i, T_i, TTL_i)\}$ and exclusion triggers $\mathcal{E} = \{(Q_j, TTL_j)\}$, where each Q_j registered by a contract is scoped to transactions targeting that contract, an exclusion guarantee that a user attaches to its own transaction is scoped to that transaction, and triggers registered by the protocol itself are exempt.*
- *Ordering policies: a map from contract addresses to ordering policies $\{(\pi_C, \{\tau_i\})\}_{C \in \mathcal{C}}$, where $\mathcal{C}$ is the set of contracts with a registered ordering policy.*

An inclusion or exclusion trigger is *active* from its registration until its TTL expires, and an ordering policy is active until it is deregistered; each active trigger or ordering policy is an *active guarantee*.

State transitions fall into the following categories:

1. *Replicated execution*: deterministic state transitions executed by all correct replicas (standard EVM semantics).
2. *Trigger registration/deregistration*: adds or removes inclusion/exclusion triggers and ordering policies.

The block validity function over this extended state is defined in Section 3.

Delegated computations add a result registry, a verification state over proof systems, and the corresponding commit, settlement, verification and dispute transitions [31, 32]; this paper does not use them.

3 Extended External Validity

Symphony uses extended external validity as the single enforcement mechanism for all guarantees. Each guarantee from Section 2.3 adds a conjunct to the block validity function (Figure 1). The proposer retains only the residual power not constrained by these conjuncts. This section defines the extended validity function, proves that predictable validity is necessary, formalizes the efficiency of predicate evaluation, and derives the censorship resistance / liveness tradeoff.

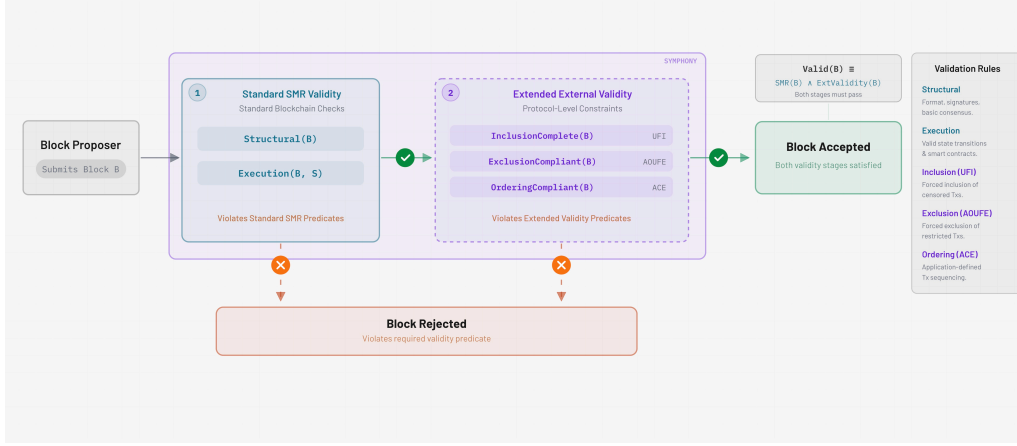


Figure 1: Extended external validity. Symphony augments standard SMR validity rules with three additional conjuncts—inclusion, exclusion, and ordering compliance—each enforced deterministically by every validator. Blocks violating any conjunct are rejected. In the figure, UFI, AOUE and ACE label the inclusion, exclusion and ordering conjuncts, respectively; $\text{Valid}(B) \equiv \text{SMR}(B) \wedge \text{ExtValidity}(B)$ abbreviates Definition 3.1, with $\text{SMR}(B)$ the conjunction of **Structural** and **Execution**, $\text{ExtValidity}(B)$ the conjunction of the three extensions, and the state argument S omitted.

3.1 Extended Block Validity

Definition 3.1 (Extended Block Validity). *A block B proposed for state S is valid if and only if all of the following conjuncts hold:*

$$\text{Valid}(B, S) = \text{Structural}(B) \quad (1)$$

$$\wedge \text{Execution}(B, S) \quad (2)$$

$$\wedge \text{InclusionComplete}(B, S) \quad (3)$$

$$\wedge \text{ExclusionCompliant}(B, S) \quad (4)$$

$$\wedge \text{OrderingCompliant}(B, S) \quad (5)$$

The first two conjuncts are standard in any BFT protocol:

- **Structural**(B): the block is well-formed, all transaction signatures are valid, the parent hash is correct, and the proposer is the designated leader for this round.
- **Execution**(B, S): the proposed post-state S' is the correct result of applying the block's transactions to S . For replicated transactions, every correct validator re-executes and checks.

The final three conjuncts are Symphony's extensions:

- **InclusionComplete**(B, S): for every active inclusion trigger $(P_i, T_i, \text{TTL}_i) \in \mathcal{I}$ such that $P_i(S) = 1$, the transaction T_i appears in B . The semantics are defined in [33].
- **ExclusionCompliant**(B, S): for every active exclusion trigger $(Q_j, \text{TTL}_j) \in \mathcal{E}$, no transaction $T \in B$ satisfies $Q_j(S, T) = 1$. The semantics are defined in [33].
- **OrderingCompliant**(B, S): for every contract C with a registered ordering policy $(\pi_C, \{\tau_i\})$, the subsequence of transactions in B that target C appears in the order specified by π_C and the tiebreakers τ_i evaluated on S . The semantics are defined in [34].

3.2 Power Guarantees as Validity Conjunct

All three guarantees follow the same structure. Each is implemented as a conjunct in Definition 3.1:

1. A beneficiary (user, application, or the protocol itself) registers a guarantee by publishing a predicate on-chain. This registers a trigger in $\mathcal{I}$, $\mathcal{E}$, or $\{(\pi_C, \{\tau_i\})\}$.
2. The predicate becomes a conjunct of the block validity function.
3. Every block proposed by a correct proposer must satisfy all active conjuncts.
4. Every correct validator independently evaluates all active conjuncts and rejects blocks that violate any. This deterministic evaluation is the default and the basis for Symphony’s BFT safety argument.
5. The proposer retains only the *residual* discretion: the freedom to act that is not constrained by active guarantees.

In the limit, if every possible proposer action is constrained by some guarantee, the proposer becomes a deterministic automaton: the block’s contents are uniquely determined by the state and the active predicates [30]. This extreme is undesirable (it eliminates the proposer’s ability to perform discretionary optimization), but it illustrates that proposer discretion lies on a continuum.

3.3 Predictable Validity Requirement

Proposition 2.2 (Section 2.2) establishes that extended external validity with current-state predicates requires predictable validity: when a proposer constructs their block, they can exactly compute the state resulting from confirmation of their block. We restate the implication in terms of system design.

Corollary 3.2 (Protocol Compatibility). *Symphony’s extended external validity with current-state predicates requires an SMR protocol that provides predictable validity with staleness 0; protocols that provide predictable validity only over a committed prefix with bounded staleness $\delta > 0$ support only predicates evaluated on that prefix.*

For protocols with staleness $\delta > 0$ (e.g., DAG-based protocols where the committed prefix lags the latest state), predicates must be modified to account for staleness: a predicate $P(S)$ becomes $P_\delta(S) = P(S_{\text{committed}})$, where $S_{\text{committed}}$ is the state corresponding to the last committed prefix of the ledger. The staleness δ is the number of uncommitted blocks (or rounds) between $S_{\text{committed}}$ and the current tip. Validators evaluate P against the same $S_{\text{committed}}$, which is determined by SMR, avoiding the validator-disagreement risk inherent in existential quantification over arbitrary past states.

In practice, any linear-commit BFT protocol (e.g., CometBFT [4], HotStuff [26], MonadBFT [19], Alpenglow [16], Simplex [17], Minimmit [18]) provides staleness 0.

3.4 Censorship Resistance / Liveness Tradeoff

This section formalizes the fundamental tradeoff between censorship resistance (enforced through inclusion guarantees) and liveness (the protocol’s ability to produce valid blocks in a timely manner).

3.4.1 Setup

Let $g_1, \dots, g_k$ be the active guarantees (inclusion triggers, exclusion triggers, and ordering policies). Let Δ_B be the block construction time budget: the maximum time a correct proposer has to construct a valid block before the round times out and the protocol moves to the next proposer.

Each active guarantee imposes two costs on the proposer:

1. *Evaluation cost*: the proposer must evaluate the predicate $P_i(S)$ or $Q_j(S, T)$ for each relevant transaction. Let c_i be the computational cost of evaluating the i -th predicate.
2. *Constraint cost*: each satisfied predicate constrains the set of valid blocks. Let $\mathcal{B}_i(S)$ be the set of blocks valid under the conjunct contributed by g_i . The set of valid blocks under all conjuncts is $\mathcal{B}(S) = \bigcap_{i=1}^k \mathcal{B}_i(S)$.

3.4.2 Block Construction Feasibility

Definition 3.3 (Feasibility). *A state S with active guarantees $g_1, \dots, g_k$ is feasible if $\mathcal{B}(S) \neq \emptyset$: there exists at least one valid block.*

Proposition 3.4 (Infeasibility from Conflicting Guarantees). *There exist sets of active guarantees that render a state infeasible. In particular, if an inclusion trigger requires inclusion of transaction T and an exclusion trigger requires exclusion of T , and both predicates are satisfied, then $\mathcal{B}(S) = \emptyset$.*

Proof. The inclusion conjunct requires $T \in B$, and the exclusion conjunct requires $T \notin B$. No block can satisfy both simultaneously. $\square$

Conflict detection is a satisfiability problem over the predicate set, formally related to constraint satisfaction [25] and the consistency of access control policies [13]. The protocol addresses this through conflict detection at registration time [33].

3.4.3 Liveness under Power Guarantees

Adding guarantees increases the proposer’s critical-path work and constrains the set of valid blocks, creating a tension between censorship resistance and liveness.

Evaluation cost. Each active predicate adds evaluation work to block construction. With k active predicates and per-predicate costs $c_1, \dots, c_k$, the total work is $\sum_{i=1}^k c_i$. Predicate evaluation is embarrassingly parallel: inclusion predicates ($P_i : \mathcal{S} \rightarrow \{0, 1\}$) are independent of each other, exclusion predicates ($Q_j : \mathcal{S} \times \mathcal{T} \rightarrow \{0, 1\}$) are independent per-transaction, and ordering policies $\{\pi_C\}$ are independent per-contract. With N_{par} parallel evaluation slots, the wall-clock cost is $\sum_{i=1}^k c_i / N_{\text{par}}$. The protocol cap k_{max} bounds total work: $\sum_{i=1}^{k_{\text{max}}} c_i \leq N_{\text{par}} \cdot \alpha \cdot \Delta_B$ for configurable $\alpha < 1$ and block construction budget Δ_B . The proposer faces one serial dependency: it must know all predicate outcomes before finalizing the block. Validators check a fixed block against predicates with no inter-predicate dependency, enabling full parallelism.

Feasibility. Each active predicate constrains the set of valid blocks: $\mathcal{B}(S) = \bigcap_{i=1}^k \mathcal{B}_i(S)$. Adding a predicate can only shrink this intersection. In the extreme, conflicting predicates make $\mathcal{B}(S) = \emptyset$ and no valid block exists (Proposition 3.4). Conflict detection at registration time [33] bounds this risk.

Monotonicity. The magnitude of this tension is monotone in k : more active predicates mean more evaluation work and a more constrained valid-block set. Both effects are individually monotone (adding a predicate never reduces evaluation cost or enlarges the valid-block set). Parallelism and the protocol cap reduce the cost but do not eliminate the tension, which is inherent in any system that adds validity constraints.

Protocol mitigation. Symphony bounds the tradeoff through three mechanisms:

1. *Protocol cap on active predicates:* $k_{\max}$ limits the number of simultaneously active inclusion/exclusion triggers.
2. *Economic filtering:* registering a trigger requires a deposit that is slashed if the predicate is malformed or excessively expensive to evaluate.
3. *Conflict detection:* overlapping inclusion and exclusion predicates are checked at registration time. Conflicting registrations are rejected unless a resolution policy is declared. A natural relaxation (future work) is to resolve conflicts via an auction over predicate slots, pricing slot contention rather than rejecting it outright.

3.5 Predicate Evaluation

Evaluating all active predicates on every block proposal is the critical-path cost of extended external validity.

3.5.1 Default: Full Deterministic Evaluation

In the default mode, every correct validator independently evaluates every active predicate. The proposer includes an *evaluation certificate*, a summary of which predicates are satisfied and which transactions are mandated or excluded. Validators treat this as advisory and perform full re-evaluation. This is the mode under which Symphony’s BFT safety proofs hold: all correct validators agree on validity because they all evaluate the same deterministic predicates over the same predictably valid state.

The total work is $O(n_{\text{trig}})$ per validator per block, where n_{trig} is the number of active predicates. Since predicates are independent, this work is parallelizable across N_{par} cores, yielding wall-clock cost $O(n_{\text{trig}}/N_{\text{par}})$. The protocol cap $k_{\max}$ (Section 3.4.3) bounds the total work.

Dependency tracking. Each predicate declares the set of state slots it reads. When a block modifies state slot s , only predicates with s in their dependency set need re-evaluation. This reduces the amortized cost to $O(\text{changed_slots} \times \text{dependent_predicates})$.

4 Liveness

Liveness means that every correct transaction eventually makes progress.

Symphony inherits SMR liveness from the underlying BFT protocol [9]. After GST, every correct proposer eventually produces a block that is committed by all correct replicas. The extended validity conjuncts add computational cost to the critical path but do not fundamentally alter the SMR liveness proof, provided the predicate evaluation budget $\alpha \cdot \Delta_B$ is respected (Section 3.4.3).

Liveness additionally requires that the active guarantees leave at least one valid block, $\mathcal{B}(S) \neq \emptyset$ (Definition 3.3). Adding a guarantee can only shrink $\mathcal{B}(S)$, and conflicting guarantees can make it empty (Proposition 3.4); conflict detection at registration time [33] bounds this risk.

5 Discussion

5.1 Limitations

Predictable validity. Symphony’s extended external validity requires predictable validity: the proposer can exactly compute the state resulting from confirmation of their block (Proposition 2.2). This limits compatibility to SMR protocols that provide this property, excluding or limiting protocols with high staleness. DAG-based protocols can use bounded-staleness predicates (Corollary 3.2), but this weakens the precision of inclusion/exclusion triggers.

Predicate evaluation overhead. Extended external validity adds computational cost to the block production critical path. The censorship-resistance/liveness tradeoff (Section 3.4.3) is fundamental: stronger censorship resistance comes at the expense of liveness. However, predicate evaluation is embarrassingly parallel (Section 3.4.3): inclusion, exclusion, and ordering predicates are independent, and the effective budget scales as $N_{\text{par}} \cdot \alpha \cdot \Delta_B$ with N_{par} parallel evaluation slots. The protocol cap k_{max} bounds total work, but the wall-clock cost scales as $O(k_{\text{max}}/N_{\text{par}})$, making the overhead practical for modern multi-core validators.

6 Related Work

The SMR framework [15, 22] is the theoretical foundation of blockchain SMR. PBFT [7] demonstrated an SMR protocol with three rounds. Subsequent lines of work—Simplex [17], Tendermint (CometBFT) [4], and HotStuff [26]—improved on it in several metrics. Recent protocols such as MonadBFT [19], Alpenglowl [16], and Minimmit [18] hyper-optimize latency and block dissemination throughput. DAG-based protocols (Narwhal/Tusk [8], Bullshark [24]) decouple data dissemination from ordering to achieve higher throughput. Avalanche [20] takes a metastable approach with probabilistic finality.

All of these optimize *performance* (latency and throughput) under the assumption that execution is cheap. Symphony is orthogonal: it addresses the execution model, not the SMR protocol, and can layer on top of any SMR protocol that provides predictable validity (Section 2.2).

Cachin, Kursawe, and Shoup [6] introduced external validity in the context of SMR: the validity condition on proposed values can reference application semantics. Ethereum’s block validity rules (correct state root, valid transactions, valid proposer) are a form of external validity. Fork-choice rules (LMD-GHOST, Casper FFG) impose additional validity constraints on block acceptance.

Symphony’s contribution is extending external validity with guarantee-specific conjuncts (Section 3): forced inclusion, forced exclusion, and ordering compliance [33, 34]. No prior work uses external validity as the enforcement mechanism for application-defined transaction policies.

The CAP theorem [3, 12] is the structural analogue of the CR/liveness tradeoff (Section 3.4.3): adding constraints to validity reduces the likelihood of producing valid blocks.

7 Conclusion

Inclusion, exclusion, and sequencing rights are transferred from the proposer to the protocol, users, and applications through three mechanisms, each enforced as a conjunct of the block validity function in the SMR critical path. The enforcement is deterministic and parallelizable across predicates.

Predictable validity is necessary for extended external validity with current-state predicates (Proposition 2.2): if a correct proposer and a correct validator can derive different pre-states,

some state predicate makes them disagree on the validity of a block, so current-state predicates require an SMR protocol with predictable validity at staleness 0 (Corollary 3.2). Each active guarantee adds evaluation work to block construction and can only shrink the set of valid blocks, and conflicting guarantees can leave no valid block (Section 3.4.3); a protocol cap on active predicates, registration deposits and conflict detection at registration bound this tradeoff.

The process of building and distributing a block is larger than what is explored here. Beyond inclusion, exclusion, and ordering guarantees, there exist other dimensions such as proposal timing, blob/block data dissemination, the right to fees, and priority guarantees.

References

- [1] Ethereum Research. Application-controlled execution: A case study on cancel prioritization. <https://ethresear.ch/t/application-controlled-execution-a-case-study-on-cancel-prioritization/23977>.
- [2] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.* 5(8):776–787, 2012.
- [3] E. A. Brewer. Towards robust distributed systems. *PODC Keynote*, 2000.
- [4] E. Buchman, J. Kwon, Z. Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938, 2018.
- [5] V. Buterin. Two-slot proposer-builder separation. Ethereum Research, <https://ethresear.ch/t/two-slot-proposer-builder-separation/10980>, 2021.
- [6] C. Cachin, K. Kursawe, V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *J. Cryptology* 18(3):219–246, 2005.
- [7] M. Castro, B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd Symp. Operating Systems Design and Implementation (OSDI)*, pp. 173–186, 1999.
- [8] G. Danezis, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proc. 17th EuroSys Conf.*, pp. 34–50, 2022.
- [9] C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 35(2):288–323, 1988.
- [10] Ethereum Foundation. EIP-7547: Inclusion lists. 2023.
- [11] Flashbots JSON-RPC Endpoints. 2026. https://docs.flashbots.net/flashbots-auction/advanced/rpc-endpoint#eth_sendbundle
- [12] S. Gilbert, N. Lynch. Brewer’s conjecture. *SIGACT News*, 33(2), 2002.
- [13] M. A. Harrison, W. L. Ruzzo, J. D. Ullman. Protection in operating systems. *Commun. ACM* 19(8):461–471, 1976.
- [14] M. Kelkar, F. Zhang, S. Goldfeder, A. Juels. Order-fairness for Byzantine consensus. In *Proc. CRYPTO*, pp. 451–480, 2020.
- [15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21(7):558–565, 1978.
- [16] Anza. Alpenglow: Solana consensus upgrade. 2025.

- [17] B. Y. Chan, R. Pass. Simplex consensus: A simple and fast consensus protocol. IACR ePrint 2023/463, 2023.
- [18] B. K. Chou, A. Lewis-Pye, P. O’Grady. Minimmit: Fast finality with even faster blocks. arXiv:2508.10862, 2025.
- [19] M. M. Jalalzai, K. Babel, J. Komatovic, T. Klenze, S. Das, F. Elsheimy, M. Setrin, J. Bergschneider, B. Gilkalaye. MonadBFT: Fast, responsive, fork-resistant streamlined consensus. arXiv:2502.20692, 2025.
- [20] Team Rocket, M. Yin, K. Sekniqi, R. van Renesse, E. G. Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. arXiv:1906.08936, 2020.
- [21] T. Roughgarden. Transaction fee mechanism design. In *EC*, 2021.
- [22] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.
- [23] Shutter Network. 2022.
- [24] A. Spiegelman, N. Giridharan, A. Sonnino, L. Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*, pp. 2705–2718, 2022.
- [25] E. Tsang. *Foundations of Constraint Satisfaction*. 1993.
- [26] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proc. 2019 ACM Symp. Principles of Distributed Computing (PODC)*, pp. 347–356, 2019.
- [27] B. Monnot. Decoupling throughput from local building. *Ethereum Research*, <https://ethresear.ch/t/decoupling-throughput-from-local-building/22004>, 2024.
- [28] B. Monnot. Unbundling staking: Towards rainbow staking. *Ethereum Research*, <https://ethresear.ch/t/unbundling-staking-towards-rainbow-staking/18683>, 2024.
- [29] B. Monnot. Unbundling PBS: Towards protocol-enforced proposer commitments (PEPC). *Ethereum Research*, <https://ethresear.ch/t/unbundling-pbs-towards-protocol-enforced-proposer-commitments-pepc/13879>, 2022.
- [30] M. Bahrani. Beyond Multidimensional Fee Markets. *Talk, Devcon* Available at: <https://www.youtube.com/watch?v=qf51v48KhH0>. 2024.
- [31] Ritual. Superposition of Execution Models. 2026.
- [32] Ritual. Multi-System Verification Model. 2026.
- [33] Ritual. Inclusion and Exclusion Guarantees. 2026.
- [34] Ritual. Application-Defined Ordering Guarantees. 2026.
- [35] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.