

The Reality Code Engine: One Lossless Form for Native, Verified, Low-Energy Programming

David Charlot, Open Interface Engineering

Research paper · OpenIE research.openie.dev

<https://research.openie.dev/papers/rce/> · PDF <https://research.openie.dev/pdfs/rce.pdf>

Measurement. Research paper, not a journal final. Handler figures are `est_j`. Track B figures are `reported_j` from a RAPL package counter less a stated idle baseline. `measured_j` is reserved for a wall meter and is empty. No FPGA board result in this paper. No fabricated citations.

A programming system in which a program is written once as a state machine and every target language is generated from it, with measurements of round trips, refusal of malformed input and repair of lifted code.

Version 1.21, 24 October 2026. Specification: [current draft \(draft 0.9\)](#), from [openIE-dev/rce](#); the counts in the text below are at the commits they name. Data and code: [research.openie.dev/data/](#). Playground (reference checker): [openie-dev.github.io/rce](#). Engine demo: [/demos/rce-engine/](#). LaTeX PDF: [/pdfs/latex/rce.pdf](#) ([source](#)).

Provenance of counts and reproducibility. Every count in this paper comes from [src/data/rce-facts.json](#), produced by `scripts/claim-counts.py` in [openIE-dev/rce](#) at the commit each figure or caption names; where a figure names an earlier commit than the text, it describes the repository at that commit. The specification document is draft 0.9; theorems T1 to T56 are stated, T42 to T56 (Appendix F) stated and unproved; there is one language and one text (Section 3.7), and a failed call restores its `&mut` arguments (F.4.4), which is what theorem T33 proves. Two Lean counts exist and are not interchangeable: the source holds 1,116 theorem and lemma declarations at `e416fd0`, while the audit (`scripts/lean-audit.sh`) counts every theorem constant in the kernel environment, including constants the elaborator generates, and reads 4,877 at `e416fd0` over 58 files with no sorry and standard axioms only; the 2,192 quoted in Section 3.7 is the audit count at engine commit `1eb5f02` over 31 modules. Neither number says anything about how many theorems are proved about the Rust evaluator.

- **Reproducible from the public repositories:** the Lean build and axiom audit, the specification check (grammar tables and example corpora), the reference checker and its tests, the canonical JSON (JavaScript Object Notation) vectors, and the registered runs of Sections 3.8 and 8.
- **Not yet reproducible from the public repositories:** the 10,000-Form ecosystem run (`bench/ecosystem` is not public), the Kani results, the 589 engine tests, the repair rates of Section 6 and any Defects4J number. They are reported as the authors' own runs and should be read that way until the sources and seeds are published.
- **Falsifier RCE-F0.** See Section 2.3.

Abstract

This paper describes the Reality Code Engine (RCE), a system in which a program is written once as a state machine, called a Form, and code in other languages is generated from it. It reports what was measured on 10,000 Forms and on public bug datasets, and states how the claims could be shown false.

In plain words

1. A program is written once, in one precise form, and the computer produces the same program in many languages from it.
2. Because everything a program can do is declared in its form, every input has a declared outcome, a step or a recorded refusal, rather than an undefined one that is found and fixed later.
3. A fix that passes the checks is saved and reused, so the same problem does not cost time or energy twice.

1. Thesis

Autonomous computer automation is a design space with many points in it. Every approach sits somewhere along five continuous axes: the energy spent per correct result; the entropy the system must resolve at run time, and with it how deterministic its output is; the strength of the verification that accepts a result; how far each accepted result is reused and compounds; and how much the approach relies on a model. A large language model (LLM), a statistical text model, that writes each change from the prompt sits near one corner of that space: it relies on the model for every task, draws a fresh sample from a high-entropy distribution each time, and pays for generation again on every task. Retrieval, rules, synthesis, verification and caching occupy many other points, and real systems combine them.

The Reality Code Engine (RCE) is one point in the space, and this paper describes and measures it. RCE models a program as a database functional state machine. Lopes and colleagues measured that about 70% of the code on GitHub consists of clones of earlier files [33], so RCE first tries to retrieve an existing fit, adapt it and check it, and calls a model only when nothing cheaper closes the task.

The core claim. RCE is a complete code ecosystem in which programs are authored natively from intent. Intent is written once, as a Form: a state, a set of events and a pure, total transition. The Form is a lossless encoding of that intent. Every target (statechart, Rust, Python, XState, SCXML, WASM) is a projection of the Form, and each projection replays the Form's trace byte for byte. Every state a program can be in is declared in its Form, and every input has a declared outcome: a step that takes it, or a refusal by its contract that leaves the state unchanged and is recorded in the log (T8). A program in the ecosystem therefore cannot drift from its intent the way a program in a common language drifts from its comments, its tests

or its other implementations, and it needs no after-the-fact diffing or error tracking. Every fit that passes the gate is stored and reused, so the ecosystem compounds as it grows (§2.4). Code written in other languages is one door in: it enters by lift (§6).

Every point in the space faces the same gate. The gate is correctness: a task's completeness predicate $C(z)$, here the task's tests, a proof or an agreement trace. Among runs that pass the gate, systems are ranked by joules to completion, J^* . Correctness is a constraint, not a score.

RCE's mechanisms, called gears, run in order of cost, and the first gear that passes the gate closes the task.

Energy figures are labeled by source. `est_j` is an analytical estimate. `reported_j` is an operating-system or on-chip counter. `measured_j` is reserved for a wall meter and is empty in every result here.

2. The design space of autonomous coding

 One sentence in. One floor out. Human intent is high entropy; a funnel of conversions from intent to spec to 626 surface languages to an intermediate representation to machine code loses information at each boundary; all code reaches a few machine shapes; RCE writes the meaning once in one canonical Form, treats every surface as a projection, and lets the cheapest correct form win.

Figure 1. Intent entropy and the machine floor. A [PNG copy](#) is provided. Sources: Shannon (1951); Lopes et al. (2017); the 626-language count is an unpublished survey by the OpenIE authors and supports no claim in this paper. The Python and C results are language specification behavior; C signed overflow is undefined behavior. [Open full size](#) · [PNG](#)

Intent is high entropy. "Make the checkout faster" carries many meanings: shorter page load, fewer steps, faster card approval, a cached cart, or faster on one device class. Context is implied and never written. Shannon measured printed English at about 0.6 to 1.3 bits per letter [1]. The letters are predictable. The meaning is not.

Every conversion loses information. Intent becomes a specification, and context is lost. The specification becomes one of hundreds of surface languages, and alternatives and edge cases are lost. The source becomes an intermediate representation (IR), and names, style and the author's reasons are lost. The IR becomes machine code, and types become sizes and loops become jumps.

The floor is low entropy. Every program ends as loads, stores, arithmetic and jumps. The destination has room for only a few shapes, so code bends toward them. Convergence is by design.

Boundaries are where meaning diverges. The same line gives different answers at the edges. In Python, $-3 \% 2$ is 1 and $-7 / 2$ is -3.5. In C, $-3 \% 2$ is -1, $-7 / 2$ is -3, and signed overflow is undefined. A system that regenerates code at every boundary guesses at every boundary.

The axes, stated as quantities.

Axis	One end	Other end	Where RCE sits
Energy per correct result	Paid again in full on every task	Paid once per new case, then the handler floor	At the floor on covered tasks
Entropy resolved at run time, and determinism	The full distribution of the prompt, a fresh sample each time	Only the residual no stored fit covers, one deterministic output	The residual only; every projection deterministic
Verification strength	Output accepted on inspection	Tests, coverage and a proof certificate	The full gate on every result
Reuse and compounding	Each result used once	Each accepted fit stored and reused	Every fit that passes the gate is stored
Model reliance	A model on every task	No model	A model only as the last gear

Approaches differ by degree on each axis, and the axes interact: reuse lowers energy and the entropy left to resolve, and strong verification is what makes reuse safe. RCE writes the meaning once, in one canonical Form. Every surface language is a projection of that Form, checked to match it. Among equivalent forms, the cheapest correct one runs.

Figure 2 draws the five axes and places the main classes of approach on them.

 Radar chart with five axes: energy per correct result, determinism, verification strength, reuse and compounding, and independence from models. Five approach classes are drawn as polygons. A model on every task sits near the centre on all axes. An agent loop gated by tests is higher on verification. A model with retrieval and caching is higher on reuse. Classical synthesis and repair is high on determinism and model independence. RCE, marked with stars, sits at or near the outer ring on all five axes. Placements are qualitative.

Figure 2. The design space of autonomous coding. Qualitative diagram: each class of approach is placed by the paper's reading of Section 2 on ordinal levels, not by measurement. Model reliance is drawn as independence from models so that outward is better on every axis. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

2.1 Programming is coding theory

The thesis of this section is that programming can be read as coding theory. A programmer converts intent along a gradient between two entropic regimes. Intent is high entropy, a distribution over meanings. Code is low entropy, one deterministic transition per state. Every

step of software work, from a requirement to a register, is a step of that conversion. The literature has said so in pieces for seventy years, in many languages. This section assembles the pieces into one argument and places each RCE gear on it.

The measure. Shannon defined the entropy of a source and the capacity of a channel [40], and measured printed English at about 0.6 to 1.3 bits per letter [1]. Kolmogorov, writing in Russian in *Problemy Peredachi Informatsii* in 1965, gave three approaches to the quantity of information and made the algorithmic one primary: the information in an object is the length of the shortest program that produces it [41]. Solomonoff had reached the same object as a theory of inductive inference [42], and Chaitin showed that program size obeys identities formally identical to Shannon's [43]. Rissanen turned the idea into a rule for choosing models, the minimum description length [44]. Together they say that a program is a code word: a description of a behavior, and the shortest correct one is the floor. RCE's extraction rule, keep the least-joule fit among equivalent fits (Proposition 3), is that rule with joules in place of bits.

Code sits in the low-entropy regime, and people measured it. Hindle, Barr, Su, Gabel and Devanbu found that code is "very repetitive, and in fact even more so than natural languages," at lower cross-entropy under the same n-gram models [35]. Gabel and Su found that most lines of a project already exist elsewhere [34], and Lopes et al. found that most files on GitHub are clones [33]. Tu, Su and Devanbu found that code is also local: what was written nearby predicts what comes next [46]. Ray and colleagues found that buggy lines are less natural, with higher entropy, and that the entropy falls when the bug is fixed [45]. In 2025 Yang, Kolak, Hellendoorn and Martins used the entropy change a patch induces to rank repair candidates and to separate correct patches from overfitting ones [51]. Allamanis, Barr, Devanbu and Sutton surveyed the field under the name naturalness [47]. The result is not only an English-language one. Prem Devanbu stated it as a keynote at the India Software Engineering Conference in 2013 [50]. Sridhara, Sinha and Mani at IBM Research India measured the natural-language artifacts around code at the same venue and found that commit messages are the most regular and Stack Overflow prose behaves like ordinary English [49]. In China, the *Journal of Software* (软件学报) published a review of code naturalness research in 2022, which states the cross-entropy measure and the finding that defective code has higher entropy [48]. In Japan, Doi, Higo, Arima, Shimonaka and Kusumoto at Osaka University used n-gram naturalness, first in a Japanese-language IEICE technical report and then at ICPC 2018, to tell machine-generated source files from human-written ones [73, 74]. Generated code sits even lower on the entropy scale than human code, because a generator is a fixed conversion. In Brazil, Arrebola's doctoral thesis built code suggestion on n-gram and topic models segmented by project and maintainer, and evaluated it by cross-entropy, the bits needed to encode the next token [75]. The two regimes are visible in one repository: the prose around the code is high entropy, and the code is low.

The measure matters. Febres, Jaffé and Gershenson, working from Venezuela and Mexico, compared English, Spanish and program text by vocabulary diversity per message length and found the program text more diverse [76]. Rahman, Palani and Rigby found that much of the n-gram predictability of code comes from syntax tokens, and that code is still repetitive, less

than first reported, once those are removed, with API usage the most repetitive part [77]. Both results sharpen the claim rather than reverse it. Diversity of symbols is not predictability, and the low-entropy regime is a property of the conditional distribution of the next step, which is what a deterministic transition fixes. RCE's G0 keys on shapes, not on surface tokens, for that reason.

Intent sits in the high-entropy regime. Zave and Jackson showed that a requirement is a statement about the environment, a specification is a statement at the interface, and the gap between them is filled only by domain knowledge that is often unwritten [64]. Brooks split the difficulty of software into essence and accident: the essence is the conceptual construct, and most of it is specification and design, not representation [61]. Naur argued that a program is the visible part of a theory held by its programmers, and that the theory is not fully recoverable from the text [62]. Read through Shannon, all three say the same thing. Intent carries more uncertainty than any one program, and writing the program is the act that removes it. What remains in a programmer's head after the code is written is the part of the intent the code did not encode.

The conversion is a refinement, and refinement is a narrowing. Hoare gave programs a logic in which a program is correct relative to a pre- and postcondition [54]. Wirth taught stepwise refinement: decompose a task into instructions by successive decisions, each one fixing a choice the specification left open [53]. Dijkstra's guarded commands made the open choices explicit as nondeterminacy, and derived programs from specifications by removing it [52]. In Munich, Bauer and Wössner's CIP project built program development as a sequence of correctness-preserving transformations from specification to program [58]. In France, Abrial's B method and then Event-B carried refinement to industrial scale, with proof obligations at each step [55, 56]. In Finland, Back and von Wright gave refinement its lattice-theoretic calculus [57]. In each, a specification denotes a set of acceptable behaviors and each refinement step shrinks the set, until one deterministic behavior remains. That is entropy reduction with a proof at every step. RCE's Form is the last element of that chain: a total, deterministic transition δ over database states, with every choice resolved (§3.2).

Synthesis is search that narrows a hypothesis space. Gulwani's FlashFill learns a string program from a few input-output examples by representing the set of all consistent programs and intersecting it with each new example [59]. Gulwani, Polozov and Singh frame program synthesis in general as search through a space of programs constrained by intent [60], and Solar-Lezama's sketching fills holes in a partial program with a solver [27]. Each example or constraint is a measurement that removes hypotheses. Shannon's measure counts how many such measurements are needed. RCE's G2 is that gear.

What the conversion produces has one shape. Gurevich proved that every sequential algorithm is captured by an Abstract State Machine at its own level of abstraction [2]. Codd made the state a relation [5], and event sourcing makes the state the fold of an append-only log [14]. Under Curry and Howard, a proof is a program and a proposition is a type [18, 72], so

a verified fit is a proof of its specification. These are the reasons the low-entropy regime has few shapes: the destination is a state machine over a database, whatever the surface language.

Low entropy decays without a gate. Lehman observed over decades of large systems that a program in use must keep changing, and that its complexity grows unless work is done to reduce it [63]. Torres, Baltes, Treude and Wagner measured structural and textual entropy over the commit histories of 25 projects and found the two rise together [78]. Entropy flows back into code from the environment. A system that keeps code in the low-entropy regime needs a gate on every change, and RCE's gate is correctness (§4.3).

Compilers already do source coding. Franz and Kistler shipped programs as compressed abstract syntax trees and generated native code on load, which is source coding of a program over its own grammar [67]. Fraser inferred statistical models of code from a corpus to compress it [68]. Futamura showed in 1971, in the Japanese journal *Systems, Computers, Controls*, that specializing an interpreter to a program yields a compiled program, and that specializing the specializer yields a compiler [70]. Ershov in Novosibirsk developed the same idea as mixed computation [71]. The Cousots' abstract interpretation computes a sound, lower-information description of every behavior of a program [69]. The analogy to channel coding is this paper's reading, not a cited theorem: types, tests and proofs add checked redundancy so that a wrong program is detected before it runs, as a parity check detects a corrupted word.

Erasure is where the joules go. Landauer showed that a logically irreversible step, the erasure of one bit, dissipates at least $k_B T \ln 2$ [65]. Bennett showed that computation can in principle be made logically reversible, so the erasure, not the computation, carries the bound [66]. Converting high-entropy intent to one low-entropy program discards every alternative the intent allowed. That discard is an erasure of information about the alternatives, and it has a thermodynamic floor. RCE pays the discard once per new case and stores the result, so a repeat case erases nothing new.

What the surface costs, measured. The naturalness result has two corrections that this paper now carries, and a third finding that bears on the name-free Form. First, the raw entropy gap between code and prose shrinks substantially once syntax tokens are removed [77], and Casalnuovo, Lee, Morgan and Devanbu found that part of what remains is a familiarity effect: in top-starred Java and Python projects the original code had lower surprisal than meaning-preserving rewrites, and 50 human raters agreed with the language model's preference 61.9% of the time [132]. What reads as natural is what the reader has seen most, so a canonical Form moves the equilibrium of that loop rather than escaping it. Second, the only quantities that track measured comprehension load are textual size, vocabulary size and data-flow dependency reach: in the one study that tested code metrics against brain activation, those three correlated at Kendall $\tau \approx 0.4$ while McCabe cyclomatic complexity correlated with nothing ($\tau \approx 0.09$) [133]; across 121 readability metrics and 444 evaluations, no single metric reached $|\tau| > 0.16$ with measured understandability [134]; and pretty-printed against disrupted layout showed no fMRI difference [135], consistent with the finding that formatting is 13 to 15% of the tokens of brace languages and can be stripped for ten language models with no

significant change in accuracy [136]. Third, names are a low-agreement, high-dependency channel: two developers choose the same name with median probability 6.9% [137], comments co-evolve with the code they describe in 7 to 13% of cases [138], and yet frontier models lose 11 to 29 points of summarisation accuracy and up to 24 points of execution-prediction accuracy when identifiers are removed or replaced [139]. Humans, by contrast, do not process identifiers as language at the neural level: swapping English identifiers for Japanese changed the language-system response by $\Delta\beta = 0.03$ [140]. The consequence for RCE is stated in §3.8: positions stay in the hash (T7) and names stay outside it, but the side table that holds them must carry the *concept* a name encodes, because the only authors that will matter read through names and the Form has not yet shown that a model can reason from a nameless one.

2.2 How RCE converts, and where each gear sits

Read as coding theory, a task arrives as a message whose remaining uncertainty is what no stored fit already resolves. Each gear removes uncertainty at a different price, and the gears are ordered by that price (§4.2).

Gear	What it does to the uncertainty	Lineage in the argument above
G0 Lookup	Recognizes that the intent was already encoded; resolves it by retrieval, at the handler floor	Repetitiveness and clones [33, 34, 35]; localness [46]
G1 Rules	Applies a known guarded rewrite; resolves a recurring residual with no search	Correctness-preserving transformation [52, 58]; refinement with proof obligations [55, 56]
G2 Synthesis	Searches a hypothesis space and narrows it by examples, constraints and a solver	Program synthesis as search [27, 59, 60]
G3 to G5 Models	Samples from a learned distribution over code when no lower gear closes the gap; a proposer only	Cross-entropy models of code [35, 47]
Gate	Accepts a candidate only with tests, coverage and a proof certificate	Hoare logic [54]; Curry-Howard [18, 72]
Promotion	Stores the accepted fit, so the same residual costs G0 next time	Minimum description length [44]; Landauer erasure paid once [65]

Positions on the axes of Section 2 differ in where the conversion is paid. An approach that samples the full distribution of the prompt on every task pays the high-entropy side again each time. An approach that stores and reuses accepted fits pays only for the residual, and only once. Most systems sit between those ends, and RCE sits at the reuse end.

2.3 The ecosystem at large N

The claim of §1 is measured directly. The benchmark was preregistered before the run ([bench/ecosystem/PREREG.json](#)). A seeded generator writes 10,000 Forms, each a source, a rule whose body is a random arithmetic term of depth up to 3 over eight operators, and a sink. The engine is frozen at commit a079967, and no model is called.

Property	What is checked per Form	N	Holds	Model calls
Lossless encoding	Form IR to canonical JSON to Form IR to canonical JSON is the identity	10,000	10,000	0
Deterministic projection	Two compiles give the same bytes; XState and SCXML projections of the original and of the round tripped Form are byte identical	10,000	10,000	0
Faithful projection, Python	The emitted Python module prints the reference trace byte for byte on 40 seeded events	10,000	9,999	0
Gate at entry, inputs	Five malformed payload kinds (wrong type, null, missing field, empty, nested) all refused, no accepted outcome	10,000	10,000	0
Gate at entry, specifications	A specification with a flow to an undeclared node is refused before compile	10,000	10,000	0

Figure 3 shows the same counts.

Horizontal bar chart of five checks on 10,000 Forms. Lossless encoding, deterministic projection, gate at entry on malformed inputs and gate at entry on specifications each hold on 10,000 of 10,000. The Python trace replay holds on 9,999 of 10,000. Zero model calls.

Figure 3. The ecosystem at large N. Data figure: the five checks of the table above on 10,000 preregistered Forms, frozen engine at commit a079967, 0 model calls, with Wilson 95% intervals computed from the counts. Source: [bench/ecosystem](#). A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

RCE-F0 and the Python miss. Section 9 states RCE-F0 as: on a preregistered run of at least 10,000 Forms, any projection prints a trace that differs from the Form's reference trace, or any malformed input is accepted. This run is preregistered, has 10,000 Forms, and one emitted Python module did not print the reference trace, so the observation as written meets the condition of RCE-F0. The paper reports the miss and the regression test below, and it does not give a root cause. Read the text of RCE-F0 as stated: the claim it defeats is that projections are faithful, and the Python projection of this run did not meet that claim on one Form. Whether the miss comes from the emitter or from the harness is open here. The fact that the same Form replays identically under 32 hash seeds does not settle it.

The one Python miss is counted as a miss; the same Form and events replay identically under 32 hash seeds in the regression test `tests/form_ecosystem_regress.rs`. Energy here is `est_j` only; the whole run took 393 s of central processing unit (CPU) time on one core. The generated Forms are small, so this table measures the encoding, the projections and the gate. Lifted Forms from real programs are measured under the protocol of §8.7.

2.4 Gated, verified recursive self-improvement

Good (1965) [111] defined the ultraintelligent machine as one that could design better machines, so that improvement would recurse. Yudkowsky (2013) [113] cast the question as one of returns: whether each unit of improvement makes the next unit cheaper or dearer. Schmidhuber's Gödel machine (2007) [112] gave a formal answer with a guarantee: a program rewrites any part of itself, including its own search, only after proving that the rewrite improves its expected utility. Löb's theorem (1955) [114] is the obstacle a self-trusting prover meets, which Yudkowsky and Herreshoff (2013) [115] named the Löbian obstacle: a system cannot in general prove the soundness of a successor that uses its own proof system. Recent systems drop the proof requirement and select by measured score: STOP (Zelikman et al., 2023) [116] improves the scaffold that improves code, the Darwin Gödel Machine (Zhang et al., 2025) [117] keeps an archive of self-modified coding agents chosen by benchmark score, AlphaEvolve (Novikov et al., 2025) [118] evolves programs under automatic evaluators, and Voyager (Wang et al., 2023) [119] grows a library of verified skills that later tasks reuse.

RCE takes the structure of the Gödel machine and the library of Voyager, with two differences that avoid the obstacle for the objects it promotes. First, the object that improves is not the prover. RCE promotes handlers, rewrites and gears, and each is checked against a fixed, separately certified kernel: the Kani harnesses and the SMT encoding never change through promotion, so no step needs to trust a successor's proof system. Second, the gate is a proof of correctness at the boundary, not a score. A score can rise while an interface assumption breaks; a proof at the boundary cannot be passed by a change that breaks it.

Proposition R1 (monotone correctness). If every promoted change passes the check of its boundary, and the checks of the other boundaries are unchanged, then every property established by those checks before the promotion still holds after it.

Proposition R2 (compounding). Let a workload be a multiset of tasks and let a promoted handler close every task of its shape. Then the number of escalations in round t is nonincreasing in t , and a correction promoted before round 1 lowers the cumulative escalations of every horizon by the escalations it closes in each later round. In the engine demo's reference workload, cumulative escalations over eight rounds fall from 21 to 15 with one correction at the sinks, and reach zero in the last round in both cases, with every task closed correctly and no promotion refused.

Figure 4 draws the mechanism of Propositions R1 and R2 with the reference workload's totals.

Diagram of the RCE stack as six levels from intent at the top through Form, gate, projections and runtime to hardware and energy. A correction promoted at the runtime level after passing its boundary check is shown with arrows rising through every level above it. A small bar chart beside it gives cumulative escalations over eight rounds on the engine demo reference workload: 21 with no correction and 15 with one correction at the sinks.

Figure 4. Corrections compound upward. Diagram of the stack, with a data inset: cumulative escalations over eight rounds of the engine demo's reference workload, 21 with no correction and 15 with one correction at the sinks (Proposition R2; [RCE engine demo](#)). A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

Proposition R3 (falling cost). Since a promoted handler is executed in place of a model call, and its cost is the op count of the handler priced by the substrate table, the energy of a round falls toward the energy of the handlers alone as escalations fall toward zero.

Futamura's projections (1971, republished 1999) [120] and Turchin's supercompilation (1986) [121] give the limit of this process. Specializing an interpreter to a program yields a compiled program; specializing the specializer yields a compiler. RCE's promotions specialize the general escalation path to the shapes that actually occur, so the engine converges toward the residual program of its workload: the part of the computation that a model or a person must still supply. Recursive self-improvement under a gate is that convergence, made with checked steps. The Compounding layer of the [RCE engine demo](#) runs this workload in the browser.

3. The Form

3.1 The strongest formal statement of "same form"

The scope of the theorem is stated first. Gurevich's sequential postulates cover sequential algorithms; parallel and interactive algorithms need the further postulates of Blass and Gurevich [3]. Blass, Dershowitz and Gurevich argue that no single equivalence relation on algorithms fits every purpose [79], and Moschovakis and Vardi set out the debate over what an algorithm is [80, 81]. A web handler that does input and output sits outside the sequential theorem. The Form handles interaction the way the interactive line does: each input is an event in E , and the state machine steps on it. So "one Form" is a modelling choice grounded in the sequential theorem and extended by convention, with production precedents in the Elm architecture [83], durable workflow event histories [84] and stream semantics such as Flo [82]. The propositions of Section 3.4 are stated for that choice.

Gurevich's Abstract State Machine (ASM) theorem is the strongest formal statement that every algorithm has one shape [2]. An abstract state machine is a machine whose states are first-order structures and whose step is a finite set of guarded updates. Gurevich states three postulates. Sequential Time: an algorithm has states, initial states and a one-step map τ . Abstract State: states are first-order structures over one fixed vocabulary, closed under isomorphism. Bounded Exploration: a finite set of terms decides each step. His Main Theorem

(Theorem 6.13) reads: “For every sequential algorithm A, there exists an equivalent sequential abstract state machine B.” Blass and Gurevich extend it to parallel algorithms [3]. Dershowitz and Gurevich add a fourth axiom, that initial operations are computable, and from it derive Church’s thesis [4].

3.2 Definitions

Definition 1 (the Form). A program is a tuple

$$P = (S, E, \delta, D, \lambda)$$

- $S = \Sigma_c \times \Sigma_d$, the state space: structures over a fixed vocabulary Σ , split into control regions Σ_c and data Σ_d . Σ_d holds the logical clock, the input count and the typed relations (in the language, also the declared regions). Each state is a database instance in Codd’s sense [5].
- E , the event alphabet, including the logical clock tick.
- $\delta : S \times E \rightarrow S \times O^*$, the transition: total, deterministic and pure, written as a finite set of guarded updates “if φ then $f(\bar{t}) := t_0$ ”, each step reading a bounded set of locations. It returns the next state and the outputs of the step. Write $\delta_1 = \pi_1 \circ \delta$ for its state component.
- D , the durable log: a sequence of entries $\{seq, event, prev_hash, state_hash, hash\}$, where $state_hash$ is the SHA-256 of the canonical JSON of the state after the step, $hash$ is the SHA-256 of the canonical JSON of the first four fields, and the first $prev_hash$ is 64 zeros. The log is a hash chain, so changing any past entry changes every later $hash$.
- $\lambda = \pi_2 \circ \delta : S \times E \rightarrow O^*$, the output map. A Mealy machine reads both arguments [6]; a Moore machine reads the state only [7].

Definition 2 (Form and fit). The *form* is the tuple shape with four invariants: δ is deterministic and pure; exploration is bounded; the vocabulary is fixed; state is the fold of the log,

$$state(D) = \text{foldl}(\delta_1, s_0, D).$$

A log may also carry each entry’s outcome, outputs and state. These are derived: they are not hashed, and they must equal replay (T1 and T3 of §3.7). Two kinds of rejection are distinct. In the engine profile every event is applied, and an input that fails its contract produces a reject output and an error row. In the language a refused event leaves the state unchanged (T8) and is recorded as refused.

The *fit* is the data that fills the shape: the vocabulary Σ , the alphabets E and O , the rule program Π that defines δ , the output rules that define λ , and the initial data s_0 . In Gurevich’s terms the fit is the vocabulary, the ASM program and the initial structures. In Codd’s terms Σ is the schema and s_0 is the instance. In product-line terms a fit is a feature selection [8]. In syntax-guided synthesis (SyGuS) terms a fit is a term from a grammar that meets a specification [9].

Definition 3 (fit equivalence). Two fits are equivalent when, for every event sequence L , their output sequences under λ are equal.

Definition 4 (fit language). A grammar whose terms denote fits. Programming languages, SCXML documents, XState configurations and .rce specifications are fit languages. A fit language is complete when every ASM program over Σ has a term in it.

Definition 5 (control projection). Split $\Sigma = \Sigma_c \cup \Sigma_d$ into finite control symbols and data relations. The statechart of P is the projection of δ onto Σ_c : states are values of Σ_c , transitions carry an event and a guard over Σ_d , and actions are updates to Σ_d . Harel's statecharts give the hierarchy and the orthogonal regions [10].

Definition 6 (fit e-graph). For a fit F and a set R of sound rewrites, the fit e-graph $EG(F, R)$ is the saturated e-graph of F 's rule terms under R [11, 12]. Each e-class holds fits equivalent by Definition 3.

Definition 7 (joule cost). $\text{cost}_J(t)$ is the estimated energy (est_j) to run term t once on a named substrate, weighted by the expected number of runs.

3.3 Why “database functional state machine”

A finite relational structure is a database instance, so a Gurevich state is a database and each step is a bounded transaction. *Database*: S is a relational structure. *Functional*: δ and λ are pure. *State machine*: runs are folds of δ over events. Moseley and Marks draw the same line in architecture [13]. Event sourcing draws it in application state: “Capture all changes to an application state as a sequence of events” [14]. Dedalus draws it in logic, where mutable state is a relation indexed by time [15].

3.4 Propositions

Proposition 1 (the Form captures every sequential algorithm). For every sequential algorithm A there is a Form instance P whose runs are the runs of A .

Proof sketch. Theorem 6.13 gives an ASM B with the states, initial states and one-step map of A . Set $E = \{\varepsilon\}$, $\delta(s, \varepsilon) = \tau_B(s)$, $D = (s_0, \varepsilon^n)$ and λ the identity. The fold after n ticks is $\tau_A^n(s_0)$. By Gurevich's Corollary 3.3, P and A have the same runs. ■

Proposition 1b (interactive programs). Extend Σ with a nullary input symbol set to e before each step; then $\delta(s, e) = \tau(s[in := e])$ is a sequential algorithm over the extended vocabulary, and Proposition 1 applies. The construction is ours.

Proposition 1c (computability). Under the Dershowitz and Gurevich axiom, every Form instance computes a Turing-computable function [4, 16, 17]. The Form adds no power. It gives every program in the class one shape.

Proposition 1d (proofs are fits). “For each proof of a given proposition, there is a program of the corresponding type” [18]. A type is a specification and a proof is a fit that meets it. The correctness gate is the existence of that proof or of a passing acceptance test.

Proposition 2 (one generator). One function $G : \text{Fit} \rightarrow \text{Program}$ generates a runnable program for every fit and reads no domain. *Basis:* every program is a Form instance, so G emits a typed store, an event type, compiled rules, a log with a fold and an output dispatcher. Scaffolders such as Rails generators, JHipster and Yeoman are G restricted to small fit languages. Product-line engineering starts from “the systematic discovery and exploitation of commonality across related software systems” [8]; the commonality is the Form.

Proposition 3 (least-energy extraction). Extraction with cost_J returns the least-joule term the extractor finds in the e-class of F . *Proof sketch.* The rewrites are sound, so every term in the e-class has F ’s observable behavior and passes the same gate. Extraction minimizes the cost, and the cost is joules. ■ This is Mixture of Limits applied to code: keep the minimal fit that does the task, and the rest is waste.

Proposition 4 (replay determinism). If δ reads no wall clock, orders updates without names and logs every input, two replays of one D give identical state hashes at every step.

Proposition 5 (statechart projection is sound). Project each state of a run onto Σ_c . The result is a run of the emitted statechart under the same events.

Proposition 6 (coordination budget). Rules tagged monotone run on replicas without coordination; only non-monotone rules need a coordination point. *Basis:* the CALM theorem, “the programs that have consistent, coordination-free distributed implementations are exactly the programs that can be expressed in monotonic logic” [19].

The tag is derived from the rule, never declared, and the IR check refuses a Form whose stored tag differs from the derived one (specification, Appendix D.5):

Rule has	Tag	Reason
no resolve, read, join or scan	monotone	its writes only add keys and grow counters
a resolve	not monotone	it deletes escalation rows
a read of its own sink	not monotone	read, modify, write
a keyed join	not monotone	the joined row may be absent now and present later, or overwritten
a scan	not monotone	it observes the whole relation, including what is absent

The tag is sound and not complete: a join against an insert-only relation would be monotone, but the profile does not prove that a relation is insert-only, so the tag does not claim it. A Form is coordination-free when every rule is monotone. Every read, join and scan sees the state at the event's entry, so the post-state of an input event does not depend on the order in which rules for different sinks run; only the output order does, and it is fixed by content hash.

Proposition 7 (intelligence as a joule derivative). Let X count tasks through the gate and J the joules spent. $\iota = dX/dJ$ is read in hindsight from D , which records every event and its `est_j`. Extraction under `costJ` holds the gate fixed and does not raise J , so at equal X it does not lower ι .

3.5 Graph, e-graph and statechart

Structure	Part of the Form	What it decides
Statechart	Guards of δ over Σ_c	Which transition fires
Dataflow graph	Read and write dependencies of δ and λ (Kahn process networks [20])	What runs, and in what order
E-graph	The equivalence class of the fit	Which equivalent fit runs

The statechart and the graph describe one fit. The e-graph describes every fit equivalent to it. Extraction chooses among them by joules.

3.6 One ecosystem, lossless, and the universality thesis

Lossless by construction. Shannon's source coding theorem fixes the floor for a lossless code at the entropy of the source [40]. A Form is a lossless code for intent in a narrow sense that can be checked: the canonical serialization of the Form is injective, so two Forms that differ in any state, event or transition have different shape hashes, and decoding the serialization gives back the same Form. Each projection to a target is a function of the Form alone. Bidirectional transformations make the round trip precise. Foster, Greenwald, Moore, Pierce and Schmitt define a lens as a pair of get and put functions with round trip conditions that make a view and its source agree [122], Hofmann, Pierce and Wagner extend the conditions to symmetric lenses [123], and Czarnecki and colleagues survey the same consistency conditions across model engineering, databases and programming languages [124]. RCE's round trip condition is the GetPut condition at the level of the whole program: lift a projection back and the shape hash is unchanged.

Correct by construction. Dijkstra proposed in 1968 that a program and its proof be developed together, so that correctness is designed in rather than tested for [125]. Hall and Chapman carried the approach into a commercial secure system with formal specification and refinement at every step [126]. Minsky's rule for typed functional programming, make illegal states unrepresentable, says the same at the type level: a type that cannot hold a bad value removes the need to check for it [127]. Turner's total functional programming requires every function to terminate on every input, so that a program denotes a mathematical function with

no hidden failure [128]. RCE's Form is total in Turner's sense, its contract types make undeclared states unrepresentable in Minsky's sense, and the refinement chain of §2.1 ends in it in Dijkstra's sense. The consequence is the one stated in §1: what a program does is what its Form declares, so there is nothing to diff or track after the fact.

The universality thesis. Stated in theory terms, the thesis is that there is one universal Form. Programs are authored in it natively, every existing language is reachable from it as a faithful projection, and every existing program enters it by lift. Gurevich's thesis gives the ground: every sequential algorithm is captured, step for step, by an Abstract State Machine at its own level of abstraction [2], and parallel algorithms likewise [3]. A Form is such a machine over a database state.

Proposition U1 (capture). If a program in language L is a sequential or parallel algorithm in Gurevich's sense, then there is a Form whose transition simulates it step for step at the program's level of abstraction [2, 3].

Proposition U2 (faithful projection). If the projection π_L of a Form to L preserves the Form's trace on every input in the contract, then the Form and its projection are interchangeable for every observer of the trace, and L adds nothing the Form lacks.

Proposition U3 (lift by construction). Every programming language in use has a formal grammar, a parser that yields a concrete syntax tree, an abstract syntax tree and a semantics fixed by its specification or reference implementation. A lift λ_L is built per language in three stages: the grammar, then the abstract syntax tree, then the Form, with each construct of L given its Form rule by the semantics of L . A lift so built satisfies $\pi_L(\lambda_L(p)) \equiv p$ on behavior and $\lambda_L(\pi_L(F)) = F$ on shape hash, so the pair is a lens in the sense of [122], and a program in L is held as a Form with no loss.

Proposition U4 (replacement). If U2 and U3 hold for language L , then no program needs to be written in L : it is written once as a Form and emitted as L where a platform requires it.

Every language is a target and a source. The Form places no bound on targets. A target language is a row of the emit registry (the proposal spec/proposals/emit-targets.md in the specification repository): one emitter over the Form IR and one table per language fixing how that language keeps the numeric and text contract of specification 6.3, each row passing the same gate, the reference trace of every example program byte for byte (T9). The lift λ_L is the reverse map, from the language's pinned grammar and a semantics table into the Form, so every language is both an output of the Form and an input to it. The reference checker emits Rust, Python, JavaScript and Scala today, the engine's compiler emits the five targets of §5.1, and the registry's tiers list the systems, platform, dynamic and non-von-Neumann languages that follow, including the new entrants of 2024 to 2026 catalogued in §3.7.

U1 is a theorem. U2 is checked per target by differential testing (§5.1) and at large N in §2.3. U3 is a construction, built language by language from the grammar up, and §8.7 states how the construction is measured as it is built.

Two qualifications. The lift of U3 is a construction, and its domain is smaller than the sentence above suggests. The Form requires that every program finishes (T2), so the lift from any Turing-complete language is at best a map from the terminating fragment of that language, and membership in that fragment is undecidable (Rice’s theorem applied to a total target [141]). On the terminating fragment the lift is still not canonical: closure representation, which sub-terms become control states, and evaluation order are free choices, and the inverse exists only for programs in defunctionalized form [142, 143]. So λ_L is a construction on the image of a partial map whose domain is undecidable, the lens round-trip conditions hold on that image, and “every existing program enters it by lift” is a statement about the fragment the construction reaches, measured as §8.7 measures it. The second qualification concerns the hash over which the injectivity claim is made. Specification 6.1 hashes the positional forms of the state and of the event (regions and fields as lists in declaration order, a variant by its index, an event by its index among the module’s events); the stored entry keeps its readable event and state, and a trace repeats the hashed forms under hashed (trace format rce-trace/2). The Lean module RCE/NameFree.lean hashes De Bruijn keys, and RCE/Positional.lean models the positional embedding (records of declared types as field lists in declaration order, variants as index and fields, events by index, undeclared records by name) and proves it unchanged under any renaming that is injective and fixes undeclared names, which is what the nameless projection performs, so the state hash and the entire entry chain are unchanged for every hash function; the event index it hashes is by definition the declaration index of the De Bruijn model, so the two encodings agree on the component they share. The reference checker and the Python, JavaScript and Rust runtimes agree on the hashes, and a test checks that a program and its nameless projection have the same log hashes, as T7 states; the audit passes at 5,052 theorems with standard axioms only. An earlier reference checker hashed the named canonical JSON, so its log hashes depended on names while the model’s did not; the specification’s register of proposals records that finding, the test that demonstrated it, and its resolution as an erratum under C.15 (the log entry is outside a program’s meaning), with the trace fixtures, application goldens, 130 conformance pins and the standard library’s feed vectors regenerated.

3.7 The language: one text, costs and theorem status

The RCE language is defined by a written specification (</rce/spec/>; source in the [rce repository](#)). There is one language, the text of the specification at one commit, and every module of a program is in it (specification 2.7). Sections 2 to 10 define the core; Appendix C defines concurrency (par, for ... par, with budget, workgroups, LVars, sessions and supervisors) and applies to every module; Appendix D defines the engine profile, the handler specs and fits that this paper’s engine runs. No declaration selects a dialect: a first-line # rce-version: comment is a release label that names the specification commit a module was written against, is ignored by the checker and is not hashed, and a construct the checker does not implement is refused as reserved, not implemented (W-RES-1, Appendix F.12). The language is in development and changes as the project learns: a form can be added, generalised or replaced, and the change and every program of the repository it touches move together in one

commit, never as a per-module choice, so a program never depends on the version of another program or library: everything checked at one commit agrees with the specification at that commit.

Why one text and no version selection. The record on versioning supports this rule rather than the per-module selection that earlier drafts had. Dependency resolution with one version per package and arbitrary constraints is NP-complete, established for Linux distributions [176], restated in 2020 for every non-trivial component model [177] and proved independently for language package managers by reduction from 3-SAT [178]. A semantic version number does not carry the promise it makes: about one third of Maven Central releases over seven years introduced a breaking change, in minor and patch releases about as often as in major ones [179], a replication on 119,879 upgrades still found 20.1% of non-major releases breaking [180], and the Google engineering text concludes that semantic versioning is a lossy summary of the risk of adopting a change and keeps one version of everything in one repository [181]. Exposure is a property of the transitive graph: for more than 80% of the 35,863 Maven artifacts affected by the Log4j vulnerability the dependency was more than one level deep [182]. Where the record converges is on identity derived from content, so that two different things never share a name and variants coexist [183, 184], on change by accretion under names that are never reused [185], and on a direct proof that a change preserved behaviour rather than a number asserting it [186]; hardware design has run such a proof, equivalence checking, as the gate on every revision for two decades [187], and reproducible builds tie an artifact to its source by its hash [188]. The RCE design takes each of these as a rule: one text at one commit, definition hashes with a closed compatibility table (F.4.3), a manifest that pins by hash with no resolver (F.5.3), name-free log hashes (T7) and behaviour as the only sense of two programs being the same (§3.8). The survey of 24 October 2026 that reads the record against these rules, with the gaps it lists (a whole-program hash, the equivalence check run as a gate on every change, provenance in emitted code), is in the data repository (</data/>) and its proposal is `spec/proposals/one-commit-change.md`.

Budgets count work. A `with budget` block charges each operation a fixed cost from the table of Appendix C.6.4 and traps with budget when the count runs out. The count is a proxy for work, not a measurement of time or energy; `est_j` is separate.

Operation	Cost
call of a user function or a closure, c_{call}	4
standard library method on receiver size n , first-argument size k	$4 + w(m, n, k)$
channel send or receive	8
spawn	32
one <code>par</code> branch or <code>for ... par</code> chunk	16
list or map literal, or comprehension, with n results; <code>[x; n]</code>	$4 + n$

Operation	Cost
index, slice, member access; unary operators; and, or; integer, bool and text operators and comparisons	1
float +, -, *	2
float /, %	8

Sizes are the UTF-8 byte length of a text, the length of a list and the number of entries of a map; any other value has size 0. The method work w is fixed by the method's complexity class in the reference evaluator: $n + k$ for the linear methods on text, lists and maps (maps are ordered association lists), $n \lceil \log_2 n \rceil$ for sort (0 when $n \leq 1$), and 0 for the remaining methods. The calls that map, filter and fold make to their function argument are charged as calls in addition. The arithmetic saturates at the largest 128-bit integer, and w never decreases when n or k grows. The reference evaluator and the emitted Rust, Python and JavaScript runtimes compute the same function, and a test checks that the budget trap falls on the same operation in all four.

One cost rule. The cost of a standard library method call is the one cost of the text, in every module and wherever the call is executed from (specification 2.7, C.15). The rule is lexical to the text, not to a module, so linking modules changes no cost; tests check it at module boundaries in both directions on the reference evaluator and on the emitted Rust, Python and JavaScript programs.

Theorem status. The specification states theorems T1 to T34 and records the status of each exactly. The proofs are in Lean 4 (v4.12.0, no Mathlib) over a core calculus. The words used for the level of a result are these. *Proved* means a Lean proof about the core semantics of the language. *Proved on a semantics* means a Lean proof over a small-step semantics of a core of RCE. *Model* means a Lean proof about an abstract model that is not connected to the RCE semantics or to the reference evaluator. *Vectors* means the reference evaluator's results on generated programs are checked against the model by the Lean kernel, which is a correspondence on samples, not a proof about the evaluator. *Tested* means differential testing on stated inputs. *Checked mechanically* means decided by a program, such as an LALR(1) table builder.

	Statement	Level
T1	determinism: one result per state and event; one result per tick of the concurrency semantics	proved
T2	totality of δ , with an explicit bound on evaluation steps; structural recursion and bound composition along the call graph	proved (the Lean expression core has no user function calls; recursion and composition are proved on separate models)
T3	the runtime state is $\text{foldl}(\delta_1, s_0, D)$ and replay reproduces every hash and output	proved

	Statement	Level
T4	progress, including ticks (some task moves, or the run ends or reports Deadlock)	proved
T5	preservation, including the integer range of every written field	proved
T6	memory safety, full borrow paths, no mutable sharing between tasks	proved for the ownership core and an instrumented semantics of borrow paths
T7	name independence of log hashes, ids and select choices	proved
T8	atomicity: a refused event leaves the state unchanged	proved
T9-F	the Form IR targets (Rust, WASM, Python, XState, SCXML) print the reference trace	tested
T9-L	the program emitters (rust, python, js) agree with the reference evaluator	proved for the Rust emitter on a core, as a verified translation into MiniRust, a deep embedding of the emitted Rust subset; that rustc implements MiniRust is trusted and tested; the other emitters tested
T10	the grammar is unambiguous	checked mechanically (LALR(1), 0 conflicts, for both versions) and sampled by a fuzzer
T11	select fairness	proved on the rotation model
T12	a linear handle is consumed exactly once on every path	proved on a statement model
T13	f32 operations give the same bits on targets that agree on IEEE 754	proved for multiplication; for addition, subtraction and division the single-rounding property is Figueroa's theorem, cited and not mechanized
T14	checked u64 arithmetic is exact in range	proved
T15	iterators end, respect the loop limit and are prefix-consistent	proved
T16 to T31	Appendix C concurrency: par determinism, commutation, reductions, deadlock freedom, deadlines, workgroups, vector clocks, priorities, scopes, actors, sessions, LVars, atomic batches, replicas	proved on a semantics: T16, T17, T19a, T20, T23; proved on a model of runs: T24; proved on a session calculus: T28; model: T18, T19b, T21, T22, T25, T26, T31; model with vectors: T27, T29, T30. None is a proof about the Rust reference evaluator
T32	foreign replay: for every oracle, replay of the log equals the live run, and an answer contract holds of every logged answer	proved
T33	&mut restore on failure (F.4.4): a failed call leaves the caller's &mut state as at the call;	proved

	Statement	Level
	a returning call writes back (the earlier copy-out-on-every-outcome rule is retired)	
T34	the order-preserving ANF pass before every emitter gives the same value, store and trace of charges and emits	proved for a core fragment with full ANF; the implementation is tested differentially on every target, including a sweep of budgets across the trap point

T9 has two target sets because the paper's engine and the language have two compilers. T9-F is the claim about `rce form --emit`, which §5.1 measures. T9-L is the claim about the language's program emitters. A target outside these sets has no T9 claim.

 Grid of 35 entries, T1 to T34 with T9 split into T9-F and T9-L and T19 into T19a and T19b, each with a coded level. Proved: T1, T2, T3, T4, T5, T7, T8, T14, T15, T32, T33; Proved for a stated core or fragment; rest tested or cited: T6, T9-L, T13, T34; Proved on a small-step semantics of a core of RCE: T16, T17, T19a, T20, T23; Proved on a named model (rotation, statement, runs, session calculus): T11, T12, T24, T28; Model: T18, T19b, T21, T22, T25, T26, T31; Model, with evaluator vectors checked by the Lean kernel: T27, T29, T30; Tested: T9-F; Checked mechanically (LALR(1) tables): T10.

Figure 5. Theorem status, T1 to T34. Level of each theorem as recorded in the specification at the commit named below; theorems added after T34 are not shown. P: Lean proof about the core semantics. P*: proved for a stated core or fragment, the rest tested or cited. S: proved on a small-step semantics of a core of RCE. M+: proved on a named model (rotation model, statement model, model of runs, session calculus). M: Lean proof about an abstract model. MV: model, with reference evaluator vectors checked by the Lean kernel. T: differential testing. C: checked mechanically. T16 to T31 are the Appendix C concurrency theorems, and none is a proof about the Rust reference evaluator. Lean 4 v4.12.0, no Mathlib. Source: `spec/book/10-theorems.md` and Appendix C.13. A [PNG copy](#) is provided.

[Open full size](#) · [PNG](#)

Checks run. At engine commit 1eb5f02, `lake build` completes over the 31 files of `spec/lean/RCE/` with no sorry (46 files at public 4304048, 58 at e416fd0), and the standing audit reports the axioms of 2,192 theorem constants (the audit counts every kernel theorem constant; see the corrections note above) as only `propext`, `Quot.sound` and `Classical.choice`. Kani verifies 10 of 10 kernel harnesses (§5.3). The engine tests (`cargo test --release`) pass 589 distinct tests with 0 failures (2 ignored); `make check` reports 596 because its proof step runs 7 of those tests a second time. The reference checker, built from the public repository as its continuous integration builds it, passes 56 tests in the default build and 112 with the Appendix C suites enabled, with 0 failures, and 57 with the tree-sitter agreement test. The specification check builds the LALR(1) tables of the core grammar (sections 2 to 10) and of the grammar with Appendix C with 0 conflicts and parses every example with three parsers, and the WebAssembly build of the checker passes 401 of 401 checks against the example corpora.

 Summary of verification results. Engine tests at commit 1eb5f02: 589 pass, 0 fail, 2 ignored; make check prints 596 because it runs 7 proof tests twice. Lean 4: 2,192 audit theorem constants over 31 files with no sorry and only the axioms propext, Quot.sound and Classical.choice. WebAssembly checker: 401 of 401 checks. Bar chart of Kani 0.67.0: 10 of 10 kernel harnesses verified with 0 of 7,425 checks failed; checks per harness range from 216 for position_is_name_independent to 962 for handler_core_validate. Specification check: LALR(1) tables with 0 conflicts for the core grammar (932 states) and the grammar with Appendix C (1,110 states); 51 of 51 valid examples accepted, 12 of 12 syntax rejects and 64 of 64 ill-typed or inadmissible programs refused.

Figure 6. Verification at a glance, engine commit 1eb5f02. Data figure. Engine tests (cargo test --release): 589 pass, 0 fail, 2 ignored; make check reports 596 because it runs 7 proof tests twice. Lean 4: lake build over the 31 files of spec/lean/RCE/ with no sorry; the audit reports 2,192 theorem constants (audit count, not source declarations) whose axioms are only propext, Quot.sound and Classical.choice. Kani 0.67.0 verifies 10 of 10 kernel harnesses with 0 of 7,425 checks failed (bars: checks per harness). The WebAssembly build of the checker passes 401 of 401 checks against the example corpora. The specification check builds LALR(1) tables with 0 conflicts for the core grammar (932 states) and for the grammar with Appendix C (1,110 states), accepts 51 of 51 valid examples and refuses 12 of 12 syntax rejects and 64 of 64 ill-typed or inadmissible programs. The Lean, WebAssembly and specification checks were run on the public repository at ac95172, whose specification sources are identical to the engine's. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

A convergence observation. Two sweeps of 24 October 2026 catalogued 111 programming languages first made public between January 2024 and October 2026 [189, 190]: 72 designed for language models or agents as authors or implemented largely by an agent (48 from the agentlanguages.dev catalogue [166], read at its source repository, and 24 beyond it), and 39 general-purpose or systems languages, with 35 pre-2024 languages reaching a milestone in the window. Among the machine-author entrants the count by year is 3, 10 and 57 for 2024, 2025 and 2026. Two positions are near-universal: refusal of invalid input at a boundary (54 of 72) and emission to another language (57 of 72; 17 of 39 general entrants, with Go the target of 2026 and C the systems default). Minorities take a canonical form (25 of 72), examples or tests inside the program (21 of 72; 2 of 39), a written specification as the authoritative definition (16 of 72; 14 of 39), deterministic evaluation as a stated gate (13 of 72; 3 of 39, among them Gossamer, Dawn and Wyzer) and memory safety without a collector (8 of 72; 7 of 39, converging on precise reference counting or ownership rather than lifetime annotations, with Goose's data stacks the outlier [191]). An append-only log with replay (5 of 72; none of 39) and hash-identified definitions (2 of 72; 3 of 39, Bend, Scrapscript and Dawn [192]) are rare, and an intermediate form as the source of truth is absent from the new general-purpose languages. A distinct trend of 2025 and 2026 is languages whose compilers were written largely by model agents and disclosed as such, among them Goose, Rue, Cursed and Jolt, whose performance claims are self-reported. The design choices of this paper therefore recur

where they are cheap (a declared outcome for every input at the boundary, printing to a host language) and remain rare where they are theorems here (one canonical form, examples as part of the program, replay from a hash-chained log, name-free identity). This is an observation that choices recur and where; it is not evidence that they are correct, and this paper makes no comparison of quality with those projects. The emit registry of the specification repository takes every entrant as a target and a source on the same terms as every other language.

3.8 Behaviour, equivalence and dimension

Definition 3 says two fits are equivalent when their output sequences agree on every event sequence. This section gives that definition a universal property, defines the dimension of a program as a computed number, and reports the first measurement of it. The normative text is section 6.10 of the specification (draft 0.9, commit 45fc95c), with theorems T57 to T61 and the command `rce dim`.

Behaviour as a point in the final coalgebra. Fix a program P with state space S , event alphabet E and $\delta : S \times E \rightarrow S \times O^*$, and write $\delta_1 = \pi_1 \circ \delta$ and $\lambda = \pi_2 \circ \delta$ as in Definition 1. The behaviour of a state s is the function $\text{beh}_P(s) : E^* \rightarrow (O^*)^*$ with $\text{beh}_P(s)(\varepsilon) = \varepsilon$ and $\text{beh}_P(s)(e \cdot w) = \lambda(s, e) \cdot \text{beh}_P(\delta_1(s, e))(w)$: the per-step outputs along any event sequence. $(S, \langle \lambda, \delta_1 \rangle)$ is a coalgebra of the functor $F(X) = (O^* \times X)^E$; F has a final coalgebra Beh whose carrier is the set of causal stream functions, and beh_P is the unique coalgebra morphism $S \rightarrow \text{Beh}$ (Rutten [144], Theorem 2.3). Two fits are equivalent, $F \equiv G$, when $\text{beh}_F(s_0^F) = \text{beh}_G(s_0^G)$. This is Definition 3 restated; the new content is that $\equiv$ is equality in one object that every Form, in every fit language, maps into by the same construction. Behaviour space is not a manifold: it carries the prefix ultrametric $d(b_1, b_2) = 2^{-n}$, n the length of the shortest distinguishing event sequence, under which it is complete and totally disconnected [145]. Trace agreement on every sequence of length at most k therefore certifies $d \leq 2^{-(k+1)}$: a ball around the behaviour, not the behaviour. That is what the differential tests of §5.1 establish, and it never enters a hash.

Two proof methods. A relation $R \subseteq S \times S$ is a bisimulation for P when $s R t$ implies, for every e , $\lambda(s, e) = \lambda(t, e)$ and $\delta_1(s, e) R \delta_1(t, e)$. T58 states that $\text{beh}_P(s) = \text{beh}_P(t)$ if and only if some bisimulation relates s and t : to show two fits equivalent, exhibit the relation; to show them different, exhibit a distinguishing sequence, which also gives the distance. T1 and T2 (totality and determinism) are exactly the completeness and determinism that make the minimal machine unique; an incompletely specified machine has no unique minimum [146].

Dimension. $\text{reach}(P)$ is the set of states reachable from s_0 ; it is finite because S is finite under the value and size limits of the specification (N9). The dimension of P is

$$\dim(P) = |\{\text{beh}_P(s) : s \in \text{reach}(P)\}|,$$

the number of distinct behaviours P can be in, which is the Nerode index of its minimal Mealy machine [147], computable for an explicit alphabet in $n \log n$ [148]. $\log_2 \dim(P)$ is the bits of state the behaviour needs; $\log_2 |\text{reach}(P)| - \log_2 \dim(P)$ is the *redundancy* of the fit, the bits of state that change no future output. A control dimension $\dim_c(P) \leq \dim(P)$ is defined on the projection that replaces every guard over Σ_d by its predicate symbol, a symbolic finite automaton [149], and its minimal machine, numbered by a breadth-first traversal in the canonical event order, has a unique name-free string [150]; its SHA-256 over canonical JSON is `shape_ctl`, a third key beside the name-free term hash and the e-class key of Definition 6. Equality at any key implies equivalence; inequality implies nothing.

What is and is not a theorem here should be said plainly. S and E are finite by the value limits and astronomically large: $|S| \leq 256^{\text{VALUE_LIMIT}}$ per field, so finiteness makes every behavioural question decidable and none of them tractable. The minimal machine is a canonical form for the finite-control projection, where symbolic-automata minimisation is unique up to predicate equivalence [149]; for fields that are only compared and copied, register automata give a Myhill–Nerode theorem [151]; the moment δ does integer arithmetic into state, the only canonical object is the term itself. “Every Form is a finite Mealy machine, and its minimal machine is its canonical form” is therefore true as stated for Σ_c plus guard predicates and is not a computable claim for Σ_d . That is also the honest reading of Proposition 3: the least-joule extraction returns the least-joule term *the extractor finds*, and three distinct functions now act on one e-class: the target-blind budget units of specification C.6, which place the budget trap on the same operation in every runtime and are never joules; `est_j`, which is per target (§6.3); and the hashed representative, chosen by tree cost with a canonical-JSON tie-break for reproducibility. The emitted term and the hashed term of one e-class may differ; neither may differ across engines given the same log.

Status of the theorems. T57 (finality: beh_P is the unique coalgebra morphism), T58 (equivalence is bisimilarity), T59 (trace agreement to length k bounds the distance by $2^{-(k+1)}$), T60 (for finite S the quotient of $\text{reach}(P)$ by the largest bisimulation is the minimal machine, with $\dim(P)$ states) and T61 (`shape_ctl` is invariant under renaming and under predicate-equivalent rewrites of guards) are stated in chapter 10 of the specification, and `spec/lean/RCE/Behaviour.lean` is in `lake build` (Lean 4.12.0; `scripts/lean-audit.sh` reads 4,947 theorems with standard axioms only and no sorry). What is proved, in the chapter’s exact scope: the morphism equation of T57 (`beh_cons`; uniqueness of the morphism is stated, unproved); T58 for one program and for two (`bisim_beh`, `beh_bisim`, `behEq_iff_bisim`, `bisim2_beh`, `beh_bisim2`); and T59 in both directions over an explicit finite alphabet, the declared events over the boundary sets that `rce dim` explores (`trace_ball`, `ball_trace`), with the statement over all of E stated, unproved. T60 is stated, with a model proof by partition refinement as the next step. T61 is tested for renaming only, against the name-free projection `rce fmt --nameless`, which prints a program with positional names by kind in first-declaration order and is accepted whenever the source is, with the same trace hashes by T7; its invariance under guard rewrites is stated, unproved, and the symbolic quotient is not implemented.

rce equiv: the equivalence check as a tool. `rce equiv A B` explores two programs from their initial states over the same event instances, compares every step by its name-free output (emitted values in positional form, a refusal by its error's declaration index, T7), refines the two graphs together and reads whether the two initial states fall in one class (T58); when they do not it prints the shortest distinguishing event sequence and the two outputs at its end. `rce change OLD NEW` combines it with the export diff of F.4.3 and classifies a change as an erratum (every export compatible, every behaviour kept), a meaning change or an interface change, which is the rule of C.15 decided by the tool. Run between every corpus program and its name-free projection as a continuous-integration step, the check found one program whose projection was refused (a state field named like a standard library method had renamed the method); the projection was corrected and the sweep reports every corpus Form equivalent to its projection, as T7 states. It is the Form's version of the equivalence check that hardware design runs on every revision (§3.7): a change under which every program keeps its behaviour is an erratum, any other a meaning change, decided by the tool rather than by a version number. With `rce hash`, which prints a program hash over the name-free projection (one hash for a program and its renaming, a different one for a changed step), and the provenance line that every emitted file now carries, the three mechanisms the record calls rare in new languages (section 3.7) are commands of the reference checker.

rce dim, and the first measurement. `rce dim FILE [--fuel n] [--control]` explores $\text{reach}(P)$ breadth-first over the declared events with payloads drawn from each field's refinement type or, without one, from the boundary set of specification 6.3, up to `--fuel` states, then runs Hopcroft partition refinement with the outputs as the initial partition (100,000 states in 10 to 40 s; the first Moore iteration was quadratic and did not finish), and reports states explored, behaviours (exact when the bound is not reached, else a lower bound), redundancy in bits per region, and with `--control` the control states and `shape_ctl`. Three examples were written with known values: a counter with an unread flag (1.00 bit of redundancy), a mod-6 counter that emits parity (1.58 bits), and a door with three reachable states and no redundancy. The preregistration (`openie-research-experiments/rce/bench/dimension/PREREG.json`, registered 22 October 2026 before the command had been built, amended on 23 October with an observability stratum before the frozen run, both with hashed sidecars) was then run once against a fixed build (run identifier `dim-r1`, commit 22a5992, binary hash recorded before the run), `--fuel 100000 --control`, over the 60 programs of `spec/examples/` and the 17 programs of `rce-examples` that the checker accepts standalone. Of 77 programs, 49 are commands or have no explorable event, 28 are stateful, 13 are bounded within the fuel, and 15 are capped. A preliminary run the day before, at a lower exploration limit, had the same shape and is kept with the results.

Corpus	Program	States	Behaviours	Redundancy (bits)	Control states
rce-examples	ring_buffer	12,500	781	4.00	5

Corpus	Program	States	Behaviours	Redundancy (bits)	Control states
rce-examples	cellular_automaton	41	1	5.36	1
rce-examples	reconcile	4	1	2.00	1
rce-examples	thermostat, word_count	2	1	1.00	1
rce-examples	server	1	1	0.00	1
spec	30_state_machine_guarded	3	3	0.00	3
spec	17_traffic_light, 52_context_text	3	1	1.58	1
spec	51_refined_input	4	1	2.00	1
spec	57_float_vectors	8	1	3.00	1
spec	15_actor, 56_utf8	1	1	0.00	1
spec	16_multi_region (capped)	100,000	≥ 102	≤ 9.94	102
spec	58_float_keys (capped)	100,000	$\geq 2,466$	≤ 5.34	2,466
spec	31_inventory, 55_rcedb (capped)	100,000	$\geq 99,554$; $\geq 99,990$	≤ 0.01 ; 0.00	—
both	account, arm, handshake, ledger, queue_sim, 01_counter, 04_bank_account, 53_capability (capped)	100,000	$\geq 99,999$	0.00	—

The result of the run is a definitional one. Eleven of the thirteen bounded state machines emit nothing and refuse nothing on any explored path, so by 6.10 each has exactly one behaviour whatever its state: no future output depends on its state, which stays in the log and readable. Their intent is carried by example lines that read internal state, which by 6.9 still hold; the two readings are consistent, and the gap between them is what `rce dim` measures. On both corpora, most bounded state machines are silent: no explored step emits or refuses. In the output-bearing stratum ($N = 2$) the preregistered hypotheses are recorded, not claimed: H0 (at least half of bounded programs carry more than 0.5 bit) is 1 of 2, H1 is not testable, H2 (one region carries at least 80% of the redundancy when it exceeds 1 bit) holds on the one row that qualifies, and the falsifier RCE-F6 is neither triggered nor confirmed. The capped programs are data-distinguished (counters, ledgers, accounts): every explored state is its own behaviour, as expected for unbounded numeric state, with `16_multi_region` and `58_float_keys` the exceptions. `ring_buffer` is the first real measurement of the quantity for any program in any language: 12,500 reachable states, 781 behaviours, five control states; four bits of its state change no output, all in one region, and as a machine of which steps fire the buffer is a five-state thing. Two things follow for the project. The next run samples the output-bearing stratum, from the standard-library Forms and the brownfield lifts, which emit. And in a state machine whose examples read state it never emits, the intent is carried by the examples and by no

output; a report from `rce` check that names such an example (one that reads a field no step ever emits or refuses on) is an open proposal. The preliminary run had also found and fixed a definitional error: the first control label recorded which regions changed, which is not a function of the output, and gave $\dim_c = 4 > \dim = 1$ on one program; the label is now the refusal or the emitted constructors only, and $\dim_c \leq \dim$ holds on every row.

The second frozen run. The run `dim-r2` of 23 October 2026 (engine commit 1107442b, registered as a second amendment of the same preregistration before the run, with the stratum read from the key that `rce dim` reported as observable at that commit, now `has_output`) samples the output-bearing stratum by rule: every program of the merged repository under its examples, programs, applications and book listings that is accepted standalone, has a state region and contains an emit or a fail statement. That is 25 programs, 24 of them output-bearing, 14 bounded within 100,000 states. Excluding the three programs written for `rce dim` and their book copies leaves 8 bounded output-bearing programs, of which 6 carry more than 0.5 bit (Wilson 95% interval 0.41 to 0.93), with a median redundancy of 1.79 bits (interquartile range 0.25 to 3.37); over all 14, 10 of 14 (0.45 to 0.88) and a median of 1.29 bits. RCE-F6 (a median below 0.1 bit) is not triggered; the interval's lower end is below one half, so H_0 is supported and not established at this size. H_1 is not supported (the two single-emit-site programs carry 1.58 and 4.00 bits, the two multi-site programs 2.51 and 3.65), and H_2 holds on 8 of 9 qualifying rows only because 8 have one state region; the one two-region program attributes 62% of its redundancy to its larger region. The run's substantive addition is three application-sized measurements. The chat application explores 100,000 states with 176 behaviours and 69 control states, so at least 9.15 bits of its state change no output, the largest redundancy measured and a lower bound; the selective forwarding unit (40,218 states, 7,038 behaviours, 2.51 bits) and the terminal (3,390 states, 270 behaviours, 3.65 bits) are exact. The ratio of control states to behaviours varies by an order of magnitude across the population (5 to 781 for the ring buffer, 69 to 176 for the chat, 5,095 to 7,038 for the forwarding unit), which measures how much of a program's behaviour is which steps fire against which values flow. The run also shows that the stratum needs a finer registered rule: the tool's flag is true for a program whose every output is the same (a constant, or a refusal every state gives), which the first run had classed as silent by hand, so a third run should register "output-bearing, with more than one distinct output" before sampling. Rows, intervals and the amendment are in the experiments repository under `rce/bench/dimension/` (`results/dim-r2.jsonl`, `RESULTS.md`). A third run, `dim-r3` (23 October 2026, registered before it), applied the finer rule the second run asked for: `rce dim` reports the number of distinct outputs (`distinct_outputs`; emitted values and refusals counted separately as emits and refuses), and the stratum is output-bearing with at least two of them. On 28 programs (the second run's population plus the three accepted programs with two or more state regions), 22 are strictly output-bearing and 12 of those are bounded; the median redundancy is 1.29 bits (8 of 12 above 0.5 bit, Wilson 0.39 to 0.86), and 2.05 bits over the 6 not written for the tool. The three programs the finer rule removed (a thermostat, a reconciler and a cart) are each observable only by a constant, so the second run's medians had counted bits that change an output which never changes at all. H_2 is tested where it can be, on the one strictly output-bearing multi-region program above 1 bit, whose larger region carries 62%, below the 80% threshold; it stays open on one row. A fourth run

over the 86 conformance programs with state (registered before it) measured the five harvested host-contract programs for the first time, all bounded within the fuel and carrying from 0.01 bit (the terminal driver) to 2.16 bits (the channel driver) with control quotients in the thousands, and answered the multi-region question for this corpus: the only strictly output-bearing multi-region programs are data-distinguished copies of one capability example, so H2 cannot be decided from existing public material and the next step is authoring observable multi-region programs, registered as a population before the run. That step was taken on 24 October 2026: twelve two-region programs (an elevator, a vending machine, a crossing, a car park, a printer, a lobby, a bank card, a washer, a session, a player, a climate controller and a reservoir) were written into the programs corpus and registered as the fifth run, with the disclosure that the author ran the tool on the drafts. All twelve are strictly output-bearing, bounded and explored in milliseconds; eleven have zero redundancy and the twelfth, the climate controller, keeps 1.58 bits in two flags that record its last action and that nothing reads. The median is 0.00 bits, so RCE-F6 is triggered on this population: accidental state is not common in small Forms whose caps are enforced by refusals and whose phases emit. Read with the earlier runs (1.29 bits over the strictly output-bearing hand-written examples, 0.01 to 2.16 bits for the host programs, at least 9 bits for the chat application), the quantity grows with program size and with the number of fields nothing reads. The claim the data supports is the falsifier's own: `rce dim` is a diagnostic for larger programs and for regions added without an observer, not a property of small Forms in general.

The concept record. Specification 6.6 now describes an unhashed side table keyed by the name-free keys of a program and holding its names, intent strings and doc comments, and `rce emit form` prints it under names, keyed by declaration index (the index `shape_ctl` uses), after the hashed fields. Projections render it; δ never reads it; it is the complement that makes the projection of a Form a symmetric lens [123] rather than a retraction. The evidence of §2.1 says the table must hold the concept a name stands for and not only its spelling, because word choice and ordering are the two steps developers disagree on at 6.9% [137] and the step they agree on is the concept. Whether a model can reason from the Form with the record attached as well as from a named surface is the first of the three registered experiments of §12.3 and is open.

4. The engine: gears, gate and receipts

4.1 The engine is a Form

RCE runs software as a graph of small deterministic handlers. A handler that cannot decide raises an escalation. A resolver answers the escalation once, and the answer is installed as a permanent handler, so the next case of the same shape runs with no resolver. The engine meets every Form invariant, each checked by a named test in `tests/form_laws.rs`:

Invariant	What the engine does
Reload is a verified fold	Reload reads the log verbatim, recomputes every hash and link, and folds the rest with δ . Nothing is re-stamped. A broken chain refuses to load.
Name-free tick order	Handlers run in topological rank order, then by a name-free node key. Outputs are staged and delivered after all handlers run, as in ASM semantics. Renaming every node gives the same anonymous state hash at every entry.
No silent drop	A full mailbox holds the message in an ordered queue and logs Backpressure. With capacity 1 and 10 tasks, 9 Backpressure entries are logged and all 10 tasks close.
Payloads are in the log	Inputs, ticks, external handler results, installs and faults are log entries. The state hash after every entry matches a fold from an empty engine, on 1,000 random logs.
Logical clock	Liveness counts ticks, not wall time.
Stable hashes	SHA-256 (Secure Hash Algorithm, 256-bit) over canonical JSON, keys sorted at every depth, for entries, states, shapes and rule ids.

4.2 The gears

Each gear is tried only when the gear below it does not pass the gate. That order is one fixed cascade.

Gear	Method	Lineage
G0 Lookup	A shared handler library keyed by SHA-256 shape hash; for code repair, localization to ranked functions	Code search and retrieval [21]
G1 Rules	Deterministic, guarded repair templates and schema-derived validators, each checked for mutation sensitivity	PAR, TBar, AVATAR, FixMiner [22, 23, 24, 25]; codemods such as Coccinelle [26]
G2 Synthesis	Enumeration over the handler grammar, solvers and e-graphs	Sketching, SyGuS, egg [27, 9, 12]
G3 to G5 Models	A typed decision model, a local small model, then a remote model, last	Model cascades [28]

4.3 The gate

A candidate is promoted only when it passes the gate. The gate replays the node's recorded inputs, runs the candidate in shadow on fresh samples with a coverage bound (the rule of three at a declared error budget), checks the output schema, and requires the proof certificate of Section 5. A rejected candidate is logged with its counterexample.

Diagram. Four gear boxes stacked from cheapest to costliest: G0 Lookup, a shared handler library keyed by SHA-256 shape hash; G1 Rules, guarded repair templates and schema-derived validators; G2 Synthesis, enumeration over the handler grammar, solvers and e-graphs; G3 to G5 Models, a typed decision model, a local small model and a remote model, marked as proposers only. Each gear sends its candidate to one gate with five checks: replay of recorded inputs, a shadow run with a coverage bound, an output schema check, the Kani proof certificate, and for rewrites a randomized check then a Z3 proof. A pass promotes the candidate to a stored handler, which feeds G0; a fail is logged with its counterexample and escalates to the next gear.

Figure 7. Models propose, checkers decide. Diagram: gears run in order of cost: G0 Lookup, G1 Rules, G2 Synthesis, G3 to G5 Models (a typed decision model, a local small model, then a remote model). Every candidate passes through the same gate: replay of the node's recorded inputs, a shadow run on fresh samples with a coverage bound, an output schema check and the Kani proof certificate; a proposed rewrite also needs a randomized check and then a Z3 proof. A candidate that passes is promoted and closes the same shape at G0 next time; one that fails is logged with its counterexample and the next gear is tried. The Kani harnesses and the SMT encoding do not change through promotion. Source: Sections 2.2, 4.2, 4.3, 4.6, 5.2 to 5.5. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

4.4 Receipts

Every promotion, refusal, deferral and install is a mutation in an append-only log chained with SHA-256, so tampering with any past entry breaks the chain [29, 30]. Each handler's cost is priced from interpreter operation counts with the host cost table of FGIR, the native-code backend of flowG, and labeled `est_j`. Each foreman call carries its receipt. A receipt names the gear, the method of the estimate (`basis`), the candidates tried and whether the gear resolved the escalation. Its `est_j` field is required: it is a finite non-negative number, or null, and null is allowed only when `basis` says unpriced, as for a model call with no energy figure. A receipt with the field missing does not parse. An unpriced cost is never read as zero, so a total over receipts reports its priced and unpriced invocations separately, and a saving that depends on an unpriced resolution is reported as unknown. The ladder runs add a package energy counter on a subsample, labeled `reported_j`.

Diagram. Top: three log entries, seq 0, 1 and 2, each with fields `seq`, `event`, `prev_hash`, `state_hash` and `hash`. The first `prev_hash` is 64 zeros; each later `prev_hash` equals the previous entry's hash, drawn as arrows. `state_hash` is the SHA-256 of the state after the step; `hash` is the SHA-256 of the canonical JSON of `event`, `prev_hash`, `seq` and `state_hash`. Bottom: a gear receipt with fields `gear`, `gear_name`, `est_j`, `basis`, `candidates_tried` and `resolved`, beside the rule that `est_j` is required and is either a finite non-negative number or null, with null allowed only when `basis` says unpriced; unpriced cost is never read as zero joules.

Figure 8. The log is a hash chain; every cost carries a basis. Diagram: entries of the durable log D are {seq, event, prev_hash, state_hash, hash}. state_hash is the SHA-256 of the canonical JSON of the state after the step; hash is the SHA-256 of the canonical JSON of {event, prev_hash, seq, state_hash}; the first prev_hash is 64 zeros, so changing any past entry changes every later hash. A gear receipt carries gear, gear_name, est_j, basis, candidates_tried and resolved. est_j is required and is a finite non-negative number, or null only when basis says unpriced; model-call receipts carry est_uj in microjoules under the same rule. An unpriced cost is never read as 0 J. Hash labels are symbolic.

Source: specification 6.1 and Section 4.4. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

4.5 The Form compiler

rce form takes a specification and its fit and emits five targets from one Form IR:

```
.rce spec + fit
  lower → normalize → egg e-graph, equality saturation (+ admitted proposals)
  → extract (cost = est_j) → Form IR (typed relations, regions, rules with SHA-256 ids)
    | Rust crate (pure step, typed state, SHA-256 log and fold, tests)
    |   └ WASM (the same crate on wasm32-wasip1)
    | Python (standard library only)
    | XState v5 machine
    └ SCXML document, executed by SCION
```

The IR fixes one behavior at every numeric edge: checked 64-bit integer overflow is an error and never wraps; integer / truncates toward zero ($-7/2 = -3$); % takes the dividend's sign ($-3 \bmod 2 = -1$); division by zero is an error; floats are IEEE 754 binary64 and a non-finite result is an error; null and absent are distinct; time is the logical clock only. Every target is generated. None is authored.

A process, HTTP or closure handler has no term, so its logged results compile to a recorded oracle keyed by the SHA-256 of the canonical input. `rce scan DIR --form` lifts existing Rust, Python, TypeScript and Go into the Form's schema, and every lifted element is labeled as a syntactic lift.

4.6 Method: the whole stack evaluated, every interface assumption explicit

A program is a stack of levels: tokens and statements; handlers; the Form, a single step function $\delta : S \times E \rightarrow S \times O^*$ over $S = \Sigma_c \times \Sigma_d$ (control regions; data: the clock, the input count and typed relations); the log, a SHA-256 hash chain whose entries record each event, the hash of the state it produced and the hash of the previous entry; the rewrites that transform the Form; and the substrate that executes it. Each level rests on assumptions about the level below. RCE does not treat any level as fixed. It evaluates the whole stack, writes each interface assumption down as a checkable statement, and checks it on every run.

Boundary	Assumption	Check
source to handler	an input satisfies its source contract	the contract check at the input event of δ
handler to sink	every Ok output satisfies the output schema	handler proof (verify::handler_proof), total and deterministic by the Kani-checked kernel evaluator
log to state	every entry carries the hash of the state it produced	kernel fold, which refuses the first entry whose hash does not match (Kani harness fold_rejects_wrong_state)
rewrite to Form	a rewrite preserves the output for every input	randomized check, then an SMT proof with Z3
stub to foreman	a handler that cannot decide escalates instead of guessing	stub handlers emit escalations; resolutions are promoted only through the promotion path
Form to target	every emitted target computes the same δ	every target carries the same reference trace

A failed check locates the break at the boundary where it occurs. The correction is made at that level, passed through the check of that boundary, and promoted. Every level above the corrected one inherits the correction without further change, because each level above consumes the corrected level only through the interface whose assumption is now checked. This is the sense in which corrections carry upward and compound. The [RCE engine demo](#) runs this in the browser on the RCE core compiled to WebAssembly: its Interfaces layer breaks each assumption and shows the check that detects it, and its Compounding layer shows the correction carrying into every later round.

5. Verification

5.1 Differential testing across five targets

tests/form_targets.rs drives an eleven-operator arithmetic specification with 29 edge events plus 500 seeded random edge values. The generated Rust, Python, WASM, XState and SCXML targets print the reference trace byte for byte, including the arithmetic edges. This is differential testing in the line of McKeeman and of Csmith [86]: agreement among independent implementations on a stated input set. It is not translation validation [85] and not a verified compiler. It covers the eleven operators and the 529 events in the test, and it says nothing about inputs outside them. The Z3 and Kani results below are the verification.

Figure 9 shows the five projections of one Form and the round trip of the two statechart targets.

Diagram: one Form, made of state, events and a pure total transition, is projected by deterministic compilers to five targets: Rust, Python, WASM, XState and SCXML. Solid arrows run from the Form to each target. Dashed arrows run back from XState and SCXML to the Form, showing the round trip to byte-identical bytes and the same shape hash. Rust, Python and WASM replay the reference trace byte for byte.

Figure 9. One Form, many projections. Diagram: deterministic projection to five targets (§4.5); the XState and SCXML projections decode back to byte-identical bytes (§2.3); all five targets print the reference trace byte for byte on the 529 events of tests/form_targets.rs. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

Diagram in two lanes. Top lane, the engine compiler rce form: a .rce handler spec and its fit are lowered, saturated and extracted by est_j into the Form IR, which is projected to five targets: a Rust crate, WASM, Python, XState v5 and SCXML run by SCION. All five print the reference trace byte for byte on 529 events in tests/form_targets.rs; this is claim T9-F, established by differential testing. Bottom lane, the language command rce emit: a checked .rce module is emitted as form JSON, Rust, Python or JavaScript; claim T9-L, proved for the Rust emitter on a core as a verified translation into MiniRust, and tested for Python and JavaScript.

Figure 10. One Form, five targets, one reference trace; two compilers, two claims. Diagram. Top: rce form lowers a handler spec and its fit to the Form IR and projects it to a Rust crate, WASM, Python, XState v5 and SCXML; all five print the reference trace byte for byte on 529 events (29 edge events plus 500 seeded edge values of an eleven-operator specification, tests/form_targets.rs). This is T9-F, established by differential testing, not a verified compiler. Bottom: rce emit TARGET FILE translates a checked module (not the Form) to form JSON, rust, python or js; T9-L is proved for the Rust emitter on a core as a verified translation into MiniRust (that rustc implements MiniRust is trusted and tested), and tested for the others. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

5.2 Rewrites proved in Z3

Verified rewrite rules have a lineage. Ruler infers and checks rewrite rules by equality saturation with no model in the loop [87]. The 3 of 16 model-proposed rules admitted in Section 5.5 are therefore a statement about the gate, which refused 13, and not a claim that a model is the best rule source.

Every built-in e-graph rewrite is proved in the Z3 SMT solver (satisfiability modulo theories) [31] against a quantifier-free encoding of the Form's term evaluation and arithmetic: values as a datatype, 64-bit bit-vectors with checked overflow, signed division and remainder with the dividend's sign, and binary64 floats. A rewrite that changes overflow, escalation order or a message gets no proof. A proposed equivalence enters the e-graph only with a proof.

5.3 Kani proofs of the kernel

The engine's kernel (hash chain, fold, name-free positions and handler evaluation) is checked with the Kani model checker for Rust [32]. Handler evaluation is the fixed point of a non-recursive step, and Kani proves that step total and deterministic per constructor, by structural induction. Ten harnesses are verified with Kani 0.67.0, with 0 of 7,425 checks failed:

Harness	Property
chain_appended_verifies	An appended hash chain verifies
chain_detects_tampering	A changed entry breaks the chain
fold_equals_replay	The fold equals replay
fold_rejects_wrong_state	A fold that disagrees with the stored state is refused
position_is_name_independent	Rule positions do not depend on node names
reason_tokens_are_name_independent	Reasons are name-free by whole tokens
handler_core_leaves	Leaf handlers are total and deterministic
handler_core_validate	Validation is total and deterministic
handler_core_pipeline	Pipelines are total and deterministic
handler_core_ifhas	Guards are total and deterministic

5.4 Promotion gate on the proof certificate

The Kani run writes a certificate whose hash is stored in `proofs/kani.cert`. The engine, the log, rule ids and handler evaluation route through the proved kernel, and promotion requires the certificate.

5.5 The model as a proposer behind the check

A model may propose equivalences. It may not admit them. With a local 1.5-billion-parameter coding model (Qwen2.5-Coder-1.5B under `llama.cpp`) on the demonstration specification, the proposer offered 16 equivalences. The check admitted 3, by randomized trials and a Z3 proof, and rejected 13. A deterministic proposer of 87 familiar identities, 77 arithmetic and 10 structural, had 9 admitted, all structural, and 78 rejected: every arithmetic identity, for fault order, non-numbers or overflow, and one structural rewrite, `const-absorbs-pipe`, which would drop an escalating prefix (engine test `tests/form_proposers.rs`).

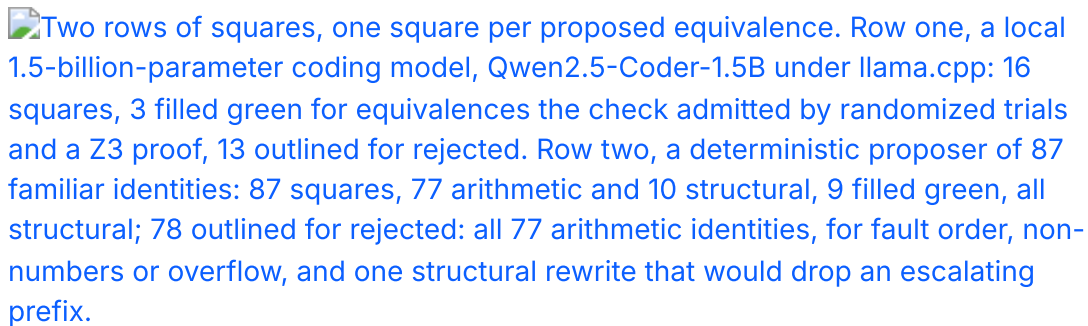
Two rows of squares, one square per proposed equivalence. Row one, a local 1.5-billion-parameter coding model, Qwen2.5-Coder-1.5B under llama.cpp: 16 squares, 3 filled green for equivalences the check admitted by randomized trials and a Z3 proof, 13 outlined for rejected. Row two, a deterministic proposer of 87 familiar identities: 87 squares, 77 arithmetic and 10 structural, 9 filled green, all structural; 78 outlined for rejected: all 77 arithmetic identities, for fault order, non-numbers or overflow, and one structural rewrite that would drop an escalating prefix.

Figure 11. The gate on proposed equivalences. Data figure: one square per proposed equivalence. A local 1.5-billion-parameter coding model (Qwen2.5-Coder-1.5B under llama.cpp) on the demonstration specification offered 16 equivalences; the check admitted 3, by randomized trials and a Z3 proof, and rejected 13. A deterministic proposer of 87 familiar identities (77 arithmetic, 10 structural) had 9 admitted, all structural, and 78 rejected: all 77 arithmetic identities, for fault order, non-numbers or overflow, and one structural rewrite that would drop an escalating prefix. Source: Section 5.5; deterministic counts from the engine test tests/form_proposers.rs. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

5.6 Brownfield agreement

The lift reads the same order machine from Rust, Python, TypeScript and Go fixtures, runs the original code on 40 seeded sequences of 30 calls, and agrees on all 1,200 calls per language. A frozen benchmark, preregistered before any result, re-expressed three existing sales-outreach programs as Forms. The Forms agree with the original code on 4,500 of 4,500 recorded inputs, and extraction lowers `est_j` on each (for example 1.69×10^{-8} J to 1.49×10^{-8} J).

6. Lift: entering the ecosystem from other languages

Lift is the door into the ecosystem for code written elsewhere. It is a construction, made per language from tools every language already has: a formal grammar, a parser such as [tree-sitter](#) that yields the syntax tree, the abstract syntax tree and the symbols that language-server tooling resolves ([Language Server Protocol](#)). `rce scan DIR --form` builds the Form from that tree for Rust, Python, TypeScript and Go (§4.5, §5.6). A lifted program carries the faults of its source, so lift includes repair, measured below on public bug corpora with the location given. Inside the ecosystem these faults cannot be written.

6.0 The repair ladder on lifted code

A coding task is measured here by the **intent bits** it needs: the bits that must be supplied from outside what the engine already holds (the program around the edit, the libraries, the tests, the index of known fixes). Writing H_{task} for the bits that pin the target artifact down and H_{corpus} for the bits the corpus already fixes, the supplied bits are $I = H_{task} - H_{corpus}$. A gear that ranks

the right edit at rank r among its candidates has supplied $\log_2 r$ bits of search, and the gate supplies the rest. The ladder orders tasks by edit size, an objective proxy for I : one token (L0), one statement (L1), one hunk (L2), several hunks in one file (L3), several files (L4), a function from a specification or tests (L5), an issue-level change (L6) and a program from a specification (L7). The rungs, corpora, splits and hypotheses were preregistered before the full runs; the files are in the [public repository](#).

Rung	Held-out test corpus	N	Solved	Rate	Wilson 95% CI	Median search bits	Model calls
L0 one token	PIT-style mutants [109] of 9 real repositories at pinned commits (Java: commons-math, commons-collections, jsoup, joda-time, gson; Python: click, attrs, pydantic, sympy)	520,821	520,541	0.9995	0.9994 to 0.9995	2.3	0
L0 one token, repositories never tuned on	commons-text, commons-io, more-itertools, httpx	23,015	23,015	1.0000	0.9998 to 1.0000	2.0	0
L1 one statement, Java	ManySStuBs4J 2.0 single-hunk fixes [107]	24,072	9,288	0.386	0.380 to 0.392	3.5	0
L1 one statement, Java, full file	ManySStuBs4J 2.0 [107], parent-commit file with the project method and guard indexes	21,320	13,586	0.637	0.631 to 0.644	4.8	0
L1 one statement, Python	TSSB-3M v1.0.0, shards 14 and 15 [108]	157,089	27,266	0.174	0.172 to 0.175	4.8	0

Figure 12 plots the rows of the table with their intervals.

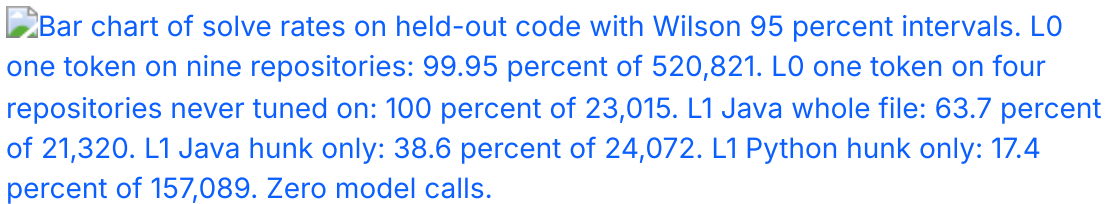
Bar chart of solve rates on held-out code with Wilson 95 percent intervals. L0 one token on nine repositories: 99.95 percent of 520,821. L0 one token on four repositories never tuned on: 100 percent of 23,015. L1 Java whole file: 63.7 percent of 21,320. L1 Java hunk only: 38.6 percent of 24,072. L1 Python hunk only: 17.4 percent of 157,089. Zero model calls.

Figure 12. The repair ladder on lifted code. Data figure: held-out test split, frozen engine, location given, token identity gate, 0 model calls; Wilson 95% intervals computed from the counts. Sources: PIT-style mutants [109], ManySStuBs4J 2.0 [107], TSSB-3M v1.0.0 [108]; per-task rows in [bench/ladder](#). A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

L2 to L4 are measured on the Defects4J held-out set of 288 bugs ([Section 8.1](#)). L5 to L7 follow the protocols of [Section 8.2](#). The method, the gate and the energy of each row follow.

Open issue: baseline for the one-token rate. The L0 rows measure how often a prior-ordered one-token neighbourhood restores a one-token mutant. If the neighbourhood is built from the same operator tables that produced the mutants, a high rate partly reflects that overlap and not repair ability on faults that arise in practice. The paper cites this coupling risk and gives no number for it. A plain operator-inversion search, run on the same mutants, is the baseline that would show how much of the rate it explains. That baseline has not been run, so no figure is given here and the one-token rate is open to this concern.

The rates read as intent bits. At L0 the missing bits are almost all on the line, and a prior-ordered single-token neighbourhood supplies them in a median of 2.3 bits. On the hunk-only L1 row the engine sees only the 7-line hunk. It closes the categories whose bits are inside the hunk (Java: changed operator 91%, swapped boolean literal 84%, swapped arguments 82%; Python: changed boolean literal 98.8%, swapped arguments 99.4%) and leaves open those whose bits are outside it: method names (22%) and identifiers (37%) need the file and its types, and new conjuncts (3% to 7%) and whole-statement rewrites (Python, 1.2% of a category that is 34% of the corpus) are new intent. New intent is what a supplier (a specification, a test, a person or a local model) provides once, after which the fit is in the index.

Method. Each corpus is shuffled with a fixed seed. The first 2,000 positions of each shuffle, and for L0 three whole repositories (commons-lang, requests, flask), form the development split, on which the gears may be tuned; development results are not reported. Every remaining task is in the test split. Before a test run the engine is committed and its binary hash recorded, and no gear changes during the run. Every row in this section is a frozen-engine result on test tasks only, with 0 model calls. The preregistration (PREREG. json), the per-task rows and the report script are in the [public repository](#).

Gate. At L0 and L1 the location is given and the gate is token identity: the repaired line or statement must equal the developer's, token for token. This is stricter than passing the tests, because a patch can pass a weak test suite and still be wrong. Qi and colleagues found that most patches GenProg reported as fixes were plausible but incorrect [92]; overfitting to tests is documented in search-based and semantics-based repair [93, 94] and addressed with generated tests [95]; tools score higher on the benchmark they were tuned on [96]; and fault localization biases comparisons [97]. Token identity removes the plausibility gap on these rungs, and giving the location removes the localization term, so the rate measures the repair gears alone. L2 and above use the tests as the gate and report plausible and correct counts separately (Section 8.1).

6.1 L0: one-token faults

The L0 corpus is every PIT-style mutant [109] with a valid replacement token in nine real repositories at pinned commits, five Java and four Python: 520,821 test tasks out of 547,448. The engine (commit 65571d0, gears G0-T and G1-J) restores 520,541, a rate of 0.9995 (95% Wilson interval 0.9994 to 0.9995) at a median of 2.3 search bits. Of the 280 unsolved, 108 are mutated digits inside floating-point literals and 2 are inside Python set literals. A float-literal

gear (G0-F, commit ecfe5a1) is tested on four repositories never used for tuning (commons-text, commons-io, more-itertools, httpx): 23,015 of 23,015 tasks, 1.0000 (0.9998 to 1.0000), a median of 2.0 search bits. Mutants are a stand-in for real one-token faults that Just and colleagues found to be coupled with real faults [110].

6.2 L1: one-statement bugs from real histories

L1 uses two public datasets of real single-statement bug fixes mined from open-source histories. ManySStuBs4J 2.0 [107] gives 24,072 Java test tasks; the engine (commit f2701b7) resolves 9,288, a rate of 0.386 (0.380 to 0.392). TSSB-3M v1.0.0 [108] gives 157,089 Python test tasks from shards 14 and 15; the same engine resolves 27,266, a rate of 0.174 (0.172 to 0.175). The engine receives the 7-line hunk only, so that Java rate is a hunk-only result. Given the whole file at the parent commit, with indexes of the project’s methods and guard conditions and the gears for changed method names, added arguments, guard conditions, throws and modifiers (commit 62fb7a0, binary sha256 prefix 3682c6515acdf722, preregistered before the run), the engine resolves 13,586 of the 21,320 Java test tasks whose file and bug statement could be located, a rate of 0.637 (0.631 to 0.644). By category it closes changed operators (95.9%), changed operands (92.1%), wrong callers (87.9%), swapped boolean literals (88.4%), deleted arguments (85.1%) and swapped arguments (83.3%); changed method names reach 61.4% and added arguments 51.4%, and new or narrowed conditions (43.5% and 51.7%) remain the main open intent. The full per-stratum tables are in the repository. These datasets have no published repair leaderboard under this gate (hunk only, token identity, location given), so no learned-repair number is set beside them; Defects4J, which does have one, is compared with TBar in Section 8.1.

6.3 Energy

Every row carries `est_j`, an estimate of CPU seconds times 10 W: 0.40 J per solved task at L0, 0.13 on the fresh repositories, 0.10 on ManySStuBs4J and 0.27 on TSSB-3M. A subsample was read on RAPL [105], the processor’s on-chip package counter, on a Linux laptop with an Intel i9-13900H (hostname jetson-hub; not an NVIDIA Jetson). The harness ran on 4 pinned CPUs, and `reported_j` is the package delta over the whole run, Python driver included, less idle power from a 10 s sample taken just before, times the run’s duration. Continuous-integration runners were up, so idle was 40 W to 44 W and the uncertainty is about ± 1 W times the duration, ± 10 J on these totals.

Subsample (test split)	Tasks	Solved	Raw J	reported_j	reported_j per solved task	Seconds
L0, gson mutants (seeded, first 3,000)	3,000	3,000	648.3	167.5	0.056	11.8
L1, ManySStuBs4J positions 2,000 to 3,999	2,000	690	377.2	17.9	0.026	8.2

The L1 subsample builds its project index from its 2,000 tasks only, so its solve rate (34.5%) is below the full run's. `measured_j`, a wall meter on a named rail, is empty in every row; its protocol is Section 8.3.

Two regimes and a calibrated vector. The estimate above, CPU seconds times 10 W, is one substrate table and cannot tell targets apart, so least-joule extraction never chooses a target. The measurement literature says what the fix must look like. On an in-order core, joules are linear in operation counts: a six-counter model on Cortex-M0 reaches $R^2 \approx 0.99$ and 2.8 to 4.3% mean absolute error, with coefficients such as 0.96 nJ per instruction and 2.1 nJ per taken branch [152]. On an out-of-order core the model collapses to time: with one core pinned and turbo off, a Xeon Gold 6430 drew 189.8 ± 0.5 W across every implementation and benchmark of a language-energy replication, and the ranking that had put Python at 75.88 times C disappeared once library choice, parallelism, warm-up and measurement bugs were controlled [153]. The counters themselves are of uneven worth: Intel RAPL predicts plug power to 1.8% mean absolute percentage error once DRAM is included, but updates about every millisecond and drifted 10 to 12% with package temperature on one generation [105]; polling it faster than 10 ms produces up to 50% stale reads [154]; NVIDIA's NVML samples about a quarter of the run time and single naive runs erred by up to 70% against an external meter [155]. Gas-style unit metering correlates with CPU time at only Pearson ≈ 0.51 on Ethereum [156], so “energy scales with the operations counted” is an assumption the `reported_j` subsample has to test. A proposal to the specification keeps three objects apart: the C.6 budget units stay deterministic, target-blind and in the log; `est_j` becomes $\langle \text{count}(\text{term}), E[\text{target}] \rangle$ plus idle power times time in regime A (microcontrollers, field-programmable gate array (FPGA) lanes, in-order cores) and $P_{\text{active}} \cdot t + E_{\text{llc}} \cdot \text{misses}$ in regime B (out-of-order CPUs and graphics processing units (GPUs)); and the coefficient vector $E[\text{target}]$ is a `host.Calibrated` log entry with its protocol, sample count and standard error per coefficient, written by `rce host calibrate` under a normative protocol (at least 30 interleaved A/B pairs against the idle baseline, blocks of at least 60 s or 10^9 operations, warm-up discarded, temperature recorded, gross and net reported, the meter named with its validation reference, RAPL sampled no faster than 10 ms), so that extraction under `est_j` is a function of the log and replays. Receipts gain a `calibration hash` and a `stale_fraction`, the share of a step's operations with no calibrated row, priced unpriced; totals keep the four labels in separate columns. The falsifier is RCE-F5 in §9. The first calibration to run is the KV260 rung of the energy ladder on its core rail with $N \geq 30$, which turns the stand-in `E_LUT_READ = 1e-12` of the specification into a coefficient with a standard error.

7. Reuse evidence

RCE's position on the reuse axis rests on an engineering fact: code is reused at every scale. Each method below measures it differently.

Method	Source	Finding
File-level clone mapping	DéjàVu, Lopes et al. 2017 [33]	Of 428 million files in 4.5 million non-fork GitHub projects, 85 million are unique: “70% of the code on GitHub consists of clones of previously created files.” In JavaScript, 6% of files are distinct.
Line-level redundancy	Gabel and Su 2010 [34]	Against 420 million lines, “between 50% and 100% of each project is redundant” at about one line.
Statistical language models	Hindle et al. 2012 [35]	Code is “very repetitive, and in fact even more so than natural languages.”
Corpus near-deduplication	The Stack [36]	Near-deduplication removes 54.7% of The Stack v1 (6.4 TB to 2.9 TB) and 52.4% of v2 (67.5 TB to 32.1 TB).
Whole-archive preservation	Software Heritage [129, 130]	30.3 billion unique file contents from 448 million origins (archive counters, read 13 October 2026), each stored once by the hash of its bytes.
Mined fix patterns	PAR, TBar [22, 23]	PAR found “several common fix patterns” in more than 60,000 human patches. TBar’s 15 pattern categories fix 43 Defects4J bugs correctly in the standard pipeline, and 74 of 101 with perfect fault localization, with no learning. These are figures as read from [23]; the meaning of the 101 (plausible patches or bugs attempted) has not been rechecked against that paper here, so treat the ratio as unconfirmed.
Cross-language tasks	Rosetta Code [131]	1,360 tasks (and 421 draft tasks), each one concept implemented across many of 1,009 known languages.

 Five small panels, each with its own unit. Lopes et al. 2017: of 428 million files in 4.5 million non-fork GitHub projects, 85 million are unique, with the quotation that 70% of the code on GitHub consists of clones of previously created files. Gabel and Su 2010: between 50% and 100% of each project is redundant at about one line, against 420 million lines. The Stack: near-deduplication removes 54.7% of v1, 6.4 TB to 2.9 TB, and 52.4% of v2, 67.5 TB to 32.1 TB. Software Heritage: 30.3 billion unique file contents from 448 million origins, archive counters read 13 October 2026, each stored once by the hash of its bytes. PAR found several common fix patterns in more than 60,000 human patches; TBar fixes 43 Defects4J bugs correctly in the standard pipeline and 74 of 101 with perfect fault localization.

Figure 13. Code is reused at every scale. Data figure: five measures of reuse, each in its own unit; the panels are not comparable with each other. File-level clone mapping (Lopes et al. 2017 [33]): of 428 million files in 4.5 million non-fork GitHub projects, 85 million are unique. Line-level redundancy (Gabel and Su 2010 [34]): between 50% and 100% of each project is redundant at about one line, against 420 million lines. Corpus near-deduplication (The Stack [36]): 54.7% removed from v1 (6.4 TB to 2.9 TB) and 52.4% from v2 (67.5 TB to 32.1 TB). Software

Heritage [129, 130]: 30.3 billion unique file contents from 448 million origins (archive counters read 13 October 2026), each stored once by the hash of its bytes. Mined fix patterns (PAR [22], TBar [23]): PAR found common fix patterns in more than 60,000 human patches; TBar fixes 43 Defects4J bugs correctly in the standard pipeline and 74 of 101 with perfect fault localization. Values as in the table above. A [PNG copy](#) is provided. [Open full size](#) · [PNG](#)

Each row is an input to the engine. Clones and dedup say a fit usually exists, which is G0. Recurring fixes say adaptation is mostly a small catalog of guarded rewrites, which is G1. Line-level redundancy shrinks the search space of G2. The ecosystem measures it on every promoted fit (Section 2.3), and the repair ladder on lifted code measures it at scale on held-out splits (Section 6). Two cautions sharpen the reading. Duplicates inflate retrieval and model scores unless removed [104]. And buggy lines are less natural than fixed ones [45], so the bug itself is the case lookup is least able to predict; the fix shape, not the buggy line, is what G0 and G1 must key on. The rule-mining line that does this at scale, from GenProg’s generate-and-validate search [88] and Prophet’s learned correctness model [89] to Getafix at Facebook [90] and Sorald’s model-free repair of static-analysis violations [91], derives templates from disjoint history and reports precision on unseen cases [98]. That is the standard Section 8.4 adopts.

8. Measurement protocols for what comes next

Each protocol is fixed before its run: preregistration committed, seeds and solver commit recorded, every artifact kept. Gate first, joules second, labels always.

8.1 Defects4J

Defects4J holds 854 active real Java bugs, each with reproducing tests [37].

1. Develop on a 100-bug subset, offline, through `defects4j checkout`; report held-out results only, on a disjoint set of 288 bugs frozen before the run (`FROZEN-heldout-v1`).
2. Run G0, then G1, then G2, then a local model last. Record the gear that closed each bug.
3. Gate: the triggering tests and the full suite pass. Report plausible and correct counts separately; correct means equivalent to the developer patch on review.
4. Report model calls per correct fix and joules per correct fix, labeled.
5. Reference points: TBar’s 43 correct fixes and its 74 of 101 under perfect localization (figures as read from [23], with the denominator not rechecked here). This section is a protocol. No Defects4J result of this engine is reported in this paper.
6. Contamination check: flag every bug whose fix appears verbatim in G0’s index, and report with and without flagged bugs.

8.2 SWE-bench Lite, offline

SWE-bench Lite is a 300-task subset of real GitHub issues resolved by hidden tests [38].

1. Use the official Docker harness with no network.
2. First run: G0 and G1 only, zero model calls. This is the floor of solving without a model.
3. Second run: add a local model as the last gear.
4. Report the resolve rate, model calls per resolved task and joules per resolved task, labeled.
5. Contamination check as in 8.1. SWE-Bench+ found that 32.67% of successful patches had the solution in the issue text [39].

8.3 A wall meter

1. Attach a wall meter or a shunt meter on the host's supply, logged at a stated rate with its clock synchronized to the run log.
2. Idle machine: CI runners and other builds stopped, and a quiet window gate checked before each block and recorded.
3. Subtract a measured idle baseline, and state the method.
4. Fill `measured_j` from the meter alone. Report `reported_j` (RAPL) and `est_j` beside it in separate columns. Never sum across labels.
5. Repeat the RAPL subsamples of Section 6.3 under the meter, then the Defects4J held-out set.

8.4 A held-out run with the gears frozen

This is the method of Section 6, extended to new corpora.

1. Freeze G0, G1 and G2 at a tagged commit, record the binary hash, and publish the rule set with its authoring timestamps.
2. Draw a disjoint held-out slice: the Rust and Python instances of Multi-SWE-bench [102] and BugsInPy [103], de-duplicated against G0's index [104].
3. Preregister the slice, the seeds and the hypotheses before the first run.
4. Report resolved, plausible and byte-identical counts separately, each with a 95% interval.

8.5 Baselines in the same table

1. Model-free: cargo fix and clippy [106], and Sorald or a TBar port [91].
2. Cheap model: Agentless with a local model [99].
3. Remote model: an agent with a frontier model, its energy estimated from provider disclosures and labeled `est_j`.
4. Report cost-of-pass [101] in joules and in dollars for every row, with cost controlled as Kapoor and colleagues require [100].

8.6 Convergence of model calls at large N

Promotion is the mechanism by which the model leaves the loop: a resolver is paid once for a new case, the gate admits the fit as a handler, and every later case of the same shape closes in G0 or G1 ([Section 4.2](#)). Convergence is the claim that model calls per round fall to zero on a

fixed specification under live traffic. It is measured as follows.

1. Draw at least 1,000 specifications from public repositories, stratified by language and size, and freeze the engine and the resolver before the first round.
2. Replay recorded production traffic, at least 10,000 events per specification, in rounds of fixed size.
3. Record per round: new escalations, model calls, promotions, deferrals and joules per closed task, each labeled (est_j, reported_j, measured_j).
4. Primary endpoint: the share of specifications reaching a round with zero model calls within 20 rounds, with a 95% Wilson interval. Secondary: model calls per closed task by round, and the round of the last model call.
5. Report specifications that never converge with their cause (a stuck input, a missing dependency, new intent), and run stuck-input deferral as a preregistered arm.

Proposition 8 (floor approach). As G0 and G1 coverage grows, joules per solved task approach the gold floor of applying a known fix and running the tests. The ladder rungs and the Defects4J held-out set under the wall meter test it.

8.7 Lift coverage across languages

Lift is measured as it is built. For language L , a program counts as lifted when $\pi_L(\lambda_L(p))$ passes the program's own tests and replays its recorded traces byte for byte, and $\lambda_L(\pi_L(\lambda_L(p)))$ has the same shape hash as $\lambda_L(p)$. The sample is at least 10,000 programs per language for at least ten languages, drawn with a fixed seed from pinned public corpora, with the lifter frozen and its binary hash recorded before each run, and 0 model calls. Each run reports the share lifted with a Wilson 95% interval, and each construct not yet lifted is a work item for the lifter of that language, built and frozen before the next run. The target is every program in every language.

9. Falsifiers

Fixed before the runs they test.

ID	Observation	Defeats
RCE-F0	On a preregistered run of at least 10,000 Forms, any projection prints a trace that differs from the Form's reference trace, or any malformed input is accepted	The claim that projections are faithful and the gate refuses at entry
RCE-F1	On the held-out slice, G0 to G2 with frozen rules resolve under 10% of tasks with 0 model calls	The claim that most fits already exist for repair at this scale
RCE-F2	Byte-identical or equivalent patches are under half of the resolved held-out tasks	The reading of the gate as more than plausibility on the covered tasks

ID	Observation	Defeats
RCE-F3	Under the wall meter, gross joules per solved task of the RCE solver exceed those of the cheapest model baseline on the tasks both solve	The claim of lower energy per solved task on covered tasks
RCE-F4	In the convergence protocol of Section 8.6, fewer than half of the specifications without a stuck input reach a round with zero model calls within 20 rounds	Compiling the model out of the loop
RCE-F5 (registered 22 October 2026)	On the ladder subsample under the wall meter of Section 8.3, with a host.Calibrated entry for the host, the per-target R^2 of reported_j on the C.6 count vector is below 0.8 in regime A, or est_j in regime B is off by more than 25% at the median	The claim that the calibrated estimate is a usable proxy; the receipt's basis for that host becomes unpriced until a better model is calibrated
RCE-F6 (registered 22 October 2026)	In the registered dimension measurement of Section 3.8, the median redundancy over bounded, output-bearing programs (the stratum the registered runs call observable) is below 0.1 bit. Status, 24 October 2026: triggered on the authored two-region population of the fifth run (median 0.00 bits over twelve programs), not triggered on the hand-written examples (1.29 bits, third run) or the host programs (0.46 bits by family, fourth run)	The claim that accidental state is common in small Forms at this scale; rce dim is then a diagnostic for large programs only, which is the reading Section 3.8 now gives

10. Work to finish

Each item has its protocol named.

1. Lift built and measured across at least ten languages under the protocol of Section 8.7.
2. The spec arm: specifications in the style of MBPP and HumanEval authored as Forms and emitted to every target.
3. Held-out results on L2 to L4, the Defects4J set of 288 bugs (Section 8.1), and on L5 to L7 (Section 8.2).
4. Baselines in one table, cost-of-pass in joules and dollars (Section 8.5).
5. A measured_j series: a wall meter on an idle host, at least 10 repeats, gross and net side by side (Section 8.3).
6. The 626-language survey published with its method (Figure 1).
7. New held-out corpora with the gears frozen: Multi-SWE-bench and BugsInPy (Section 8.4).
8. Convergence of model calls across at least 1,000 specifications (Section 8.6).
9. T60, T61 and T59 over all of E proved (T60 by mechanised partition refinement), and T62 linked from its model to the admitter's order (Section 3.8).
10. The T7 state_hash finding is resolved (Section 3.6) and RCE/Positional.lean proves the positional embedding the checker hashes unchanged under the renamings of specification 10.6 for every hash function. What remains is a refinement from the Lean

model to the Rust checker itself, which is the Phase 2 gate of Section 13.

11. A dimension population registered blind (programs written without the benchmark in mind, or lifted), large enough to test H2 beyond the two rows that now carry it; the five registered runs recorded RCE-F6 as triggered on one population and not on the others (Section 3.8).
12. The three preregistered experiments of Section 12.3: model accuracy on the nameless Form with and without the concept record; the commutation rate of adjacent statements in lifted code; and whether replay-certified reversible deletion moves the 20% deletion baseline.
13. The first host.Calibrated entry, on the KV260 rung with $N \geq 30$, and the regression that RCE-F5 tests (Section 6.3).
14. Form IR format 2 (content-addressed definitions) made the only format, so that the 77 standard-library bricks refused for term size alone lower, and the round-trip condition $\text{hash}(\text{lift}(\text{emit}(F))) = \text{hash}(F)$ checked per target at $N = 10,000$ (Section 13).

11. Conclusion

RCE is a native code ecosystem. A program is authored once, from intent, as a Form: a lossless encoding of state, events and pure transitions, and every target language is a faithful projection of it. Invalid states cannot be written and malformed inputs are refused at entry, so a program in the ecosystem cannot drift from its intent and needs no after-the-fact diffing or error tracking. Every fit that passes the gate is stored and reused, and the ecosystem compounds under a gate whose correctness is monotone. On 10,000 preregistered Forms with a frozen engine and 0 model calls, every Form round trips to identical bytes, every projection is deterministic, every malformed input and specification is refused at entry, and the Python projection replays the reference trace on 9,999. Code from other languages enters by lift, a construction from each language's grammar to its syntax tree to the Form; on lifted code the repair gears restore 99.95% of 520,821 one-token faults and 63.7% of 21,320 single-statement Java bugs. Sections 3.6 and 8.7 state the universality propositions and how their construction is measured.

12. The Form against the record

This section places the Form against three bodies of evidence: the theory and tooling of each area the design touches, the history of programming-language design read as a comparison of what each language fixes and what it leaves free, and the measured causes of redundancy in human-written code. Each subsection states what the evidence shows, with its source, and then the position this paper takes and the reason. Where a position is a judgement rather than a measured result, the text says so. The three surveys behind this section were conducted in October 2026; their full texts and working notes are published in the experiments repository (openie-research-experiments/rce/surveys).

12.1 Seven areas of the design

Canonical forms. For a complete, deterministic machine over a finite alphabet the minimal machine is unique up to isomorphism, and a fixed traversal order turns that into a unique string [150]. The Form supplies completeness and determinism (T1, T2), so the finite-control projection of §3.8 has a canonical form. The data component does not: symbolic finite automata have a canonical minimum up to predicate equivalence [149], register automata over equality or dense order admit a Myhill–Nerode theorem [151], and a state updated by integer arithmetic has no canonical representative but the term itself. Extraction from an e-graph under a cost that charges shared sub-terms once is NP-hard and not approximable to a constant factor [159], so a hashed representative must use a cost that is reproducible across extractors. Position: three shape keys, each naming its tier in the hashed input (the name-free term hash, the control form of §3.8, and the e-class key with its saturation budget); equivalence defined once, as bisimulation; guarded step bodies checked first by the nearly linear decision procedure of guarded Kleene algebra with tests [158]. Refused: any claim that the minimal machine is *the* canonical form of a Form with arithmetic state; a hashed representative chosen by shared-sub-term cost. Open: for which fragment equal meaning implies an equal control form (the converse of T61).

Content-addressed intermediate representation. Unison’s five-year experience with hash-identified definitions reduced to one invariant, that the hash covers semantics and never names, and one cost, re-hashing dependents under edit [160]; Dhall hashes the normal form of a total language [161]; Nix’s attempt to content-address build outputs stayed experimental for fifteen years because self-references and non-determinism broke it [162]. Position: definitions keyed by the digest of the α -erased, normalised body, with names, source positions and literal spellings in an unhashed table rendered by the projections; one emitter that walks the intermediate representation through a per-target table, replacing four emitters that walk the surface syntax; every runtime that serialises checked against the canonical-JSON vectors. Refused: replacing canonical JSON as the hashed bytes with RFC 8785 (which cannot carry an integer distinct from a float, negative zero or an exact 64-bit unsigned integer) or with deterministic CBOR (whose two specifications disagree on float width and key order); content-addressing emitted artefacts as identity; full β -normalisation before hashing, since a recursive function then has no finite term. Open: component-level keys for mutual recursion.

The log as a partial order. A trace in the sense of Mazurkiewicz is a word modulo the commutation of independent events, and all linearisations of one configuration read off the same trace. The consistency-as-logical-monotonicity theorem states which programs need no coordination [19], and its extension to reads says that observations of convergent state are unsafe unless they are threshold-shaped [82]. Position: an entry of the log hashed over its chain, sequence number, event and the hashes of the entries it causally observed, with the state hash recorded only at cuts, so that the digest is the same under every linearisation the semantics admits; the monotone tag extended to reads; every taken arm of a cross-chain select logged. The theorem that any order consistent with the dependencies folds to the same

cut state (T62) is proved on a model and is a proposal, not normative text. Refused: letting a target choose the linearisation of a non-monotone event. Open: the extension of the model proof to the full evaluator.

Synthesis from intent. A verifier checks only soundness: a postcondition that is always true verifies. The completeness of a specification has to be measured, and the measure that works is the share of mutated outputs the specification rejects; removing that measure from the gate in the study that introduced it dropped verifiability to 1.56% [164]. Type-directed enumeration over equality-constrained tree automata raised the solved share of a library-search benchmark from 64% to 88% [165], and self-refinement and majority voting rarely justify their cost once cost per correct result is accounted for [166]. Position: intent as a program with holes carrying obligations, examples, a named set of usable definitions and a budget; a rule that refuses an intent whose obligations reject no behaviour; a measured completeness reported with every intent; a derivation record per hole giving the candidates before and after each check. Refused: promotion by score; refinement loops inside one model call. Open: a corpus-independent measure of the information an intent carries.

Energy cost models. The measurement literature is summarised in §6.3: joules are linear in operation counts on an in-order core [152] and linear in time on an out-of-order core [153]; on-chip counters differ in worth by an order of magnitude [105, 154, 155]; unit metering of the kind used by Ethereum correlates with CPU time at only 0.51 [156]. Position: three distinct objects, never merged — the deterministic, target-blind budget units of the specification; an estimate that is a calibrated per-target coefficient vector applied to the counts (in-order) or to time (out-of-order); and the receipt labels, kept in separate columns. Refused: joules in the budget table; per-language constants; a counter read as a wall-meter measurement.

Languages designed for machine authors. A catalogue of 42 such languages, 36 of them first listed in 2026, shows every one of this paper's mechanisms with a peer: hash-chained evidence, mandatory contracts split into proved and run-time-checked, a proof-assistant core, syntax trees as the source of truth with hashed definitions [168]. The convergence observation of §3.7 stands. What distinguishes this design is not any mechanism but that replay and injectivity are theorems (T3, T7) rather than run-time promises, and that examples are typed event traces over the same semantics replay uses, so the oracle a model cannot bypass is the program itself. Position: a versioned registry of rule codes with machine-readable diagnostics and repair plans; test-validated translation pairs manufactured from the emitters and the checker, which is the recipe that recovered low-resource languages in [167] run in reverse. Refused: any claim to be first or only.

Proof engineering. The standards that closed the gap between a specification and its mechanised model did so by generating prose, checker tables and definitions from one authored source: SpecTec for WebAssembly found ten errors in five proposals and was adopted for all future editions [169]. The projects that proved an implementation correct by refinement paid for it: about twelve person-years for 8,700 lines of C [171]. CompCert's maintainers list every trusted item with its reason [170]. Position: one authored rule source generating the specification text, the checker tables and the Lean definitions, with a check that

fails on any divergence; the Lean core as the executor of record for the subset it covers, with the count of unsupported constructs published and bounding every theorem's stated scope; a chapter listing every trusted component with its test. Refused: a refinement proof from the Lean core to the Rust interpreter; theorem counts as a measure of assurance; two authored semantics of anything.

12.2 The Form among programming languages

About a hundred languages in twelve families were placed on eight coordinates — state layout, event alphabet, transition, order, resource, trust, binding time and intent — recording for each whether the author fixes it, the compiler or runtime chooses it, or the language has no word for it. Three regularities follow. Only three families obtain global determinism without the author's effort, by three different theorems: the synchronous languages by a causality check, the stream and logic languages by monotonicity, and guarded-atomic-action languages by one-rule-at-a-time scheduling with static conflict analysis. Every family that reached safety certification did so by removing the same three coordinates the Form removes: dynamic allocation, unbounded recursion and nondeterministic choice. And the two coordinates no family fixes well are trust and intent; the Form's intent and example blocks and its capability values fix both.

The five designs nearest the Form are Dedalus, whose time-indexed relations are the log and whose next-step rules are the transition [15]; Bluespec, whose guarded atomic rules are scheduled by a compiler-chosen order proved equal to one-rule-at-a-time execution; the Elm architecture, which is the Form's four components under other names [83]; Lustre, whose bounded-memory, statically scheduled discipline has been certified in about a hundred avionics systems; and Unison, on identity [160]. The record also fixes five constraints, each a place where an earlier language paid for the opposite choice. Concurrent tasks must lower to state machines without a stack, as Rust's asynchronous functions do, because a stackful coroutine cannot be logged step by step. Parallel composition must be emitted as a structured step, never as an explicit product of automata, which in Esterel's compilers failed beyond about a thousand lines of concurrent source. Effect handlers must be one-shot, as OCaml 5 chose, and the reason is ownership and the totality bound (each additional resumption copies captured state a data-dependent number of times), not loggability. The only dimension of a program that is a theorem is the index of its finite-control projection (§3.8). And the lift from a Turing-complete language is a partial map whose domain is undecidable (§3.6). The vocabulary of differential geometry that an earlier working draft of this project used (charts, gauge, holonomy, geodesics) has no counterpart in the literature of programming languages and is not used here; the objects that exist are the final coalgebra, functors from syntax into it, the groupoid of rewrite proofs whose connected components are e-classes, and minimum description length with joules in place of bits.

The universality propositions of §3.6 are made measurable by an encapsulation criterion. The Form encapsulates a language L when, for every program p of L in a fixed corpus, the lifted Form is no worse than p on five coordinates and strictly better on at least one: joules measured on the same tasks and host; redundancy in the sense of §3.8 against the state bits p declares;

decidability, meaning every property decided for p is decided for the Form; the round trip, meaning behaviour agreement to a stated length and the lens condition on the shape hash; and the cost for a model to reach a gate-passing program from the same intent. The claim that the Form is better than a programming language is the conjunction of this criterion over the corpus, one registered run per language, and nothing else: it claims neither expressiveness, since a Form cannot say more than p , nor readability, which is a property of a projection, nor adoption. A language on which the Form is strictly worse on any coordinate after two registered runs is not encapsulated, and the record of what that language fixes that the Form does not becomes a work item for the Form.

12.3 Why human-written code is redundant

Human-written code carries redundancy wherever the medium gives the author no channel for what is meant, and the redundancy is each author's private substitute. The measured evidence sorts it into three layers. Formatting is 13 to 15% of the tokens of brace-delimited languages and can be removed without loss of accuracy for ten language models [136], and layout has no measured effect on human comprehension [135]. The only quantities that track measured comprehension load are textual size, vocabulary size and the reach of data-flow dependencies, each at Kendall's $\tau \approx 0.4$, while cyclomatic complexity tracks nothing ($\tau \approx 0.09$) [133]; and 97% of real concurrency bugs other than deadlocks are unenforced assumptions about atomicity or order, 96% of them involving at most two threads, because sequential text cannot state a partial order and the author writes a total one [172]. Names are chosen with a median agreement of 6.9% between developers and decoded at 97% [137]; comments stop tracking the code they describe within a few commits [138]; yet language models lose up to 29 points of accuracy when identifiers are removed [139]. Two further causes are organisational. Developers do not delete: in three production systems 54 to 60% of files were never executed over six to sixteen months, developers removed 20% of the code they were shown to be unnecessary, and the caution was reasonable, since the detector was 32% precise [173]. And organisations ship their boundaries: eight organisational metrics predicted failure-prone binaries at 86.2% precision and 84.0% recall, ahead of every code metric [174].

The position that follows has four parts. The Form takes order, locality, idiom and reachability, each decidable over a state, an event set and a total step. The projection takes layout and idiom, rendered per community, because an expert's advantage is confined to code written in familiar plans and a projection that emits merely correct code costs them it. Intent stays a structured, checked record, from which the projection renders names (§3.8). And the specification is governed by a mechanical admission check, because authorship by models raises the rate of proposals without changing the rule. Three questions on which the design depends had no measurement in the literature and have been registered as experiments with sealed protocols: whether a model reasons as accurately from a nameless Form with its concept record as from a named surface; what share of adjacent statements in ordinary code commute under the Form's dependencies, which is the size of the slack whose cost the 97% figure measures; and whether deletion that is checked by replay and reversible by one command moves the 20% deletion rate. Two claims that circulated in the project's working

material have no traceable source and are not used: a 2014 study of language-manual page counts, and the statement that experienced programmers write less code, which traces to a weblog post citing an unnamed study.

12.4 How the language grows and shrinks

Every language that stayed small replaced the taste of reviewers with a rule about what counts as admissible. Go's process is slanted towards declining, and in one study declined 40.9% of 1,095 proposals with a published set of reasons [175]; Lua's rule of mechanisms rather than policies held its implementation at about 25,000 lines of C; Elm stopped. The regime that grows is also documented by its own chair: the benefit of a feature is private and immediate, its cost diffuse and deferred, and it is easier to summon enthusiasm for something new than for opposing it [175]. In RCE the Form grows inside the one text. When a program needs a form the text lacks, including one nobody foresaw, the form is written into the specification itself, with no version and no macro, flag, directive or plugin. The specification repository states seven rules for how a form enters the text, each naming the artefact that enforces it. A construct enters the normative grammar only with an accepted and a refused example, a Lean statement of the theorem it relies on linked to the checker by a generated rule table or by test vectors, and no parsing conflicts; anything less is a proposal. A second spelling of something the Form already says is declined on that ground alone. Change runs both ways: any construct can be changed, generalised, replaced or removed when a better form is found, and the change moves together with every program, example and library module of the repository it touches in one commit, with a ledger row saying what changed and why. Each interface has one decision locus, because an organisational boundary that runs through an interface is a predicted defect site [174]. The decisions and their reasons are published in a ledger with a fixed set of reasons, so the decline rate is a number anyone can read. The market does not prune, so use is never a reason to keep a construct. And authorship by a model changes the rate, not the rule.

12.5 The library and the compute fabric

Whether a language should ship a large library ("batteries included") or a small specified core with a free registry is usually argued as convenience against flexibility. The record shows both positions failing at the same point [193]. A library shipped with the language ties its interface to the language's compatibility promise and loses flexibility [195]. Its modules are removed once an installer exists: Python Enhancement Proposal 594 removed 22 of them, citing maintenance cost and the ease of installing from the registry [194]. A free registry produces deep transitive graphs: half of the dependent packages in Cargo, npm and NuGet have at least 41, 21 and 27 transitive dependencies [196]. A specification with many implementations diverges at its edges, as more than 30 JSON parsers do on one under-specified grammar [197]. Where the record converges, it converges on a curated tier that is delivered and changed as one unit.

The same failure appears across compute fabrics. On graphics processors, single-source models match native code on streaming kernels and lose on reductions: on the dot product, the performance-portability scores of Kokkos, RAJA and SYCL lie between 0.80 and 0.91

[198]. On an FPGA, one SYCL source migrated from CUDA and tuned for a GPU reaches 0.2% to 47% of the speed of its FPGA-optimised version, by ratios derived from the published timings at the largest problem size [193, 199]. The surveyed systems that give one meaning in software and in hardware achieve it by cutting their source language down to what their hardware can mean (state of statically known size, no heap or recursion in the hardware part, a logical tick), and their proofs cover parts of the chain, not all of it [193, 200]. Portable code formats work where they fix semantics [201]. Bit-identical results cost little when designed in [202]. Soundness results for memory models are pairwise [203]. Placement from measured costs is nondeterministic unless each decision is recorded and replayed, or the run is simulated in virtual time [193, 204]. For machine authors the question is also one of knowledge: of the packages named by code that 16 models generated in Python and JavaScript, 19.7% did not exist [205].

Both debates ask where a library's meaning is held: by one implementation, by each registry package, by a specification checked with tolerances, or by a vendor binary for one fabric. The RCE position holds it in the one text.

- **The library is part of the text.** It is written in the language and built into every tool (specification 4.1). It is pinned by the hash of its manifest, which a program names as its one dependency (F.5.3).
- **It has no platform variant.** It performs input and output only through adapters.
- **Its cost is the cost of its source on every binary.** A step therefore means the same on every host (HC-7).
- **The per-target choice belongs to the engine.** The engine admits a fabric for a class of operations only after a logged bit-exactness probe against the CPU reference (HC-13), and it logs every placement for replay (HC-12).
- **What is available is a list a tool prints, complete at each commit and growing in the same text.** Whether a function exists becomes a lookup rather than a recollection.

The Form does this as a shape, not as a gate. The systems of the record cut their source language down because it says more than their hardware can mean. The Form $P = (S, E, \delta, D, \lambda)$ is the shape every program already has: the state it keeps, the events it answers, its step, its log and its outputs. That is why programs of other languages enter it by lift, within the domain §3.6 states, and why no part of this position sorts programs into admitted and refused ones.

The Form is rigid at each commit, with one text and one meaning. It is flexible across commits. When a program needs a form the shape does not have, the shape gains it in the text, never through a macro, flag, directive or plugin beside it. A change may extend, generalise or replace a form. It moves together with the programs of the repository in one commit, and the program hashes, the equivalence check of §3.8 and the log record what moved. Behaviour and equivalence (specification 6.10) and the host's fabric entries were added inside the text without changing the meaning of any accepted program, and the log as a partial order and intent with holes are proposed the same way. Under the proposal, hardware follows the same rule: the engine reads a Form's state sizes and step costs to realise it, with a statically sized state in registers and a growing state in memory the circuit addresses.

Batteries included and specification-and-free then stop being alternatives: the batteries are part of the specification, and the program is free of any fabric.

The library half of this position holds in the repository in its main parts: the library is written in the language, built into every tool and pinned by hash. Two of its parts remain: one definition of the standard library across the sections of the specification, and use as the one way to bring in the four modules that a directive still reaches. The fabric half is a contract and a reference executor proved to replay, exercised on CPUs only. The proposal “one library and one Form on every compute fabric” lists what remains [193]:

- one definition of the standard library, and use as the one way to bring a library in;
- probe evidence re-derived on replay;
- placement on every execution path;
- one reduction tree per construct;
- calibration runs;
- integer and binary32 operation classes;
- hardware realisation read from each Form’s shape;
- a host without an operating system;
- two measurements the record lacks: one kernel’s energy across device classes under one boundary, and the share of library functionality that model assistants reuse rather than reimplement.

13. Programme of work and its gates

Two independent reviews of the public repositories in October 2026 reached the same diagnosis. The checker is not itself verified: nothing refines the Lean semantics to the Rust interpreter, the two Lean cores have no theorem relating them, and the conformance suite reports 122 tests passing under the mechanised core and 1,867 unsupported, so about 6% of the language is under the theorem the overview advertises. Emitted code is a dynamic-value runtime per target, four hand-written runtimes of about 2,000 lines each. The energy estimate cannot tell targets apart, so least-joule extraction never chooses one. The lift covers four languages and the universality claim needs ten. And the engine — the gears, the e-graph, the proved rewrites, the verified kernel and the lift — is not public, so the headline numbers of this paper cannot be rebuilt from public code; the note at the head of this paper says which. For a project whose stated reason for working in the open is traceability, that is the largest risk in the programme.

The programme is ordered by dependence: reproducibility first; then the encoding and the cost model that every later measurement rests on; then the proof chain; and only then breadth. Each phase ends at a gate that is a falsifier registered before the run, reported with a confidence interval; a phase closes when its gate observation is published, never on effort. The gates are stated in §9 and the table below.

Phase	Deliverables	What defeats the phase
0 Reproduce	The engine public at a tagged commit, or every number that depends on it moved to §10; the per-task records of the 10,000-Form and 21,320-bug runs; the held-out Defects4J set completed; every registration sealed by hash; one version string; the specification errata closed; one spelling each of termination, authority and error; a published measure of surface redundancy (distinct constructs per Form over the example corpus)	A number in this paper that cannot be rebuilt from a public commit with one command
1 Freeze the Form	Content-addressed definitions as the only intermediate form; typed relations as a Form value; one lowering that every emitter consumes; the round-trip condition checked per target; a per-target cost model calibrated as §6.3 states	A definition refused for size; a target whose round trip changes a shape hash; the cost model ranking two targets opposite to the wall meter
2 One core, verified gate	A single Lean core with T1 to T9 restated on it; the typing rules generated from the checker's rule table so that a rule without a Lean counterpart fails the build; the checker and the interpreter extracted from Lean and used as the gate, with the Rust checker as a cross-checked front end	The extracted gate and the Rust checker disagreeing on any program; a restated theorem needing an admitted step; unsupported conformance tests above 10%
3 Targets priced in joules	Rank-n arrays and bulk operations as the one spelling for numeric work; Zig and Fortran emitters with fixed floating-point and overflow modes; layout hints in the fit and never in the Form; a translation-validation certificate per emitted module; the wall-meter protocol of §8.3	RCE-F3 and RCE-F5; a module emitted without a certificate; extraction choosing the same term for every pair of targets
4 Lift as the entry	A lift per language, Go and Python first, then TypeScript and Rust, then C and Zig; 10,000 programs per language with the lift frozen; the encapsulation criterion of §12.2 run per language; a public, append-only log of admitted shape hashes with their certificates	Lift coverage under 10% on any of the first four languages after the second registered run; the Form strictly worse on any coordinate of the criterion
5 The ecosystem compounds	Synthesis from intents with holes; the first gears public with their verification certificate; the convergence protocol of §8.6 on 1,000 public specifications; baselines in one table with cost per correct result in joules and dollars	RCE-F1, RCE-F2, RCE-F4; the tools unable to check or emit themselves

Three positions where the reviews differed from the surveys of §12 are resolved as follows. Canonical JSON remains the only hashed encoding, for the reasons given in §12.1; a binary encoding may mirror it for transport only if it is a bijection to those bytes, checked on the same vectors. The one spelling of termination is a declared decreasing measure, with the loop and recursion limits as the semantics; the content-addressed form still needs a bounded fixed point for recursion, and the measure is what bounds it. Reducing the concurrency vocabulary

to two primitives is recorded as an open proposal rather than a decision, because the theorems of the two primitives must first be proved on the checker rather than on a model. Not on the programme at all: a self-hosting compiler before phase 2, further targets before phase 1, any construct of the appendices before its theorem, and a production compiler, which the specification's own overview says should not be built until the document stops moving.

Since the reviews, the six repositories that must agree with the specification — the book, the example programs, the standard library, the conformance suite, the grammar package and the editor — have been merged into the specification repository with their histories, and each is checked by a workflow that builds the checker from the same commit. Nothing in the project pins or vendors an older specification; the editor's vendored checker, six commits behind, was the last such pin. The research site and the experiments repository remain separate and read the specification repository; it never reads them.

References

- [1] C. E. Shannon, "Prediction and Entropy of Printed English," *Bell System Technical Journal* 30(1), 1951. doi:10.1002/j.1538-7305.1951.tb01366.x
- [2] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms," *ACM Transactions on Computational Logic* 1(1), 2000. doi:10.1145/343369.343384
- [3] A. Blass and Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms," *ACM Transactions on Computational Logic*, 2003. doi:10.1145/937555.937561
- [4] N. Dershowitz and Y. Gurevich, "A Natural Axiomatization of Computability and Proof of Church's Thesis," *Bulletin of Symbolic Logic*, 2008. doi:10.2178/bsl/1231081370
- [5] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 1970. doi:10.1145/362384.362685
- [6] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, 1955. doi:10.1002/j.1538-7305.1955.tb03788.x
- [7] E. F. Moore, "Gedanken-Experiments on Sequential Machines," *Automata Studies*, 1956. doi:10.1515/9781400882618-006
- [8] K. Kang et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," CMU/SEI-90-TR-021, 1990.
- [9] R. Alur et al., "Syntax-Guided Synthesis," FMCAD 2013. doi:10.1109/FMCAD.2013.6679385
- [10] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 1987. doi:10.1016/0167-6423(87)90035-9
- [11] R. Tate, M. Stepp, Z. Tatlock and S. Lerner, "Equality Saturation," POPL 2009. doi:10.1145/1480881.1480915

- [12] M. Willsey et al., "egg: Fast and Extensible Equality Saturation," POPL 2021. doi:10.1145/3434304
- [13] B. Moseley and P. Marks, "Out of the Tar Pit," 2006.
- [14] M. Fowler, "Event Sourcing," 2005. <https://martinfowler.com/eaDev/EventSourcing.html>
- [15] P. Alvaro et al., "Dedalus: Datalog in Time and Space," LNCS 2011. doi:10.1007/978-3-642-24206-9_16
- [16] A. M. Turing, "On Computable Numbers," *Proceedings of the London Mathematical Society*, 1937. doi:10.1112/plms/s2-42.1.230
- [17] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Mathematics*, 1936. doi:10.2307/2371045
- [18] P. Wadler, "Propositions as Types," *Communications of the ACM*, 2015. doi:10.1145/2699407
- [19] J. M. Hellerstein and P. Alvaro, "Keeping CALM," *Communications of the ACM*, 2020. doi:10.1145/3369736
- [20] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," IFIP 1974, pp. 471 to 475.
- [21] H. Husain et al., "CodeSearchNet Challenge," 2019. arXiv:1909.09436
- [22] D. Kim, J. Nam, J. Song and S. Kim, "Automatic Patch Generation Learned from Human-Written Patches" (PAR), ICSE 2013. doi:10.1109/ICSE.2013.6606626
- [23] K. Liu et al., "TBar: Revisiting Template-based Automated Program Repair," ISSTA 2019. arXiv:1903.08409
- [24] K. Liu et al., "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations," SANER 2019. arXiv:1812.07270
- [25] A. Koyuncu et al., "FixMiner: Mining Relevant Fix Patterns for Automated Program Repair," 2018. arXiv:1810.01791
- [26] Y. Padiou et al., "Documenting and Automating Collateral Evolutions in Linux Device Drivers" (Coccinelle), EuroSys 2008. doi:10.1145/1352592.1352618
- [27] A. Solar-Lezama et al., "Combinatorial Sketching for Finite Programs," ASPLOS 2006. doi:10.1145/1168857.1168907
- [28] L. Chen, M. Zaharia and J. Zou, "FrugalGPT," 2023. arXiv:2305.05176
- [29] S. Haber and W. S. Stornetta, "How to Time-Stamp a Digital Document," *Journal of Cryptology*, 1991. doi:10.1007/BF00196791

- [30] S. A. Crosby and D. S. Wallach, "Efficient Data Structures for Tamper-Evident Logging," USENIX Security 2009.
- [31] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," TACAS 2008. doi:10.1007/978-3-540-78800-3_24
- [32] Kani Rust Verifier. <https://github.com/model-checking/kani>
- [33] C. V. Lopes et al., "DéjàVu: A Map of Code Duplicates on GitHub," OOPSLA 2017. doi:10.1145/3133908
- [34] M. Gabel and Z. Su, "A Study of the Uniqueness of Source Code," FSE 2010. doi:10.1145/1882291.1882315
- [35] A. Hindle et al., "On the Naturalness of Software," ICSE 2012. doi:10.1109/ICSE.2012.6227135
- [36] D. Kocetkov et al., "The Stack: 3 TB of Permissively Licensed Source Code," 2022. arXiv:2211.15533; The Stack v2 dataset card, <https://huggingface.co/datasets/bigcode/the-stack-v2>
- [37] R. Just, D. Jalali and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," ISSTA 2014. doi:10.1145/2610384.2628055
- [38] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" 2023. arXiv:2310.06770
- [39] R. Aleithan et al., "SWE-Bench+: Enhanced Coding Benchmark for LLMs," 2024. arXiv:2410.06992
- [40] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal* 27, 1948. doi:10.1002/j.1538-7305.1948.tb01338.x
- [41] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Problemy Peredachi Informatsii* 1(1), 1965 (in Russian); English translation, *International Journal of Computer Mathematics* 2, 1968. doi:10.1080/00207166808803030
- [42] R. J. Solomonoff, "A formal theory of inductive inference. Part I," *Information and Control* 7(1), 1964. doi:10.1016/S0019-9958(64)90223-2
- [43] G. J. Chaitin, "A Theory of Program Size Formally Identical to Information Theory," *Journal of the ACM* 22(3), 1975. doi:10.1145/321892.321894
- [44] J. Rissanen, "Modeling by shortest data description," *Automatica* 14(5), 1978. doi:10.1016/0005-1098(78)90005-5
- [45] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli and P. Devanbu, "On the 'naturalness' of buggy code," ICSE 2016. doi:10.1145/2884781.2884848

- [46] Z. Tu, Z. Su and P. Devanbu, "On the localness of software," FSE 2014. doi:10.1145/2635868.2635875
- [47] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys* 51(4), 2018. doi:10.1145/3212695
- [48] 陈浙哲, 刘忠鑫, 鄢萌, 徐洲, 夏鑫, 雷晏 (Z. Chen, Z. Liu, M. Yan, Z. Xu, X. Xia and Y. Lei), "代码自然性及其应用研究进展" (Research progress of code naturalness and its application), *软件学报 (Journal of Software)* 33(8), 3015 to 3034, 2022. doi:10.13328/j.cnki.jos.006355
- [49] G. Sridhara, V. S. Sinha and S. Mani, "Naturalness of Natural Language Artifacts in Software," ISEC 2015 (India Software Engineering Conference). doi:10.1145/2723742.2723758
- [50] P. Devanbu, "On the naturalness of software," keynote, ISEC 2013 (India Software Engineering Conference). doi:10.1145/2442754.2442763
- [51] A. Z. H. Yang, S. Kolak, V. J. Hellendoorn, R. Martins et al., "Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models," ICSE 2025. doi:10.1109/ICSE55347.2025.00089
- [52] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM* 18(8), 1975. doi:10.1145/360933.360975
- [53] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM* 14(4), 1971. doi:10.1145/362575.362577
- [54] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM* 12(10), 1969. doi:10.1145/363235.363259
- [55] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996. doi:10.1017/CBO9780511624162
- [56] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2010. doi:10.1017/CBO9781139195881
- [57] R.-J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*, Springer, 1998. doi:10.1007/978-1-4612-1674-2
- [58] F. L. Bauer and H. Wössner, *Algorithmic Language and Program Development*, Springer, 1982. doi:10.1007/978-3-642-61807-9
- [59] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," POPL 2011. doi:10.1145/1926385.1926423
- [60] S. Gulwani, O. Polozov and R. Singh, "Program Synthesis," *Foundations and Trends in Programming Languages* 4(1-2), 2017. doi:10.1561/25000000010

- [61] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer* 20(4), 1987. doi:10.1109/MC.1987.1663532
- [62] P. Naur, "Programming as theory building," *Microprocessing and Microprogramming* 15(5), 1985. doi:10.1016/0165-6074(85)90032-8
- [63] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE* 68(9), 1980. doi:10.1109/PROC.1980.11805
- [64] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology* 6(1), 1997. doi:10.1145/237432.237434
- [65] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," *IBM Journal of Research and Development* 5(3), 1961. doi:10.1147/rd.53.0183
- [66] C. H. Bennett, "Logical Reversibility of Computation," *IBM Journal of Research and Development* 17(6), 1973. doi:10.1147/rd.176.0525
- [67] M. Franz and T. Kistler, "Slim binaries," *Communications of the ACM* 40(12), 1997. doi:10.1145/265563.265576
- [68] C. W. Fraser, "Automatic inference of models for statistical code compression," PLDI 1999. doi:10.1145/301618.301672
- [69] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," POPL 1977. doi:10.1145/512950.512973
- [70] Y. Futamura, "Partial evaluation of computation process: an approach to a compiler-compiler," *Systems, Computers, Controls* 2(5), 1971; reprinted in *Higher-Order and Symbolic Computation* 12(4), 1999. doi:10.1023/A:1010095604496
- [71] A. P. Ershov, "Mixed computation: potential applications and problems for study," *Theoretical Computer Science* 18(1), 1982. doi:10.1016/0304-3975(82)90111-6
- [72] W. A. Howard, "The formulae-as-types notion of construction," in J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.
- [73] 土居真之, 肥後芳樹, 有馬諒, 下仲健斗, 楠本真二 (M. Doi, Y. Higo, R. Arima, K. Shimonaka and S. Kusumoto), "ソースコードの「自然さ」を利用した自動生成ファイルの特定" (Identifying auto-generated files using the naturalness of source code), 電子情報通信学会技術研究報告 (IEICE Technical Report) 117(380), 19 to 24, January 2018. <https://sdl.ist.osaka-u.ac.jp/pman/pman3.cgi?D=517>

- [74] M. Doi, Y. Higo, R. Arima, K. Shimonaka and S. Kusumoto, "On the naturalness of auto-generated code: can we identify auto-generated code automatically?" ICPC 2018. doi:10.1145/3196321.3196356
- [75] F. V. Arrebola, "Modelo de sugestão de código-fonte baseado em modelos de linguagem organizados em um contexto hierárquico" (A source code suggestion model based on language models organized in a hierarchical context), doctoral thesis in electrical engineering, Fundação Educacional Inaciana Padre Sabóia de Medeiros (FEI), Brazil, 2018. Advisor R. A. C. Bianchi.
- [76] G. Febres, K. Jaffé and C. Gershenson, "Complexity measurement of natural and artificial languages," *Complexity* 20(6), 2015. doi:10.1002/cplx.21529
- [77] M. Rahman, D. Palani and P. C. Rigby, "Natural Software Revisited," ICSE 2019. doi:10.1109/ICSE.2019.00022
- [78] A. Torres, S. Baltes, C. Treude and M. Wagner, "Applying Information Theory to Software Evolution," NLBSE 2023. doi:10.1109/NLBSE59153.2023.00017
- [79] A. Blass, N. Dershowitz and Y. Gurevich, "When are two algorithms the same?," *Bulletin of Symbolic Logic* 15(2), 2009. arXiv:0811.0811. <https://arxiv.org/abs/0811.0811>
- [80] Y. N. Moschovakis, "What is an algorithm?," in *Mathematics Unlimited: 2001 and Beyond*, Springer, 2001. <https://www.math.ucla.edu/~ynm/papers/eng.pdf>
- [81] M. Y. Vardi, "What is an algorithm?," *Communications of the ACM* 55(3), 2012. doi:10.1145/2093548.2093549
- [82] S. Laddad, A. Cheung, J. M. Hellerstein and M. Milano, "Flo: a Semantic Foundation for Progressive Stream Processing," POPL 2025. arXiv:2411.08274. <https://arxiv.org/abs/2411.08274>
- [83] The Elm Architecture, Elm guide. <https://guide.elm-lang.org/architecture/>
- [84] Temporal, "Workflows" (event history), documentation. <https://docs.temporal.io/workflows>
- [85] A. Pnueli, M. Siegel and E. Singerman, "Translation validation," TACAS 1998. doi:10.1007/BFb0054170
- [86] X. Yang, Y. Chen, E. Eide and J. Regehr, "Finding and understanding bugs in C compilers," PLDI 2011. doi:10.1145/1993498.1993532
- [87] C. Nandi, M. Willsey, A. Zhu et al., "Rewrite Rule Inference Using Equality Saturation," OOPSLA 2021. arXiv:2108.10436. <https://arxiv.org/abs/2108.10436>
- [88] C. Le Goues, T. Nguyen, S. Forrest and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE TSE* 38(1), 2012. doi:10.1109/TSE.2011.104

- [89] F. Long and M. Rinard, "Automatic patch generation by learning correct code," POPL 2016. doi:10.1145/2837614.2837617
- [90] J. Bader, A. Scott, M. Pradel and S. Chandra, "Getafix: Learning to Fix Bugs Automatically," OOPSLA 2019. arXiv:1902.06111. <https://arxiv.org/abs/1902.06111>
- [91] K. Etemadi, N. Harrand, S. Larsen et al., "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations," *IEEE TDSC*, 2022. arXiv:2103.12033. <https://arxiv.org/abs/2103.12033>
- [92] Z. Qi, F. Long, S. Achour and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," ISSTA 2015. doi:10.1145/2771783.2771791
- [93] E. K. Smith, E. T. Barr, C. Le Goues and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," ESEC/FSE 2015. doi:10.1145/2786805.2786825
- [94] X.-B. D. Le, F. Thung, D. Lo and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering* 23, 2018. doi:10.1007/s10664-017-9577-2
- [95] Y. Xiong, X. Liu, M. Zeng, L. Zhang and G. Huang, "Identifying patch correctness in test-based program repair," ICSE 2018. doi:10.1145/3180155.3180182
- [96] T. Durieux, F. Madeiral, M. Martinez and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," ESEC/FSE 2019. arXiv:1905.11973. <https://arxiv.org/abs/1905.11973>
- [97] K. Liu, A. Koyuncu, T. F. Bissyandé et al., "You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems," ICST 2019. arXiv:1812.07283. <https://arxiv.org/abs/1812.07283>
- [98] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys* 51(1), 2018. doi:10.1145/3105906
- [99] C. S. Xia, Y. Deng, S. Dunn and L. Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," 2024. arXiv:2407.01489. <https://arxiv.org/abs/2407.01489>
- [100] S. Kapoor, B. Stroebl, Z. S. Siegel, N. Nadgir and A. Narayanan, "AI Agents That Matter," 2024. arXiv:2407.01502. <https://arxiv.org/abs/2407.01502>
- [101] M. H. Erol, B. El, M. Suzgun, M. Yuksekgonul and J. Zou, "Cost-of-Pass: An Economic Framework for Evaluating Language Models," 2025. arXiv:2504.13359. <https://arxiv.org/abs/2504.13359>
- [102] D. Zan, Z. Huang, W. Liu et al., "Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving," 2025. arXiv:2504.02605. <https://arxiv.org/abs/2504.02605>

- [103] R. Widyasari, S. Q. Sim, C. Lok et al., "BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies," ESEC/FSE 2020. doi:10.1145/3368089.3417943
- [104] M. Allamanis, "The Adverse Effects of Code Duplication in Machine Learning Models of Code," Onward! 2019. arXiv:1812.06469. <https://arxiv.org/abs/1812.06469>
- [105] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM TOMPECS* 3(2), 2018. doi:10.1145/3177754
- [106] The Rust project, rustfix and cargo fix. <https://github.com/rust-lang/rustfix>
- [107] R.-M. Karampatsis and C. Sutton, "How Often Do Single-Statement Bugs Occur? The ManySSuBs4J Dataset," MSR 2020. doi:10.1145/3379597.3387491
- [108] C. Richter and H. Wehrheim, "TSSB-3M: Mining single statement bugs at massive scale," MSR 2022. doi:10.1145/3524842.3528505
- [109] H. Coles, T. Laurent, C. Henard, M. Papadakis and A. Ventresque, "PIT: a practical mutation testing tool for Java," ISSTA 2016. doi:10.1145/2931037.2948707
- [110] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" FSE 2014. doi:10.1145/2635868.2635929
- [111] I. J. Good, "Speculations Concerning the First Ultrainelligent Machine," *Advances in Computers* 6, 1965. doi:10.1016/S0065-2458(08)60418-0
- [112] J. Schmidhuber, "Gödel Machines: Fully Self-referential Optimal Universal Self-improvers," in *Artificial General Intelligence*, Springer, 2007. doi:10.1007/978-3-540-68677-4_7
- [113] E. Yudkowsky, "Intelligence Explosion Microeconomics," Technical report 2013-1, Machine Intelligence Research Institute, 2013. <https://intelligence.org/files/IEM.pdf>
- [114] M. H. Löb, "Solution of a Problem of Leon Henkin," *Journal of Symbolic Logic* 20(2), 1955. doi:10.2307/2266895
- [115] E. Yudkowsky and M. Herreshoff, "Tiling Agents for Self-Modifying AI, and the Löbian Obstacle," Machine Intelligence Research Institute, 2013. <https://intelligence.org/files/TilingAgentsDraft.pdf>
- [116] E. Zelikman, E. Lorch, L. Mackey and A. T. Kalai, "Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation," 2023. doi:10.48550/arXiv.2310.02304
- [117] J. Zhang, S. Hu, C. Lu, R. Lange and J. Clune, "Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents," 2025. doi:10.48550/arXiv.2505.22954
- [118] A. Novikov et al., "AlphaEvolve: A coding agent for scientific and algorithmic discovery," 2025. doi:10.48550/arXiv.2506.13131

- [119] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan and A. Anandkumar, "Voyager: An Open-Ended Embodied Agent with Large Language Models," 2023. doi:10.48550/arXiv.2305.16291
- [120] Y. Futamura, "Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler," *Higher-Order and Symbolic Computation* 12(4), 1999 (first published 1971). doi:10.1023/A:1010095604496
- [121] V. F. Turchin, "The Concept of a Supercompiler," *ACM Transactions on Programming Languages and Systems* 8(3), 1986. doi:10.1145/5956.5957
- [122] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce and A. Schmitt, "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem," *ACM Transactions on Programming Languages and Systems* 29(3), 2007. doi:10.1145/1232420.1232424
- [123] M. Hofmann, B. Pierce and D. Wagner, "Symmetric Lenses," *POPL*, 2011. doi:10.1145/1926385.1926428
- [124] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr and J. F. Terwilliger, "Bidirectional Transformations: A Cross-Discipline Perspective," *ICMT*, LNCS 5563, 2009. doi:10.1007/978-3-642-02408-5_19
- [125] E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," *BIT* 8(3), 1968. doi:10.1007/BF01933419
- [126] A. Hall and R. Chapman, "Correctness by Construction: Developing a Commercial Secure System," *IEEE Software* 19(1), 2002. doi:10.1109/52.976937
- [127] Y. Minsky, "OCaml for the Masses," *Communications of the ACM* 54(11), 2011. doi:10.1145/2018396.2018413
- [128] D. A. Turner, "Total Functional Programming," *Journal of Universal Computer Science* 10(7), 2004. doi:10.3217/jucs-010-07-0751
- [129] R. Di Cosmo and S. Zacchiroli, "Software Heritage: Why and How to Preserve Software Source Code," iPRES 2017, Kyoto. <https://hal.science/hal-01590958>
- [130] Software Heritage, archive counters, read 13 October 2026: 30,253,266,524 file contents and 448,311,919 origins. <https://archive.softwareheritage.org/api/1/stat/counters/>
- [131] Rosetta Code, main page, read 13 October 2026: 1,360 tasks, 421 draft tasks, 1,009 known languages. https://rosettacode.org/wiki/Rosetta_Code
- [132] C. Casalnuovo, K. Lee, H. Wang, P. Devanbu and E. Morgan, "Do Programmers Prefer Predictable Expressions in Code?" *Cognitive Science* 44(12), 2020; preprint arXiv:1910.03704. <https://arxiv.org/abs/1910.03704>

- [133] N. Peitek, S. Apel, C. Parnin, A. Brechmann and J. Siegmund, "Program Comprehension and Code Complexity Metrics: An fMRI Study," ICSE 2021. <https://www.se.cs.uni-saarland.de/publications/docs/PAP+21.pdf>
- [134] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk and R. Oliveto, "Automatically Assessing Code Understandability," *IEEE Transactions on Software Engineering* 47(3), 2021. <https://www.cs.wm.edu/~denys/pubs/TSE'19-Understandability.pdf>
- [135] N. Peitek, J. Siegmund, C. Parnin, S. Apel and A. Brechmann, "Beyond Gaze: Preliminary Analysis of Pupil Dilation and Blink Rates in an fMRI Study of Program Comprehension," and the companion layout study, 2017. <https://www.se.cs.uni-saarland.de/publications/docs/SPP+17.pdf>
- [136] "The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget," arXiv:2508.13666, August 2025; ICSE 2026. <https://arxiv.org/abs/2508.13666>
- [137] D. G. Feitelson, A. Mizrahi, N. Noy, A. Ben Shabat, O. Eliyahu and R. Sheffer, "How Developers Choose Names," *IEEE Transactions on Software Engineering* 48(1), 2022; preprint arXiv:2103.07487. <https://arxiv.org/abs/2103.07487>
- [138] F. Wen, C. Nagy, G. Bavota and M. Lanza, "A Large-Scale Empirical Study on Code-Comment Inconsistencies," ICPC 2019. <https://www.inf.usi.ch/faculty/lanza/PUBS/P/Wen2019a.pdf>
- [139] "When Names Disappear: Revealing What LLMs Actually Understand About Code," arXiv:2510.03178, October 2025. <https://arxiv.org/abs/2510.03178>
- [140] A. A. Ivanova, S. Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O'Reilly, M. U. Bers and E. Fedorenko, "Comprehension of Computer Code Relies Primarily on Domain-General Executive Brain Regions," *eLife* 9, 2020. doi:10.7554/eLife.58906
- [141] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society* 74, 1953. doi:10.1090/S0002-9947-1953-0053041-6
- [142] O. Danvy and K. Millikin, "Refunctionalization at Work," BRICS RS-08-4, 2008; *Science of Computer Programming* 74(8), 2009. <https://www.brics.dk/RS/08/4/>
- [143] M. S. Ager, D. Biernacki, O. Danvy and J. Midtgaard, "A Functional Correspondence between Evaluators and Abstract Machines," PPDP 2003. doi:10.1145/888251.888254
- [144] J. J. M. M. Rutten, "Universal Coalgebra: A Theory of Systems," *Theoretical Computer Science* 249(1), 2000. doi:10.1016/S0304-3975(00)00056-6
- [145] J. W. de Bakker and J. I. Zucker, "Processes and the Denotational Semantics of Concurrency," *Information and Control* 54(1-2), 1982. doi:10.1016/S0019-9958(82)91013-8

- [146] C. P. Pfleeger, "State Reduction in Incompletely Specified Finite-State Machines," *IEEE Transactions on Computers* C-22(12), 1973. doi:10.1109/T-C.1973.223591
- [147] A. Nerode, "Linear Automaton Transformations," *Proceedings of the American Mathematical Society* 9(4), 1958. doi:10.2307/2033204
- [148] J. Hopcroft, "An $n \log n$ Algorithm for Minimizing States in a Finite Automaton," in *Theory of Machines and Computations*, Academic Press, 1971. doi:10.1016/B978-0-12-417750-5.50022-1
- [149] L. D'Antoni and M. Veanes, "Minimization of Symbolic Automata," POPL 2014. doi:10.1145/2535838.2535849
- [150] R. Reis, N. Moreira and M. Almeida, "On the Representation of Finite Automata," DCFS 2005. <https://www.dcc.fc.up.pt/~nam/resources/publica/dcfsrcma05.pdf>
- [151] M. Bojańczyk, B. Klin and S. Lasota, "Automata with Group Actions," LICS 2011. doi:10.1109/LICS.2011.48
- [152] J. Lee et al., "A Linear Energy Model for Cortex-M0 Class Cores from Six Hardware Counters," arXiv:2104.01055, 2021. <https://arxiv.org/abs/2104.01055>
- [153] N. van Kempen, H. Kwon, D. T. Nguyen and E. D. Berger, "It's Not Easy Being Green: On the Energy Efficiency of Programming Languages," arXiv:2410.05460v4, 2025. <https://arxiv.org/abs/2410.05460>
- [154] Dauner et al., "Stale Reads: Sampling Hazards of On-Chip Energy Counters," HotCarbon 2026. <https://hotcarbon.org/assets/2026/paper-46.pdf>
- [155] Z. Yang et al., "Part-Time Power Measurements: nvidia-smi's Lack of Attention," arXiv:2312.02741v3, 2024. <https://arxiv.org/abs/2312.02741>
- [156] D. Perez and B. Livshits, "Broken Metre: Attacking Resource Metering in EVM," NDSS 2020; arXiv:1909.07220. <https://arxiv.org/abs/1909.07220>
- [157] S. Smolka, N. Foster, J. Hsu, T. Kappé, D. Kozen and A. Silva, "Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time," POPL 2020. doi:10.1145/3371129
- [158] Y. Sun, Y. Zhang and Z. Ni, "E-Graph Extraction with DAG Cost Is NP-Hard and Inapproximable," arXiv:2408.17042, 2024. <https://arxiv.org/abs/2408.17042>
- [159] Unison Computing, "The Big Idea," Unison language documentation; and "A New Update Process," 20 October 2023. <https://www.unison-lang.org/docs/the-big-idea/>
- [160] Dhall language standard, "Binary Semantics and Semantic Integrity Checks," binary.md. <https://github.com/dhall-lang/dhall-lang/blob/master/standard/binary.md>

- [161] Tweag, "Towards a Content-Addressed Model for Nix," 10 September 2020, and "Self-References in a Content-Addressed Nix," 18 November 2020.
<https://www.tweag.io/blog/2020-09-10-nix-cas/>
- [162] "SpecRL: Measuring Specification Completeness for Verified Program Synthesis," arXiv:2604.05820, 2026. <https://arxiv.org/abs/2604.05820>
- [163] J. Koppel, Z. Guo, E. de Vries and A. Solar-Lezama, "Searching Entangled Program Spaces," ICFP 2022. doi:10.1145/3547622
- [164] M. Erol et al., "Cost-of-Pass: An Economic Framework for Evaluating Language Models," arXiv:2504.13359, 2025. <https://arxiv.org/abs/2504.13359>
- [165] F. Cassano, J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda and A. Guha, "Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs," OOPSLA 2024. doi:10.1145/3689735
- [166] agentlanguages.dev, catalogue of agent-oriented programming languages, read 19 October 2026 (42 entries). <https://agentlanguages.dev>
- [167] WebAssembly Community Group, "SpecTec Has Been Adopted," 27 March 2025; D. Youn et al., "Bringing the WebAssembly Standard up to Speed with SpecTec," PLDI 2024.
<https://webassembly.org/news/2025-03-27-spectec/>
- [168] D. Monniaux and S. Boulmé, "The Trusted Computing Base of the CompCert Verified Compiler," ESOP 2022. <https://arxiv.org/abs/2201.10280>
- [169] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems* 32(1), 2014; cost figures from NICTA technical report 4180.
https://www.trustworthy.systems/publications/nicta_full_text/4180.pdf
- [170] S. Lu, S. Park, E. Seo and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," ASPLOS 2008. doi:10.1145/1346281.1346323
- [171] R. Haas, R. Niedermayr, T. Röhm and S. Apel, "Is Static Analysis Able to Identify Unnecessary Source Code?" *ACM Transactions on Software Engineering and Methodology* 29(1), 2020. doi:10.1145/3368267
- [172] N. Nagappan, B. Murphy and V. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study," ICSE 2008. doi:10.1145/1368088.1368160
- [173] "An Empirical Study of Go's Language Proposal Process," arXiv:2510.06984, October 2025 (1,095 proposals; 448 declined, 337 accepted). <https://arxiv.org/abs/2510.06984>
- [174] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H. Lee and Y.-N. Chen, "Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models," EMNLP Industry 2024; arXiv:2408.02442. <https://arxiv.org/abs/2408.02442>

- [175] B. Stroustrup, "Thriving in a Crowded and Changing World: C++ 2006–2020," *HOPL IV, Proceedings of the ACM on Programming Languages* 4(HOPL), 2020. doi:10.1145/3386320
- [176] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy and R. Treinen, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions," *ASE* 2006, pp. 199–208. doi:10.1109/ASE.2006.49
- [177] P. Abate, R. Di Cosmo, G. Gousios and S. Zacchiroli, "Dependency Solving Is Still Hard, but We Are Getting Better at It," *SANER* 2020, pp. 547–551. doi:10.1109/SANER48275.2020.9054837
- [178] R. Cox, "Version SAT," research!rsc, 13 December 2016. <https://research.swtch.com/version-sat>
- [179] S. Raemaekers, A. van Deursen and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software* 129:140–158, 2017. doi:10.1016/j.jss.2016.04.008
- [180] L. Ochoa, T. Degueule, J.-R. Falleri and J. Vinju, "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central," *Empirical Software Engineering* 27:61, 2022. doi:10.1007/s10664-021-10052-y
- [181] T. Winters, T. Manshreck and H. Wright (eds.), *Software Engineering at Google*, chapter 21, O'Reilly, 2020. <https://abseil.io/resources/swe-book/html/ch21.html>
- [182] J. Wetter and N. Ringland, "Understanding the Impact of Apache Log4j Vulnerability," Google Open Source Insights, 17 December 2021. <https://blog.deps.dev/log4j/>
- [183] E. Dolstra, *The Purely Functional Software Deployment Model*, PhD thesis, Utrecht University, 2006. <https://edolstra.github.io/pubs/phd-thesis.pdf>
- [184] R. Di Cosmo, M. Gruenpeter and S. Zacchiroli, "Referencing Source Code Artifacts: a Separate Concern in Software Citation," *Computing in Science & Engineering* 22(2):33–43, 2020. doi:10.1109/MCSE.2019.2963148
- [185] R. Hickey, "Spec-ulation," keynote, Clojure/conj, December 2016. Transcript: https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/Spec_ulation.md
- [186] B. Godlin and O. Strichman, "Regression verification," *DAC* 2009, pp. 466–471. doi:10.1145/1629911.1630034
- [187] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman and G. Janssen, "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations," *ICCD* 2006. <https://research.ibm.com/publications/scalable-sequential-equivalence-checking-across-arbitrary-design-transformations>
- [188] C. Lamb and S. Zacchiroli, "Reproducible Builds: Increasing the Integrity of Software Supply Chains," *IEEE Software* 39(2):62–70, 2022. doi:10.1109/MS.2021.3073045

[189] Open Interface Engineering, "Languages for or by machine authors, January 2024 to October 2026," working note 5 of the survey of 24 October 2026, `rce/surveys/2026-10-24/notes/languages_for_machine_authors.md`, openIE-dev/openie-research-experiments. <https://github.com/openIE-dev/openie-research-experiments>

[190] Open Interface Engineering, "New general-purpose and systems languages, January 2024 to October 2026," working note 6 of the survey of 24 October 2026, `rce/surveys/2026-10-24/notes/new_general_purpose_languages.md`, openIE-dev/openie-research-experiments. <https://github.com/openIE-dev/openie-research-experiments>

[191] W. van Oortmerssen, Goose: a memory-safe systems language with no heap allocator, garbage collector, or lifetime annotations, repository, 2026. <https://github.com/aardappel/goose>

[192] dawnop, Dawn: a small functional language with effects in the type signature and content-addressed packages, repository, 2026. <https://github.com/dawnop/dawn-lang>

[193] Open Interface Engineering, "Batteries included or specification and free: what the record says, and one Form on every compute fabric," synthesis of the survey of 24 October 2026, with working notes 7 to 18 and the proposal `spec/proposals/one-library-every-fabric.md` of openIE-dev/rce. <https://github.com/openIE-dev/openie-research-experiments/blob/main/rce/surveys/2026-10-24/batteries-specification-and-fabrics.md>

[194] C. Heimes and B. Cannon, "PEP 594: Removing dead batteries from the standard library," Python Enhancement Proposals, 2019. <https://peps.python.org/pep-0594/>

[195] Rust project, "RFC 1242: Policy for rust-lang crates," 2015. <https://rust-lang.github.io/rfcs/1242-rust-lang-crates.html>

[196] A. Decan, T. Mens and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," Empirical Software Engineering 24, 2019. <https://doi.org/10.1007/s10664-017-9589-y>

[197] N. Seriot, "Parsing JSON is a Minefield," 2016. https://seriot.ch/projects/parsing_json.html

[198] J. H. Davis, P. Sivaraman, J. Kitson, K. Parasyris, H. Menon, I. Minn, G. Georgakoudis and A. Bhatele, "Taking GPU Programming Models to Task for Performance Portability," Proc. ICS 2025. <https://doi.org/10.1145/3721145.3730423>

[199] C. Weckert, L. Solis-Vasquez, J. Oppermann, A. Koch and O. Sinnen, "Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs," SC '23 Workshops, 2023. <https://doi.org/10.1145/3624062.3624542>

[200] T. Bourgeat, C. Pit-Claudel, A. Chlipala and Arvind, "The essence of Bluespec: a core language for rule-based hardware design," Proc. PLDI 2020. <https://doi.org/10.1145/3385412.3385965>

- [201] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. F. Bastien, "Bringing the Web up to Speed with WebAssembly," Proc. PLDI 2017. <https://doi.org/10.1145/3062341.3062363>
- [202] A. Arteaga, O. Fuhrer and T. Hoefler, "Designing Bit-Reproducible Portable High-Performance Applications," Proc. IPDPS 2014. <https://doi.org/10.1109/IPDPS.2014.127>
- [203] D. Lustig, S. Sahasrabuddhe and O. Giroux, "A Formal Analysis of the NVIDIA PTX Memory Consistency Model," Proc. ASPLOS 2019. <https://doi.org/10.1145/3297858.3304043>
- [204] StarPU project, StarPU Handbook: performance models and simulation with SimGrid, Inria. <https://files.inria.fr/starpu/doc/html/>
- [205] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath and M. Jadliwala, "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," Proc. USENIX Security 2025.