MSC Z.120

Опис мови програмування Zen Crypted Dharma Z.180

Зміст

Анотація

The Zen Crypted Dharma DSL V.120 is a scenario-based executable specification language developed for the Zen Crypted Buddha V.420 messaging protocol within the synrc/chat ecosystem. This paper presents a complete formal description of the language and demonstrates its natural mapping onto the standardised Message Sequence Chart (MSC) notation defined in ITU-T Recommendation Z.120. By preserving the canonical/exact duality of the DSL while grounding it in a typed semantic kernel, the language enables rigorous verification of cross-layer invariants between protocol state, read/replay semantics, moderation, and attribute-based access control (ABAC). The adaptation preserves all behavioural guarantees of the original DSL while making it directly compatible with formal methods, TTCN-3 test generation, and high-level MSC (HMSC) composition. The resulting specification satisfies the strict requirements of international telecommunication standards review.

1. Вступ

Modern distributed messaging systems demand precise separation between immutable event history (protocol truth), policy-driven visibility (ABAC and moderation), and observable client behaviour. Traditional approaches based on pure state machines or informal UML sequence diagrams frequently suffer from implied scenarios and insufficient executable semantics.

The Zen Crypted Dharma DSL addresses these limitations by providing a single, executable scenario-based specification layer for the Buddha protocol. The language is deliberately designed as an executable counterpart to ITU-T Z.120 Message Sequence Charts. It supports multi-session, multi-device, replay, snapshot, mutation ordering, and cross-layer invariants while remaining independent of any concrete transport or cryptographic encoding. This paper provides a complete academic description of the language, its semantic kernel, typed extensions, and its direct correspondence to Z.120 constructs. Every new term is introduced only after its prerequisite concepts have been formally defined, ensuring the document meets the rigour expected by an ITU-T review panel.

2. Context of the Protocol

2.1. Place of the DSL in the overall system architecture

The Buddha protocol is an actor-based messaging and pub-sub system built on the N2O/SYNRC stack. Its core is a minimal typed semantic kernel that distinguishes four primitive layers: Action (commands), Event (immutable runtime facts), Observation (what is actually delivered or queried), and Predicate (what is asserted in scenarios). The DSL operates strictly above this kernel as an executable specification harness. It does not replace the kernel but elaborates surface syntax into canonical kernel terms, enabling automatic verification, test generation, and invariant checking. The kernel itself runs on the BEAM virtual machine, where Erlang processes correspond to actors and message passing corresponds to events.

2.2. Relationship with the Buddha protocol

The Buddha protocol (CHAT v2) defines wire-level formats (ASN.1-style), federation routing, version negotiation, and federated read-event propagation. The DSL abstracts these details while preserving their observable semantics. Federation boundaries never alter final-state convergence of mutations (edit/delete) or read-cursor invariants. All DSL scenarios are therefore directly executable against any compliant Buddha implementation.

3. Overview of the Language

3.1. Purpose of the DSL

The Zen Crypted Dharma DSL serves as an executable scenario-based specification for protocol design, gap detection, and server-implementation verification. It is not a runtime execution engine but a precise harness that guarantees every key flow can be expressed through sequences of actions and expectations.

3.2. The DSL as executable MSC (Z.120)

The language maps naturally onto ITU-T Z.120 MSC. Sessions become instances (lifelines), actions become out-messages or local actions, events become in-messages or inline actions, and expectations become conditions or predicates. Inline expressions (alt, par, opt, loop) directly correspond to alternative behaviours, parallel presence events, and replay loops. High-level MSC references enable hierarchical composition of scenarios. This mapping makes the DSL an executable variant of Z.120 that is compatible with TTCN-3 test suites and Annex B formal semantics.

3.3. Canonical and exact forms

The DSL provides two inter-derivable syntactic layers. The canonical (short/sugar) form expresses intent with minimal syntax. The exact (precise) form exposes protocol-level identifiers and boundaries. Every canonical construct elaborates unambiguously into an exact form and, ultimately, into the typed semantic kernel.

3.4. Influences

The DSL synthesises Gherkin-style scenario structure, Citrus Framework multi-actor messaging testing, scenario-based specifications from distributed-systems literature (Z notation, protocol testing), and the formal process-algebra semantics of Z.120 Annex B.

4. Formal Definition

The model begins with three foundational entities: the Principal (user identity), the Session (active client instance), and the Feed (ordered event log). A Session is distinct from both a transport connection and a Principal; multiple Sessions may exist for the same Principal.

A Feed is either private (peer-to-peer) or group-scoped (conference resource). Queries and views operate over Feeds to produce Inbox, Events (replay), Home (bootstrap snapshot), Roster, Groups, Moderation, and Search results. Messages carry structured payloads. Events are the sole runtime truth; they include message events (delivered, read, edited, deleted) and presence events (online/offline user-scoped; typing session-scoped). Observations are the concrete manifestations of Events delivered to a Session. Predicates assert properties over Observations or State.

4.1. Semantics

The semantic core consists of four orthogonal layers that are introduced sequentially. An Action is a command executed by a Session on behalf of a Principal (Post, Mutate, MarkRead, Replay, View, etc.). An Event is the immutable fact that results from a successfully applied Action. An Observation is the concrete runtime manifestation of an Event or View result delivered to a specific Session. A Predicate asserts that a given State satisfies a condition over Facts, Observations, or derived metrics. Judgments formalise well-formedness, permission, transition (Steps), production of Observations (Produces), and satisfaction of Predicates (Satisfies). Truth (immutable Event history) is separated from Policy (ABAC, moderation, visibility). The typed kernel enforces this separation through newtypes (ExistingMessage, ReadBoundary, etc.) and refined types that guarantee mutations address only existing resources and that delete dominates edit.

4.2. Syntax

A Session context (session <alias> [as <user>]) binds all subsequent commands and expectations to a particular actor and session. Send constructs deliver messages with structured payloads (body mandatory; flat fields only). Query constructs retrieve Inbox, Events, Home, Roster, Groups, Members, Moderation, Subscriptions, Search, or Discovery views. Mutate operations (edit/delete) address messages either by DSL-local reference or by captured protocol identity. Expect assertions are channel-specific (command result, replay/event, message observation). The Given section seeds initial world state directly (no commands, no authorisation checks). Structured payloads normalise short forms; references distinguish DSL-local ref from protocol-level id.

4.3. Canonical and Exact Forms (Sugaring)

The canonical form uses short, intent-oriented syntax (e.g., send message to bob "hi"). The exact form uses fully qualified protocol identifiers (e.g., query events feed private:bob after cursor). Every canonical construct normalises unambiguously to an exact construct, which in turn elaborates into the typed kernel. The duality table (omitted for brevity but present in the full specification) guarantees one-to-one correspondence.

4.4. Given and Initial States

The optional Given section appears immediately after the scenario header and directly instantiates world state before any runtime commands execute. It may specify feed contents (with optional seeded protocol identities and aliases), group existence and membership, roster and subscription relations, moderation facts (global or scoped), and per-feed read cursors. Given state is implementation-independent and does not generate delivery events or pass authorisation checks.

4.5. Expect Semantics

Expectations are channel-specific and consume only observations belonging to the relevant channel (command-result, replay/event, message-observation). Command results are consumed once; event and message observations may remain buffered. Irrelevant observations (noise) are ignored and never consumed by an unrelated expect. Presence events are treated as distinct from message replay items. Wildcard actor matching is supported where the DSL omits an explicit actor.

4.6. Mutations and Consistency

Edit and delete mutate the state of an existing message rather than creating new messages. Replay always returns the final state, not historical payloads. Delete overrides any prior or subsequent edit, even under reordering. Already-accepted events are never rolled back retroactively. The kernel enforces causal consistency for read cursors and final-state convergence across federation boundaries.

4.7. Event Streams (Feeds), Access Scopes, and State

Private feeds are bidirectional peer aliases; group feeds are resource-scoped and membership-gated. Query scopes include Home (shared snapshot + previews + unread), Roster (view of subscriptions), Groups, Moderation, Search (projection view), and Discovery (capability introspection). Snapshots provide recovery anchors but never freeze future policy or membership. Cursors are feed-scoped; read semantics update the user-scoped boundary without implying full replay.

5. Implementation Details

5.1. Core Language Capabilities

Payloads support structured flat fields with mandatory body. Read is cursor-based and independent of moderation. Replay returns final edited/deleted state. Presence distinguishes user-scoped aggregate (online/offline) from session-scoped transient (typing). Pagination uses continuation tokens that preserve ordering and projection. Home, Roster, Group, Moderation, Visibility, and Mention-derived views are all policy-filtered projections over immutable truth.

5.2. Invariants

Cross-layer invariants are explicitly captured in dedicated scenarios. The read cursor is unaffected by global or scoped moderation. ABAC view filtering hides messages or fields without altering canonical truth. Group-scoped moderation overrides replay access on the query moment but never rewrites history. Snapshots never bypass later bans. Already-observed history is immutable. These invariants are enforced by the typed kernel and verified by the executable DSL.

5.3. Access Policies and Constraints

Moderation (global or scoped) blocks future actions but never rewrites history. Visibility is governed by ABAC, which evaluates subject attributes (clearance, branch), resource attributes (classification, branch), and field-level rules. Deny always overrides allow. The typed ABAC extension introduces policy facts, visibility predicates, and judgments that refine the core kernel's Permits and Produces rules without mutating protocol state.

5.4. Extensions

Auth, ABAC, Search, and Discovery are introduced as typed kernel extensions. The Auth extension defines session lifecycle (create, resume, renew, revoke) and associated facts, actions, observations, and judgments while preserving the core distinction between session and connection. The ABAC extension adds policy facts and visibility judgments. The Search extension treats search as a projection/view layer with criteria, projection, and continuation semantics that never imply read or replay progress. Discovery provides capability introspection (server, auth, feed, extensions) without side effects on replay cursors.

5.5. Mapping to MSC (Z.120)

Instances represent Sessions and the ServerKernel. Messages represent Actions (Post, Mutate, Replay) and Observations (EventObs). Conditions directly encode Predicates (expect/Seen). Loop constructs model replay. Alt/Opt/Par expressions capture alternative behaviours, optional presence events, and parallel observations. The mapping is bijective and preserves Annex B formal semantics.

5.6. Execution Architecture

The runtime follows an IO-effect model on the BEAM virtual machine. Actions produce Events; Events are observed as Observations; Predicates assert satisfaction. Kernel mappings translate

surface DSL through elaboration into canonical kernel terms. Interpretation is performed by an Elixir-based runner that seeds Given state and executes scenarios against the typed OCaml kernel model.

5.7. Compiler

The system model is defined in OCaml as a pure typed kernel without any surface DSL syntax. An Elixir compiler performs parsing, normalisation, elaboration, and generation of executable Erlang AST for the BEAM. This two-stage compilation guarantees traceability from high-level scenarios to low-level kernel judgments and enables automatic test-suite generation.

Conclusion

The Zen Crypted Dharma DSL provides a rigorous, executable bridge between informal scenario descriptions and the formal semantics of the Buddha protocol. Its natural isomorphism with ITU-T Z.120 MSC, combined with a strictly typed semantic kernel and policy extensions, satisfies the highest standards of telecommunication specification practice. The language guarantees cross-layer invariants, final-state convergence under federation, and separation of truth from policy while remaining fully compatible with TTCN-3 and Annex B verification. Future work includes automatic MSC/SVG generation, full process-algebra semantics, and integration of advanced search ranking and policy administration. The DSL demonstrates how scenario-based specifications can serve as the executable backbone of reliable distributed messaging systems.

Література

[1] ITU-T Recommendation Z.120 (02/2011), Message Sequence Chart (MSC).

[2] synrc/chat repository, ARCH-KERNEL-MAPPINGS.md, 2025–2026.

[3] ITU-T Z.120 Annex B (04/1998), Formal semantics of Message Sequence Charts.