

The Circuit Model: A Formal Specification of Events and Items in Distributed Systems

Gabriel B. Rondon
me@grondon.com

October 29, 2025

Abstract

This paper introduces the formal structure of circuits as a general model for distributed coordination and state evolution. A circuit represents a composition of items and events, where items describe discrete stateful entities, and events represent atomic transitions that affect those states. The model focuses on mathematical consistency, verifiable causality, and compositionality without reliance on global consensus. This specification defines the primitives, transition functions, synchronization semantics, a pluggable adapter architecture for physical storage, and privacy-preserving mechanisms that govern the operation of circuits as distributed state machines.

1 Introduction

Distributed systems often require coordination between autonomous agents operating over shared or partially shared data. The notion of a circuit emerges as a minimal abstraction that allows multiple actors to publish, modify, and synchronize state changes in a verifiable manner. Rather than a single ledger or authority, circuits describe relational domains where items evolve independently but can reference and influence each other through structured events. This model provides a storage-agnostic foundation for composable and causally consistent data coordination, which can be implemented on a variety of physical backends through a pluggable adapter architecture.

2 Formal Definitions

Let Σ be the set of all possible item states, and E the set of all valid events. Each item $i \in I$ maintains a local state $\sigma_t \in \Sigma$ that evolves through events applied over time.

The basic state transition is defined as:

$$\sigma_t = \Psi(\sigma_{t-1}, E_t) \quad (1)$$

where $\Psi : \Sigma \times E \rightarrow \Sigma$ is a deterministic transition function mapping a prior state and event into a new state. Each event is represented by the tuple:

$$E_i = (\rho_i, \delta_i, h_{i-1}, \tau_i, \pi_i) \quad (2)$$

where ρ_i is the payload, δ_i is the author or proof of origin, h_{i-1} is the hash reference to the previous event, τ_i is a timestamp for conflict resolution, and π_i is an optional zero-knowledge proof for confidential transactions.

Event Payload Structure (ρ)

The payload ρ is not an opaque blob of data; it is a structured object that specifies the intended state transition. Its structure typically includes an operation identifier and its corresponding parameters. For example, a payload could be represented as a JSON object:

```
{  
  "op": "update_field",  
  "field": "owner",  
  "value": "new_owner_id"  
}
```

The transition function $\Psi(\sigma_t, E_t)$ interprets the ‘op’ field within the payload $E_t.\rho_t$ to deterministically compute the next state σ_{t+1} . This structure makes events explicit commands rather than ambiguous data.

The event hash is defined recursively as:

$$h_i = H(\rho_i, \delta_i, h_{i-1}, \tau_i, \pi_i) \quad (3)$$

The cumulative state hash of an item, representing its complete history, is updated with each event:

$$\Sigma_t = H(\Sigma_{t-1}, h_t) \quad (4)$$

3 Synchronization and Causality

Circuits enable asynchronous publication of events by multiple actors. When two event sequences share a common ancestor but diverge, a fork condition arises:

$$E_i^a, E_j^b \Rightarrow \text{fork if } h_{i-1}^a \neq h_{j-1}^b$$

Merging such divergent branches is handled by a merge operator M , producing a resolved event:

$$M(E_i^a, E_j^b) \rightarrow E_k \quad (5)$$

The merge operator is a deterministic function that resolves conflicts between divergent histories. Its implementation is a critical part of a circuit’s rule set \mathcal{R} .

3.1 Merge Strategies

While many merge strategies exist, a common and simple approach is Last-Write-Wins (LWW), which uses the event timestamp τ introduced previously. A LWW merge operator M_{LWW} can be defined as:

$$M_{LWW}(E_i^a, E_j^b) = \begin{cases} E_i^a & \text{if } \tau_i^a \geq \tau_j^b \\ E_j^b & \text{if } \tau_j^b > \tau_i^a \end{cases} \quad (6)$$

In this strategy, the event with the later timestamp is chosen as the “winner,” and the other is discarded, effectively pruning the fork. The choice of merge strategy is a domain-specific decision defined within the circuit’s rule set \mathcal{R} .

4 Composition of Circuits

A circuit C is defined as the tuple:

$$C = (\mathcal{R}, I_0, I_1, \dots, I_n) \quad (7)$$

where \mathcal{R} represents the set of rules governing event validation and composition.

4.1 Rule Specification (\mathcal{R})

The rule set \mathcal{R} is the logical core of a circuit, defining its behavior and ensuring integrity. It is a collection of predicate functions that must all evaluate to true for an event to be considered valid. The validation function $V_C(E_i)$ ensures that an event complies with all public rules and, if applicable, provides a valid zero-knowledge proof for its confidential components. It can be expressed as a two-part condition:

$$V_C(E_i) = (\forall r \in \mathcal{R}, r(E_i, C)) \wedge (\text{verify_zkp}(E_i.\pi_i) \vee E_i.\pi_i = \text{null}) \quad (8)$$

Each rule r is a function that takes the event E_i and the entire circuit C (including the states of all its items) as input. This allows rules to enforce complex, cross-item invariants.

4.2 Circuit State Hash

While individual items maintain their own state hashes, the collective state of the entire circuit can be represented by a single, deterministic hash. This allows for efficient verification of state consistency across distributed replicas. Let $\mathcal{I} = \{I_0, I_1, \dots, I_n\}$ be the set of items in circuit C , and let $\Sigma_{I_k,t}$ be the state hash of item I_k at time t . The circuit state hash, $H_{C,t}$, is defined as the hash of the sorted collection of its items' state hashes:

$$H_{C,t} = H(\text{sort}(\Sigma_{I_0,t}, \Sigma_{I_1,t}, \dots, \Sigma_{I_n,t})) \quad (9)$$

An event targeting any item I_k will alter its state hash Σ_{I_k} , which in turn changes the overall circuit state hash H_C . This provides a simple and powerful mechanism to confirm if two distributed copies of a circuit are in the same state.

5 Storage and Network Adapters

The Circuit Model itself is a purely logical specification, defining the rules of state evolution without mandating a physical storage medium. The link between the abstract model and a concrete implementation is provided by **Adapters**. An adapter is a pluggable component chosen by the circuit owner that dictates where and how event data is published, stored, and retrieved.

5.1 Separation of Concerns

This architecture creates a clean separation of concerns. The circuit is responsible for:

- Validating event structure and rules (\mathcal{R}).
- Executing the state transition function (Ψ).
- Creating the cryptographic hashes (h_i, Σ_t, H_C).

The adapter is responsible for:

- Persisting a valid event to a specific storage backend.
- Retrieving event histories or item states from that backend.

5.2 Adapter Interface

Conceptually, any adapter implements a simple interface, primarily:

- **publish(event)**: Takes a locally-validated event and broadcasts it to the network or writes it to the storage layer. It returns a storage-specific identifier (e.g., a transaction hash, an IPFS CID).
- **resolve(item.id)**: Fetches the event history for a given item from the storage layer, allowing a node to reconstruct its current state.

5.3 Example Adapters

The flexibility of this model is demonstrated by the variety of possible adapters:

- **Blockchain Adapter:** An event can be embedded into a transaction on a public blockchain (e.g., Ethereum). The blockchain's global consensus provides a definitive, ordered log, making fork resolution trivial. This adapter is ideal for high-security applications but may incur higher costs and latency.
- **IPFS Adapter:** Events can be stored as immutable objects in the InterPlanetary File System (IPFS). The hash-chain linkage of events is preserved by embedding the parent's IPFS Content Identifier (CID) in the child object. This approach is highly decentralized and cost-effective for data availability.
- **Federated Database Adapter:** For private or consortium-based circuits, an adapter could simply write events to a replicated, high-availability database (e.g., PostgreSQL with streaming replication) managed by a set of trusted parties.

The circuit owner's choice of adapter determines the trade-offs between decentralization, security, performance, and cost.

6 Properties and Integrity

The circuit model ensures the following properties:

- **Deterministic replay** — given an initial state and event log, any node can reconstruct the same state.
- **Causal integrity** — all events maintain cryptographic linkage ensuring lineage and immutability.
- **Local validation** — rules \mathcal{R} define event acceptance at circuit scope, allowing domain-specific enforcement.
- **Compositionality** — circuits can be combined or nested, forming higher-level coordination systems.
- **Confidentiality** — with the use of hybrid encryption and zero-knowledge proofs, event payloads can be kept private while still being publicly verifiable against the circuit rules.
- **Storage Agnosticism** — the logical model is decoupled from the physical storage layer, allowing for flexible deployment on various backends via adapters.

7 Privacy and Confidentiality

While transparency is a core feature of many distributed ledgers, practical applications often require the ability to handle sensitive information. The Circuit Model can be extended to support confidential transactions through a combination of payload structuring, hybrid encryption, and zero-knowledge proofs, addressing the inherent conflict between privacy and public verifiability.

7.1 Hybrid Event Payloads

To support confidentiality, the event payload ρ is partitioned into two components: a ‘public’ section and an ‘encrypted’ section.

```
payload = {
    "public": { ... },
    "encrypted": "ciphertext"
}
```

The ‘public’ portion contains non-sensitive data required for public processing. The ‘encrypted’ portion contains the sensitive data.

7.2 Confidential Transactions with Hybrid Encryption

The ciphertext is generated using a standard hybrid encryption scheme.

1. A fresh, random symmetric key is generated.
2. The sensitive payload data is encrypted with this symmetric key.
3. The symmetric key is then encrypted with the asymmetric public key of each intended recipient.

To send to a group, the symmetric key is encrypted for each member, and all resulting ciphertexts are included.

7.3 The Privacy vs. Verifiability Trade-off

A fundamental challenge arises when the data required to validate a transaction is sensitive. The circuit’s rules \mathcal{R} are executed by all nodes, but they cannot operate on encrypted data.

7.4 Validation with Zero-Knowledge Proofs

This trade-off is resolved by using zero-knowledge proofs (ZKPs). The author of a confidential event generates a proof π that demonstrates their secret data complies with all relevant public rules. The validation function V_C is extended to verify this proof, ensuring that even confidential state transitions adhere to the circuit's logic.

8 Related Work

The challenge of ordering events and maintaining consistency in distributed systems is fundamental. Lamport's work on logical clocks provides a foundational mechanism for capturing causal relationships [2]. Milner's π -calculus offers a formal framework for describing concurrent processes [3].

Unlike monolithic blockchain models, which typically rely on a single, global consensus mechanism for ordering a total log of transactions, the Circuit Model provides a more granular, compositional approach. Circuits define independent state machines whose evolution is locally validated, similar to the actor model, but with the addition of cryptographic proof of history. This allows for greater parallelism and modularity, resembling concepts from Directed Acyclic Graph (DAG) based ledgers.

9 Discussion

The circuit model abstracts the minimal mathematical conditions required for distributed verifiable state management. It generalizes the concept of event-driven systems, representing causality as a sequence of cryptographically linked transitions. While implementation details may vary, the formal properties remain invariant: every item is a deterministic state machine, every event is a valid transformation, and every circuit defines a verifiable local universe of rules and relationships.

10 Conclusion

This paper has introduced the Circuit Model, a formal specification for describing state evolution in distributed systems. By defining clear primitives for state transitions, event structure, and validation, the model provides a robust foundation for building causally consistent and composable systems. The cryptographic linkage of events ensures integrity, while local validation allows for domain-specific logic without global consensus. Future work will explore a library of reference implementations for various adapters, analyze performance characteristics across different backends, and further develop advanced privacy-preserving validation techniques using zero-knowledge proofs.

References

- [1] Gabriel B. Rondon, “On Circuits and Distributed Event Coordination”, 2025.
- [2] Lamport, L., “Time, Clocks, and the Ordering of Events in a Distributed System”, CACM, 1978.
- [3] Milner, R., “Communicating and Mobile Systems: The π -Calculus”, Cambridge University Press, 1999.