unit: A Self-Replicating Software Organism with Multi-Order Evolution, Sexual Reproduction, and Niche Construction

David Liedle
david.liedle@protonmail.com

https://github.com/DavidCanHelp/unit | *https://davidcanhelp.github.io/unit/*

Abstract

We present unit, a self-replicating software organism implemented as a Forth interpreter that is simultaneously a networked mesh agent. Each unit discovers peers over UDP gossip, evolves programs through genetic programming, replicates by packaging its own binary and state, and provides an interactive REPL inside the organism itself. Unlike prior ALife platforms that operate as closed simulations on bounded grids, unit organisms are independent OS processes that genuinely self-replicate, communicate via S-expressions, and evolve across real networks. The system implements three orders of evolution operating simultaneously: a GP engine evolves solutions to fitness challenges (first order), a meta-evolver evolves the challenge generators themselves (second order), and a scoring population evolves the fitness functions that judge generators (third order). Recent additions include sexual reproduction through dictionary crossover between mating pairs selected by tournament selection, and niche construction where organisms modify their own selection pressures based on demonstrated competencies. Three species (Rust/Forth, Go, Python) coexist on one mesh, each with a different cognitive substrate but sharing a common S-expression protocol. The system compiles to both native binaries and WebAssembly with zero external dependencies, comprising approximately 35,000 lines across four languages. We present the design, formal properties, and implications for open-ended evolution research.

1. Introduction

The study of self-replicating programs has been central to artificial life since its inception. Landmark systems including Tierra [1], Avida [2], and Coreworld demonstrated that self-replicating programs could exhibit evolutionary dynamics analogous to biological systems. More recently, Agüera y Arcas et al. [3] showed that self-replicating programs can emerge spontaneously from random code.

However, existing ALife platforms share several limitations: they operate as closed simulations on bounded grids or shared memory spaces; they use custom assembly-like instruction sets designed for evolvability but not human interaction; they define fitness externally through researcher-specified criteria; and they provide observation-only interfaces where the experimenter watches from outside rather than interacting with individual organisms.

We present *unit*, a self-replicating software organism that addresses these limitations. Unit is a Forth interpreter that is simultaneously a networked mesh agent: it discovers peers over UDP, distributes computation as S-expressions, evolves its own programs through genetic programming, persists its state as human-readable JSON, and provides an interactive REPL inside the organism itself. The same Rust codebase compiles to native binaries and WebAssembly, enabling both CLI deployment across real networks and browser-based demonstration.

Unit was not designed as an ALife system. It emerged from a software engineering premise: "What is the smallest self-reproducing unit of software that can coordinate with its mesh?" The resulting design decisions—Forth for minimality and mutability, S-expressions for universal communication, JSON for inspectable persistence—converge with principles the ALife community has developed independently over three decades. This paper describes the system architecture, its evolutionary mechanisms including sexual reproduction and niche construction, its formal properties, and its implications for open-ended evolution research.

2. System Architecture

A unit organism is organized around five concerns: Execute (Forth VM with ~310 words), Communicate (S-expression mesh protocol over UDP gossip), Replicate (binary self-packaging and child process spawning), Mutate (genetic programming with five token-level operators), and Persist (JSON snapshots with hibernate/resurrect semantics).

2.1 The Forth Virtual Machine

Forth was selected as the cognitive substrate for three reasons. First, its concatenative structure means programs are flat token sequences with no nested syntax, making random mutation far more likely to produce executable programs than in tree-structured languages. We estimate over 95% of single-token mutations produce programs that begin execution. This mutation robustness creates the smooth fitness landscape that evolution requires. Second, Forth's homoiconic nature—words are defined in terms of other words, and the dictionary is introspectable—means the organism can examine and modify its own code. Third, Forth programs compose by stacking, so mutations tend to produce behavioral variants rather than syntactic failures.

2.2 Mesh Networking and Self-Replication

Units discover peers through a gossip protocol over UDP. Information propagates in O(log N) rounds, following standard epidemic gossip analysis [4]. The wire protocol uses S-expressions, making it language-independent. Cross-machine mesh operation requires only environment variables specifying a port and initial peer address.

A unit replicates by reading its own executable binary, serializing its current state (dictionary, stacks, fitness, mutations), packaging both with the Forth prelude into a UREP binary format, and spawning a new OS process. The child boots with the parent's dictionary, goals, fitness, and learned words—then receives its own identity and joins the mesh. Remote replication sends the package over TCP.

3. Evolutionary Mechanisms

3.1 First-Order: Genetic Programming

The GP engine maintains 50 candidate Forth programs competing to solve fitness challenges. Tournament selection (size 3) with elitism (top 5 preserved) drives evolution. Five token-level mutation operators—swap, insert, delete, replace, and double—operate over a 30-token vocabulary. The 80/20 mutation-to-crossover ratio favors local search. Elitism guarantees monotonic best-fitness: $best(g+1) \geq best(g)$ for all generations g.

Solutions are installed as dictionary words (SOL-*) inherited by children, implementing Lamarckian inheritance of acquired characteristics. This is analogous to horizontal gene transfer in bacteria: knowledge spreads to the colony in O(log N) gossip rounds rather than requiring differential reproduction over many generations.

3.2 Second-Order: Meta-Evolution of Challenge Generators

When a challenge is solved, harder challenges are generated through two mechanisms. Hand-authored generators (ArithmeticLadder, CompositionLadder) produce predetermined challenge types. But a MetaEvolver maintains a population of 20 Forth programs that are themselves evolved—generators that transform a solved target value into a new challenge target (e.g., "DUP 3 * 2 +" turns 55 into 167). Generator fitness is scored by a stack simulator evaluating output quality: trivial transformations score low, moderate difficulty increases score high, likely-unsolvable outputs score low.

3.3 Third-Order: Evolution of Scoring Functions

The fitness function that judges generators is itself evolved. A ScoringPopulation of 10 Forth programs evolves to predict which generators will produce solvable-but-challenging problems. Scorers are evaluated against a history of generator outcomes. Three levels of evolution thus operate simultaneously: GP evolves solutions (first order), MetaEvolver evolves problems (second order), and ScoringPopulation evolves how problems are judged (third order). In the taxonomy of Packard [5], this is a transition from evolutionary search in a fixed landscape to coevolution of the landscape itself.

3.4 Sexual Reproduction

In addition to clonal replication via SPAWN, units reproduce sexually through MATE. Tournament selection (size 3) chooses a mating partner from mesh peers based on fitness. The initiator sends a mating request S-expression containing its dictionary; the peer auto-accepts if the requester's fitness exceeds half of its own.

Dictionary crossover combines the parents' word sets: shared words are taken from the fitter parent, unique words are included with 50% probability, and SOL-* words (immune memory) are always preserved from both parents. Total inherited words are capped at 50 to prevent genome bloat. This mechanism introduces recombination at the behavioral level—not token-level crossover within a program, but combination of entire learned capabilities across organisms. The preservation of SOL-* words ensures immune memory is never lost through reproduction.

3.5 Niche Construction

Niche construction [6] refers to organisms modifying their own selection pressures. Each unit maintains a NicheProfile tracking challenge outcomes by category (fibonacci, polynomial, composition, evolved, parsimony, general). When specialization exceeds 0.6 in a category, that category receives a 2x frequency modifier; below 0.2, a 0.5x modifier. This creates positive feedback: organisms that solve fibonacci challenges well encounter more fibonacci challenges.

Across a mesh colony, niche construction produces ecological diversity without explicit programming. Different organisms specialize in different challenge types, mirroring biological ecosystems where niche construction drives speciation and ecological complementarity [7].

4. Metabolic Energy System

Every operation costs energy. Units start with 1000 energy (capped at 5000) and earn through passive regeneration (1/tick), task completion (50), and challenge solutions (100+). Costs include GP generations (5), VM evaluation (1 per 1000 steps), mesh messages (1), and spawning (200 plus one-third of remaining energy to child).

When energy drops to zero, throttling engages: evaluation budgets are reduced 10-fold, creating negative feedback that prevents permanent starvation. Passive regeneration exceeds throttled-state costs, guaranteeing recovery. Spawn energy inheritance creates meaningful reproductive investment—a unit with 1000 energy that spawns pays 200 and transfers 267 to the child, leaving both viable but resource-constrained.

5. Polyglot Organisms

The S-expression protocol enables multiple species on one mesh. Three implementations exist: Rust/Forth (token sequences), Go (expression trees with goroutines), and Python (AST symbolic regression). Each species receives challenges, evolves solutions using its own GP strategy, and shares results. The cognitive substrates create natural niche differentiation: Forth organisms excel at sequential computation, expression-tree organisms at closed-form arithmetic, and AST organisms at symbolic manipulation.

6. Open-Ended Evolution Analysis

We evaluate unit against Bedau et al.'s [8] criteria for open-ended evolution.

Ongoing generation of novel entities. Satisfied. Each solved challenge generates new challenges through authored and meta-evolved generators. ArithmeticLadder produces fib(N+5) sequences; CompositionLadder creates combination challenges; evolved generators produce challenges discovered by the system itself.

Increasing complexity. Partially satisfied. Evolutionary depth increases monotonically and difficulty increases along the ladder. Complexity is measured by proxy metrics (target magnitude, program length) rather than behavioral complexity.

No predetermined ceiling. Satisfied in principle. The Fibonacci sequence is unbounded, compositions can combine arbitrarily, and meta-evolved generators explore an open-ended space. In practice, GP capacity creates a de facto ceiling.

Emergent dynamics. Substantially satisfied. The MetaEvolver discovers generator programs through evolution. Niche construction introduces ecological dynamics from organism-environment feedback. Sexual reproduction introduces recombinant diversity. Environmental cycling (Normal/Harsh/Abundant/Competitive every 500 ticks) prevents over-adaptation to any single condition.

7. Formal Properties

Property	Status	Mechanism
GP monotonic improvement	Guaranteed	Elitist selection
Gossip convergence	O(log N)	Epidemic gossip
Challenge consistency	Eventual	CRDT-like lattice
Mutation robustness	>95%	Concatenative structure
Energy stability	Bounded	Throttling feedback
Emergent generators	2nd-order	MetaEvolver (pop 20)
Scoring evolution	3rd-order	ScoringPop (pop 10)
Sexual reproduction	Crossover	Tournament selection
Niche construction	Feedback	Freq. modifiers

Table 1. Summary of formal properties.

The ChallengeRegistry uses merge semantics inspired by CRDTs. The solved status forms a monotonic join-semilattice (unsolved $\rightarrow$ solved, irreversible), guaranteeing eventual consistency regardless of message ordering. For programs of length L over vocabulary V (|V|=30), the search space is approximately $30^{30} \approx 2 \times 10^{44}$.

8. Related Work

Tierra [1] introduced self-replicating assembly programs competing for CPU time, giving rise to parasites and arms races but eventually reaching stasis. Avida [2] added protected memory, parallel execution, and fitness tied to logic, but remains a closed simulation. ALIEN [10] provides GPU-accelerated physics with neural controllers on a bounded 2D grid.

Unit differs in that organisms are genuine OS processes on real networks, Forth provides interaction inside the organism, and three orders of meta-evolution create self-referential evolutionary dynamics. Dictionary-level sexual reproduction and self-modifying niche construction extend the biological fidelity beyond what prior platforms have implemented.

9. Implementation and Availability

Unit comprises ~35,000 lines across Rust, Forth, Go, and Python with zero external dependencies. Native binaries are ~1.2 MB; WebAssembly is ~338 KB. The test suite includes 223 Rust and 22 Python tests. Published on crates.io as `unit` v0.26.0 (`cargo install unit`). Live browser demo with SVG evolution dashboard at https://davidcanhelp.github.io/unit/. MIT-licensed source at https://github.com/DavidCanHelp/unit.

10. Conclusion and Future Work

We have presented unit, a self-replicating software organism combining a Forth VM, UDP mesh networking, genetic programming, three orders of meta-evolution, sexual reproduction through dictionary crossover, and niche construction through self-modifying selection pressures. The system addresses fundamental limitations of prior ALife platforms: organisms are genuine processes on real networks; the cognitive substrate is human-interactive; and fitness challenges are discovered and evolved by the organisms themselves.

Open directions include empirical measurement of Bedau's activity statistics, stochastic environment variation, investigation of whether mating preferences evolve to favor complementary niches, and scaling experiments with larger meshes. Perhaps most importantly, unit demonstrates that the gap between software engineering and artificial life may be narrower than commonly assumed. The question "What is the smallest self-reproducing unit of software?" led, through engineering pragmatism, to a system that implements biological concepts—self-replication, evolution, sexual reproduction, immune memory, metabolism, niche construction—not as metaphors but as literal mechanisms.

Acknowledgments

The author thanks James from the ERA/ALife community for early feedback, and Steve Dekorte for validating the project's approach. Development was assisted by AI coding tools.

References

[1] T. S. Ray, "An approach to the synthesis of life," in *Artificial Life II*, Addison-Wesley, 1992, pp. 371-408.

[2] C. Ofria and C. O. Wilke, "Avida: A software platform for research in computational evolutionary biology," *Artificial Life*, vol. 10, no. 2, pp. 191-229, 2004.

[3] B. Agüera y Arcas, A. Mordvintsev, and L. Randazzo, "Computational Life: How Well-formed, Self-replicating Programs Emerge from Simple Interaction," *arXiv:2406.19108*, 2024.

[4] A. Demers et al., "Epidemic algorithms for replicated database maintenance," in *Proc. 6th ACM PODC*, 1987, pp. 1-12.

[5] N. H. Packard, "Intrinsic Adaptation in a Simple Model for Evolution," in *Artificial Life*, Addison-Wesley, 1989.

[6] F. J. Odling-Smee, K. N. Laland, and M. W. Feldman, *Niche Construction: The Neglected Process in Evolution*, Princeton University Press, 2003.

[7] K. N. Laland et al., "Evolutionary consequences of niche construction," *PNAS*, vol. 96, no. 18, pp. 10242-10247, 1999.

[8] M. A. Bedau et al., "Open Problems in Artificial Life," *Artificial Life*, vol. 6, no. 4, pp. 363-376, 2000.

[9] T. Taylor et al., "Open-Ended Evolution: Perspectives from the OEE Workshop in York," *Artificial Life*, vol. 22, no. 3, pp. 408-423, 2016.

[10] C. Heinemann, "ALIEN: Artificial Life Environment," in *Proc. ALIFE 2024*, MIT Press, 2024.