

The Pursuit of AGI Resolves the Three-Body Problem in Computing

David Charlot, Open Interface Engineering

Version 1.1, 11 October 2026. HTML: <https://research.openie.dev/papers/three-body/>

Contents

Abstract	1
1. Thesis	1
2. The three-body problem, and what the analogy carries	2
3. The three bodies of computing and their interfaces	3
4. The interface literature	4
5. Definitions and propositions	5
6. The stabilized configuration	7
7. The hardware side	9
8. Protocols	9
9. Falsifiers	10
10. Scope and method	10
11. Conclusion	10
References	10

Version 1.0, 11 October 2026. LaTeX PDF: </pdfs/latex/three-body.pdf> (source)..

Abstract

Large engineering failures often start where one group hands work to another and an assumption is never written down. This paper defines those handoffs, states when checking them makes a system stable, and gives tests that could prove it wrong.

In plain words

1. Most big computer failures happen at the handoff between teams or parts, not inside them.
2. The cause is usually a guess nobody wrote down, like which units a number is in.
3. If every such guess is written down and checked by a machine, a fix made in one place stays fixed and does not break anything above it.

1. Thesis

In Liu Cixin's novel *The Three-Body Problem* [1], a civilization lives on a planet that orbits three suns. Its seasons have no rhythm, because no one can predict where three bodies under mutual gravity will be. The title of this paper borrows that picture for computing, and then turns it around: the instability is real, but it has a cause that can be located, and the cause can be removed.

Computing has three bodies. Theorists state what a system should do: the mathematics, the specification, the algorithm, the physical model. Makers build it: the chip designers, the compiler and kernel writers, the library authors, the integrators. Users run it: the operators, the downstream programs, the people and the machines that depend on its output. Each body is competent and internally coherent. A theorem holds. A chip meets its datasheet. A user follows the documentation.

The trouble lives between them. Every handoff from one body to another carries assumptions: about units, voltages, ranges, timing, memory, encodings, versions, operating envelopes and intent. When each body assumes that the handoff is not there, or that it is somebody else’s job, the assumption is never written down, and an assumption that is never written down is never checked.

The core claim. The instability of computing systems is located at their unexamined interfaces, not in the bodies. Pursuing artificial general intelligence (AGI) seriously, as a system that must act correctly across every body and every level of the stack, forces every interface assumption to be made explicit and checked. A stack whose interface assumptions are all explicit and checked is stable in a precise sense: a correction made at one level stays correct at that level and carries into every level above it. That is the resolution the title names.

The claim is engineering and theory. It is stated with definitions in [Section 5](#), it rests on the interface literature reviewed in [Section 4](#), its evidence comes from independently tested verified systems ([Section 6](#)), and it carries protocols and falsifiers that can defeat it ([Section 8](#) and [Section 9](#)).

2. The three-body problem, and what the analogy carries

2.1 What the physics says

Newton solved the motion of two bodies under mutual gravity in closed form: each moves on a conic section about their common centre of mass. Add a third body and the problem changes character. Poincaré’s memoir for the prize of King Oscar II, published in *Acta Mathematica* in 1890, showed that, beyond the classical integrals, the three-body problem has no further integrals of the kind that would reduce it to quadratures, and in correcting an error in his first submission he found the tangle of trajectories near a periodic orbit that is now recognized as the first appearance of deterministic chaos [2, 3]. Barrow-Green’s history traces the episode from the prize to the corrected memoir [3].

“No general closed form” needs care, because it is often overstated. Sundman showed, in work published in 1912 and 1913, that the three-body problem does have a convergent series solution for all time, provided the total angular momentum is not zero [4], and Wang extended a global series solution to the n-body problem in 1991 [5]. The series converge so slowly that they are useless for prediction. The practical content of Poincaré’s result survives: there is no simple formula, and nearby initial conditions can separate exponentially, so long-range prediction of a generic configuration is not available. Lorenz found the same sensitivity in a three-variable model of convection in 1963 [6]. Stone and Leigh showed in 2019 that the chaotic, non-hierarchical three-body problem can be described statistically, by the distribution of its outcomes, even though individual trajectories cannot be predicted [7].

2.2 The special stable configurations

Chaos is the generic case, not the only case. Lagrange found in 1772 that three bodies can keep their relative positions while they rotate together: in a straight line, or at the corners of an equilateral triangle [3]. In the restricted problem, where the third body is light, these are the five Lagrange points. The triangular points are linearly stable when the larger primary is heavy enough relative to the smaller one, a condition Routh stated in 1874 (mass ratio above about 25) [8]. Spacecraft now park near the Sun and Earth Lagrange points for that reason. Further periodic solutions exist: Moore found the figure-eight orbit numerically in 1993 [9], Chenciner and Montgomery proved its existence for equal masses in 2000 [10], and Šuvakov and Dmitrašinović found new families in 2013 [11].

The lesson for this paper is in the shape of those solutions. Each stable configuration is a special arrangement in which the forces between the bodies are balanced in a frame that moves with them. Stability is not a property of any one body. It is a property of the relations between the bodies, and of a frame in which those relations are held fixed.

2.3 Analogy and mechanism

A metaphor borrowed from physics can mislead. The table separates what this paper uses as analogy from what it claims as mechanism.

The three-body problem	In computing	Status in this paper
Three masses, each simple alone	Theorists, makers and users, each coherent alone	Analogy
Instability arises from the interactions, not from any body	Failure arises at interfaces, not inside the bodies	Mechanism, supported by the failure studies in Section 3 and the literature in Section 4

The three-body problem	In computing	Status in this paper
Sensitive dependence on initial conditions	A small unchecked mismatch at one interface grows as it propagates through levels	Mechanism, stated as Proposition 2
No useful general closed form	No single body can predict the behavior of the whole system from its own model	Analogy; computing has no equations of motion
Stable configurations exist in a co-rotating frame	A stack whose interface assumptions are all checked holds a correction where it is made	Mechanism, stated as Proposition 3
Statistical description of chaotic outcomes [7]	Failure rates can be measured at large N even where individual failures are not predicted	Method, used in the protocols of Section 8

The paper never claims that computing obeys Newtonian dynamics, that interfaces exert forces, or that a Lyapunov exponent can be computed for an organization. The physics supplies vocabulary and a structural lesson: stability comes from the relations, and it is found in special configurations, not in the generic case.

3. The three bodies of computing and their interfaces

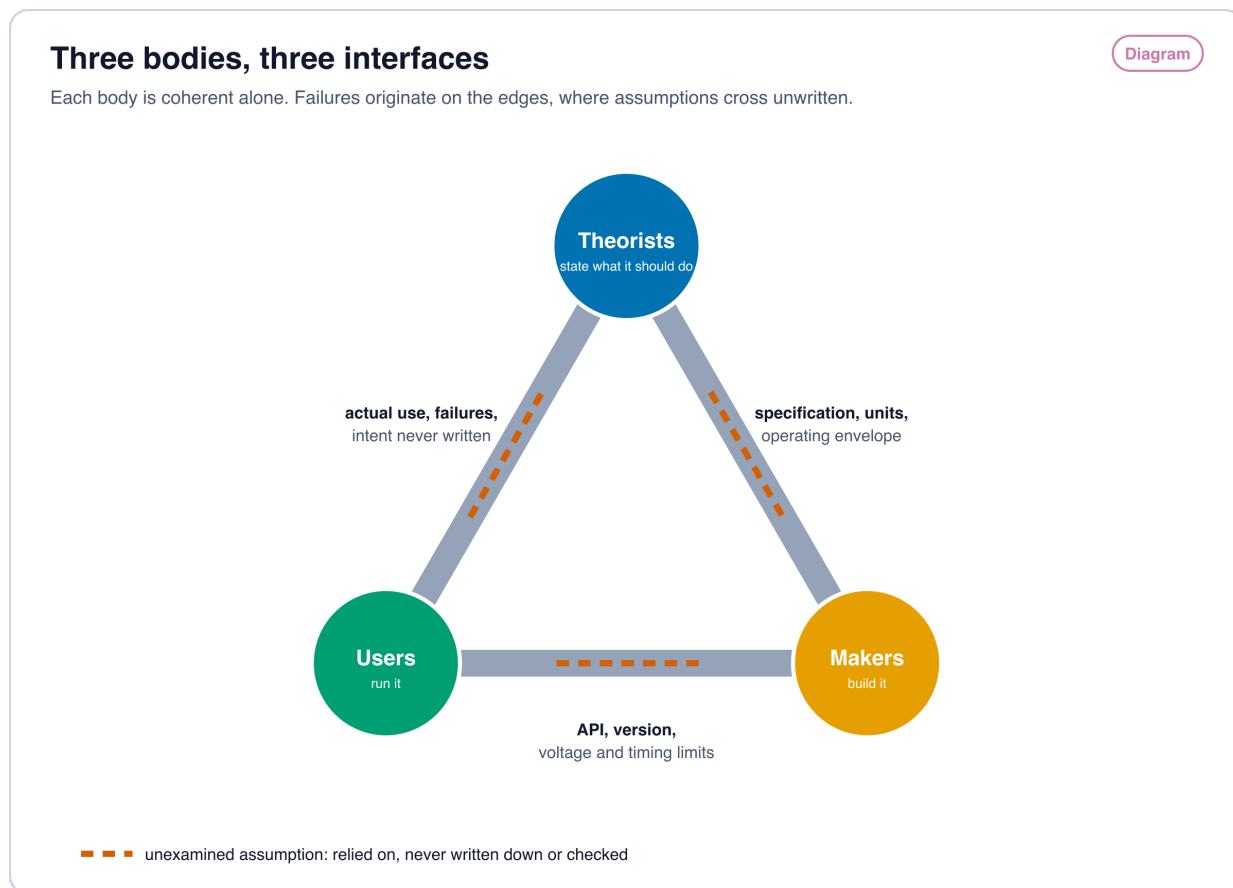


Figure 1. The three bodies and their three interfaces. Diagram. The bodies are drawn solid because each is coherent alone. The dashed bands are the unexamined assumptions where failures originate. A [PNG copy](#) is provided.

3.1 Theorist to maker

A theorist's statement assumes a domain: units, ranges, an operating envelope, an idealized machine. A maker's implementation assumes the statement is complete. When neither writes the envelope down, the implementation is correct for a world that is not the one it will run in.

The loss of Mars Climate Orbiter in 1999 is the textbook case. The Mishap Investigation Board found that ground software supplied thruster impulse data in English units (pound-force seconds) where the navigation software expected

metric units (newton seconds), and that the error accumulated over the cruise until the spacecraft entered the atmosphere far below its planned altitude [12]. Each body did its own work. The unit was an interface assumption nobody checked.

3.2 Maker to user

A maker publishes an interface and a user depends on it. The documentation states some of the contract. The rest is whatever the implementation happens to do. Hyrum’s observation states the consequence: with enough users of an API, every observable behavior of the system will be depended on by somebody, whatever the contract promises [13]. Spolsky’s essay on leaky abstractions makes the complementary point: every nontrivial abstraction exposes, to some degree, what it was built to hide [14].

The first flight of Ariane 5 in 1996 shows how a maker’s assumption crosses to a new user. The inquiry board found that the inertial reference software reused from Ariane 4 converted a 64-bit floating point value related to horizontal velocity into a 16-bit signed integer; on the faster Ariane 5 trajectory the value exceeded the range, the conversion raised an exception, both reference units shut down, and the launcher broke up about 40 seconds after the start of the flight sequence [15]. The software was correct for the envelope it was written for. The envelope was an interface assumption that traveled with the code without being restated or checked [15, 16].

The same pattern appears at the hardware boundary. An instruction set is a contract that vendors revise. RISC-V’s unprivileged specification lists several ratified bases and a long and growing list of ratified extensions [17]. Apple moved the Mac from Intel processors to Apple silicon, announced in 2020, and supplies Rosetta as a translation layer for applications built for the earlier contract; Apple’s page also states that from macOS 28 Rosetta will remain only for certain older games [18]. The translation layer is itself an interface with a published end. Software written as if the instruction contract were fixed meets a revision it did not expect, and the remedy is another interface layer.

3.3 User to theorist

The return path is the least examined. Users learn what a system actually does, where it fails and what they meant to ask for. Unless that knowledge returns to the theorists as a revised statement, the next specification repeats the old assumptions. Zave and Jackson’s account of requirements makes the gap precise: a requirement is a statement about the environment, a specification is a statement at the interface, and the domain knowledge that connects them is often unwritten [19]. Hooker’s “hardware lottery” shows the return path working in reverse: research ideas succeed or fail according to the hardware and software that happen to be available, not on their merits alone, so what makers built shapes what theorists can think [20].

3.4 Why the instability looks irreducible

Each body sees only its own coherent model. From inside any one body, the failures above look like bad luck or someone else’s mistake, because the cause lies on an edge no body owns. This is the computing counterpart of the claim that three bodies have no simple formula: no single participant’s model predicts the whole. Sculley and colleagues found the same structure inside machine learning systems, where the model code is a small part of the system and most of the maintenance cost lives in glue code, configuration, data dependencies and undeclared consumers, that is, in interfaces [21].

4. The interface literature

The thesis is not new in its parts. Its contribution is to assemble them into one statement and attach a stabilizing configuration and a test.

Structure follows communication. Conway observed in 1968 that organizations that design systems produce designs that copy the communication structure of the organization [22]. MacCormack, Baldwin and Rusnak tested this mirroring hypothesis on matched pairs of products built by tightly and loosely coupled organizations and found that the product architectures differ in the predicted direction [23]. If the bodies do not communicate across an edge, the system has an unexamined interface at that edge.

Interfaces are where design knowledge sits. Parnas argued in 1972 that a system should be divided into modules by the decisions each module hides, so that a change in one decision is confined to one module [24]. Baldwin and

Clark generalized this into design rules: visible interface decisions fixed in advance, behind which hidden modules may vary [25]. Henderson and Clark showed that established firms fail at architectural innovation, a change in how known components are linked, because their knowledge of the interfaces is embedded in their organization and goes unexamined [26].

Correctness must be checked at the ends. Saltzer, Reed and Clark’s end-to-end argument holds that a function can be completely and correctly implemented only with the knowledge of the application at the endpoints, so checks in the lower layers are at most performance aids [27]. The argument does not say that intermediate interfaces need no checks. It says that the property that matters must be checked where its meaning is known.

Contracts make assumptions explicit. Hoare gave programs preconditions and postconditions in 1969 [28]. Meyer turned them into design by contract, where every routine states what it requires and what it ensures [29]. Liskov and Wing defined when one component may replace another: when it preserves the behavior its users rely on [30]. Each is a way of writing an interface assumption down so that it can be checked.

Reuse fails at mismatched assumptions. Garlan, Allen and Ockerbloom found in 1995 that building a system from existing parts failed because each part made implicit assumptions about the others, about control, data, protocols and the environment, and that the assumptions conflicted [31]. Revisiting the problem in 2009, they found it still present [32].

Accidents are interaction failures. Leveson’s systems-theoretic account treats safety as a control problem: accidents arise from unsafe interactions between components that each meet their own specification [33]. Cook’s short treatise on complex system failure says the same from operations: catastrophe requires multiple failures, and the system usually runs in a degraded mode [34]. Lehman’s observations on software evolution add time: a program used in the world must keep changing, and its complexity grows unless work is done to reduce it [35].

Verified kernels show that a boundary can be a proof. seL4 was the first general-purpose operating system kernel with a machine-checked proof that its implementation meets its specification [36]. The proof is an interface check made total: between the specification and the code, nothing is assumed.

5. Definitions and propositions

5.1 Definitions

Definition 1 (body). A body is a party with an internally coherent model of a system: a theorist, a maker or a user. A body may be a person, a team, an organization or a program.

Definition 2 (interface). An interface is a handoff between two bodies, or between two levels of a stack, across which one side’s output becomes the other side’s input.

Definition 3 (interface assumption). An interface assumption is a predicate a over the values that cross an interface, which the receiving side relies on for its own correctness. Examples: a unit, a range, a type, a voltage, a timing bound, a version, an envelope.

Definition 4 (checked and unexamined). An assumption is checked when a mechanism evaluates a on every crossing and refuses the crossing when a fails. It is unexamined when it is relied on but is neither written down nor evaluated.

Definition 5 (stack). A stack is a sequence of levels $L_0, \dots, L_n$ where each level consumes the level below only through the interface between them. A body may own several levels; an interface may join two bodies or two levels of one body.

Definition 6 (general agent). Following Legg and Hutter, a general agent is one whose performance is measured across a wide class of environments, not one task [37]. Morris and colleagues operationalize levels of AGI by both performance and generality [38]. For this paper the relevant property is generality across bodies: the agent must act as theorist, maker and user, and across every level of the stack.

5.2 Propositions

Proposition 1 (failures originate at unexamined interfaces). If every body is correct with respect to its own model, then every failure of the composed system violates at least one interface assumption, and if every assumption is checked, the first violated assumption is refused at the interface where it occurs.

Proof sketch. If every body is correct relative to its inputs and every input satisfies the assumptions the body relies on, the composition is correct by induction over the stack. A failure therefore requires some crossing at which an assumption fails. If that assumption is checked, the crossing is refused there, so the failure is located at that interface and does not propagate. ■

Proposition 2 (unchecked mismatches propagate). If an assumption at the interface between L_k and L_{k+1} is unexamined, a violation at that interface can produce wrong values at every level above k , and no check at a level above k that is phrased only in that level's own terms can locate it.

Sketch. Levels above k see only outputs of L_{k+1} , which are well formed relative to L_{k+1} 's own model. Mars Climate Orbiter is the instance: the navigation solution was internally consistent and wrong [12].

Proposition 3 (the stabilized configuration). In a stack where every interface assumption is explicit and checked, a correction made at level k that passes the check of its own boundary leaves every property established by the checks of the other boundaries intact, and every level above k inherits the correction without change.

Sketch. The checks at the other boundaries evaluate predicates over the values that cross those boundaries. The corrected L_k changes only values that cross the boundaries of L_k , and its outgoing values satisfy the checked assumption of the boundary above it. Each level above k consumes L_k only through a checked interface, so if the corrected L_k satisfies that interface, the consumers see a correct input. ■

Proposition 4 (the pursuit of AGI forces explicitness). Let a general agent in the sense of Definition 6 be required to pass a correctness gate on tasks drawn across all three bodies and all levels of the stack. Then for every interface the task distribution crosses, either the agent makes the interface assumption explicit and checks it, or there is a task on which the agent's success depends on an unchecked guess about that assumption.

Argument. A narrow system can rely on the convention of its one environment: a fixed unit system, a fixed instruction set, a fixed version. A general agent faces environments in which those conventions differ, by construction of generality. An assumption that holds in one environment and fails in another must be detected to be handled, and detecting it is checking it. A correctness gate is a test that a run either passes or fails, with no partial credit. Under such a gate, a guess that happens to be wrong is a failed task, not partial credit. So the pursuit of generality under a gate is a pressure toward explicit, checked interfaces at every boundary the agent touches. The proposition does not say that any current system meets this standard. It says what the target requires.

Proposition 5 (joules rank the stable configurations). Among stacks that pass the gate, the one that resolves each interface with the least energy to completion is preferred, and an explicit check spent once at an interface is cheaper over a workload than a failure rediscovered at every level above it.

Sketch. The ordering is gate then rank: correctness is a constraint every candidate must satisfy, and energy to completion, in joules, orders only the candidates that satisfy it, as in constrained optimization where the objective is compared only over the feasible set. A check at the boundary has a fixed cost per crossing. An unexamined mismatch has a cost at every level it reaches, and that cost recurs on every run until the mismatch is found.

6. The stabilized configuration

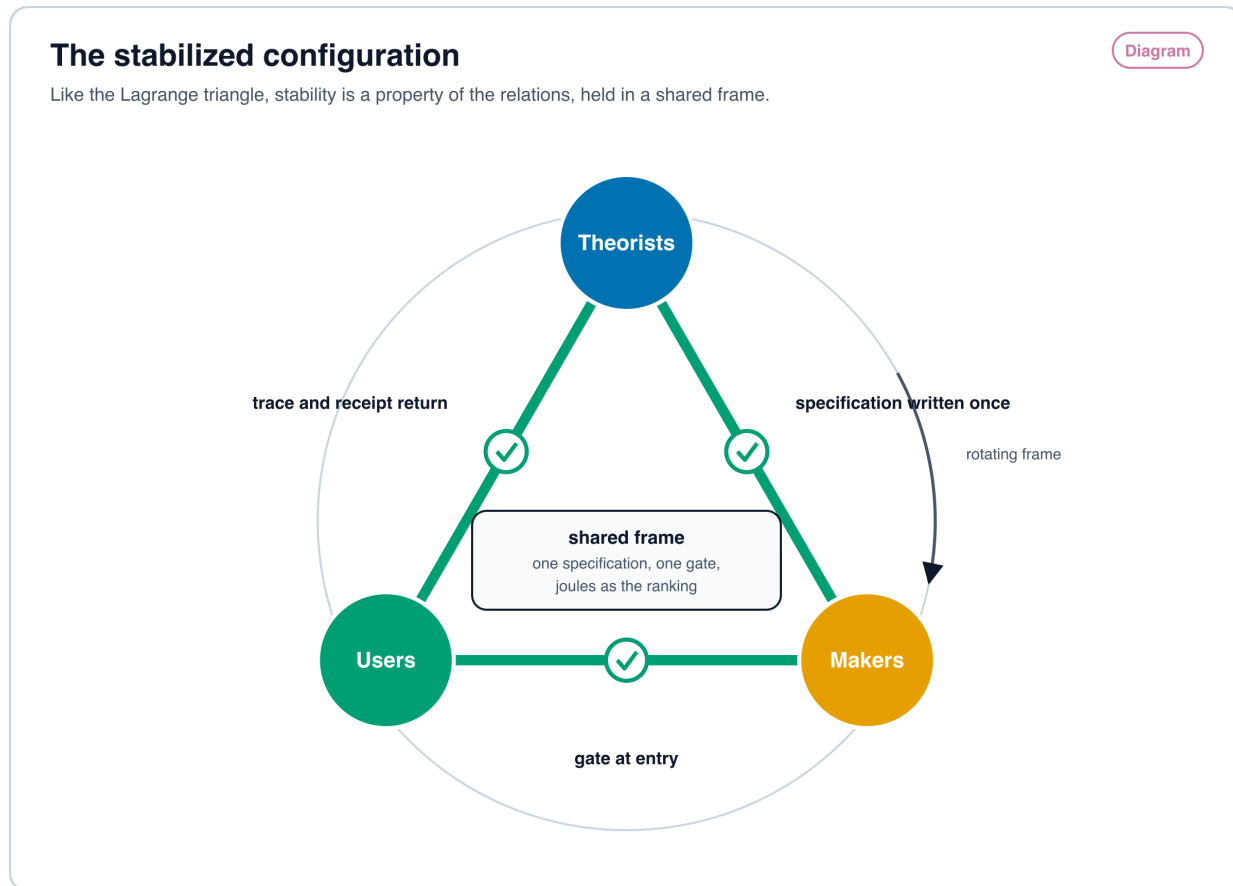


Figure 2. The stabilized configuration. Diagram. In the physics, three bodies keep their relative positions in a frame that rotates with them. In computing, the shared frame is an explicit, machine-checkable statement of intent with every interface assumption checked, so the three bodies keep their relations as the system changes. A [PNG copy](#) is provided.

6.1 The shared frame

Lagrange's configurations are stable only in a frame that moves with the bodies. The computing counterpart is a shared representation that all three bodies read and write, in which every interface assumption is a checkable statement: a formal specification, a machine-checked contract, or a type. When the theorist's statement, the maker's implementation and the user's observed behavior are related by checks against one such representation, the handoffs between them are checks rather than translations.

6.2 Every interface assumption explicit

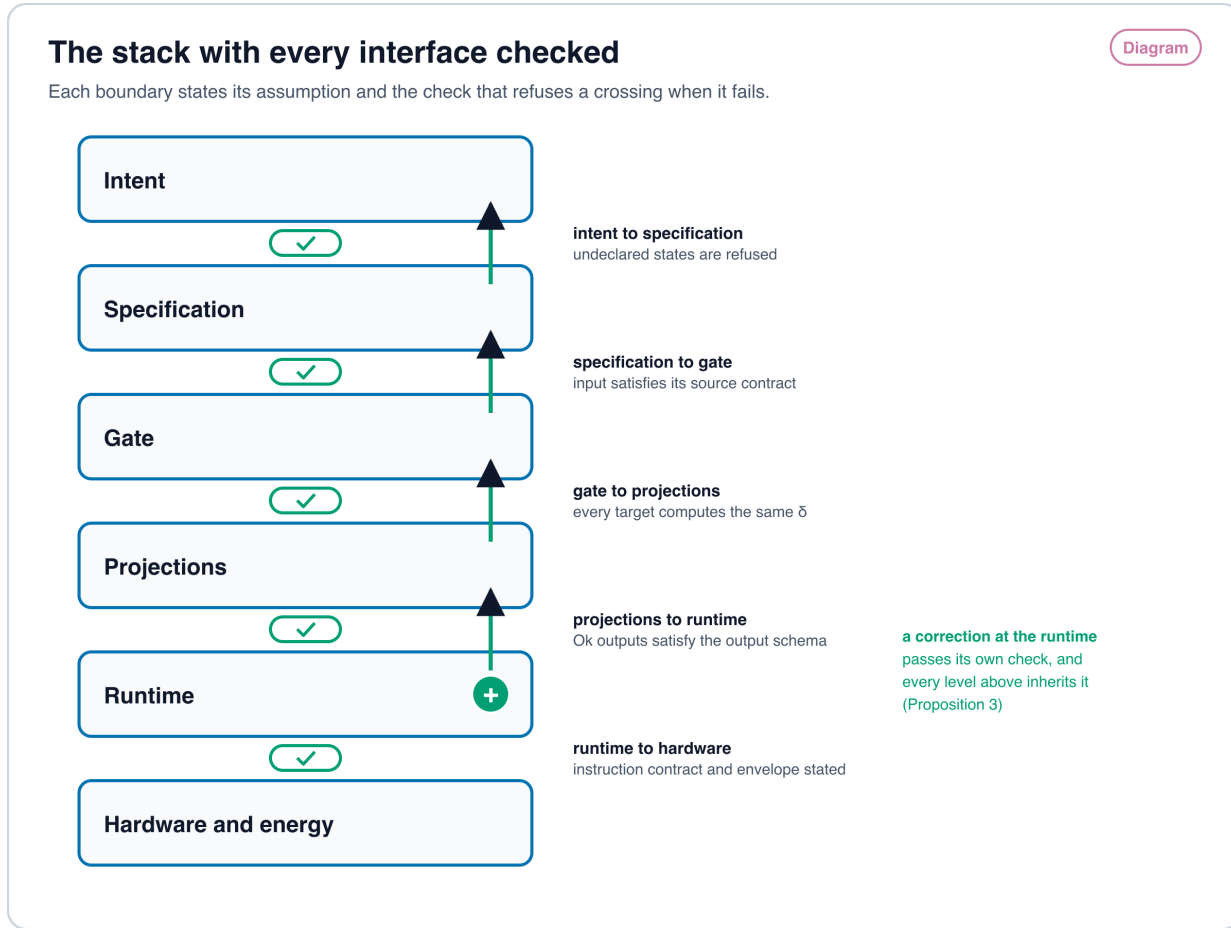


Figure 3. The stack with every interface checked. Diagram. A [PNG copy](#) is provided.

The method has three steps. List every boundary of the stack. For each, write the assumption the receiving side relies on as a predicate, and name the mechanism that evaluates it on every crossing. When a check fails, the break is located at that boundary; the correction is made at that level, passed through that boundary’s check, and inherited by the levels above.

6.3 Evidence from verified systems

The existence of stabilized configurations, in the sense of Proposition 3, is supported by systems built and tested by independent groups.

- A verified compiler under random testing. CompCert is a C compiler whose optimizing middle and back end carry a machine-checked proof that the generated code preserves the semantics of the source [39]. Yang, Chen, Eide and Regehr ran Csmith, a random program generator, against CompCert and many other compilers. They reported hundreds of previously unknown bugs in the other compilers. In CompCert they found no wrong-code errors in the verified middle end. The defects they did find sat in parts outside the proof, such as the front end and an unverified assumption about the target [40]. Failures appeared where an assumption was left unchecked, as Propositions 1 and 2 predict, and not across checked boundaries.
- A verified kernel boundary. seL4’s proof relates its C implementation to an abstract specification, so that between specification and code nothing is assumed [36]. The proof is stated relative to explicit assumptions about the compiler, the assembly code and the hardware, which shows the method of Section 6.2 in practice: the residual assumptions are written down as the remaining interfaces.
- Specifications in industrial practice. Newcombe and colleagues report that engineers at Amazon Web Services wrote TLA+ specifications for core distributed systems, and that model checking found subtle design bugs, some

needing long event sequences, that review and testing had missed [41].

- Units as a checked interface. Kennedy’s dimension types make the unit of a quantity part of its type, so that a pound-force-second value cannot be passed where a newton-second value is expected without an explicit conversion [42]. The interface assumption behind the Mars Climate Orbiter loss [12] becomes a refusal at compile time.

This evidence comes from single systems, each built with checks in mind. It shows that the stabilized configuration exists and can be maintained. It does not show that the configuration extends to stacks that span several organizations. The protocols of Section 8 are written to test that.

7. The hardware side

The hardware boundary is where interface assumptions become physical: a supply voltage, a thermal envelope, a memory model, an instruction contract. Section 3.2 gave the instruction contract as an example of a maker’s interface that is revised over time [17, 18]. A stack that treats the hardware as fixed has an unexamined interface at its lowest level, and Proposition 2 says that a mismatch there can reach every level above.

Within this paper, the hardware side is a set of interfaces to be made explicit, and a hardware result enters under the same gate and the same energy ranking as software.

8. Protocols

Each protocol is fixed before its run. Energy figures carry one of three labels: `est_j` for an analytical estimate, `reported_j` for a reading from an operating system or on-chip counter, and `measured_j` for a reading from an external power meter.

8.1 Interface inventory

For a target system, enumerate every interface between bodies and between levels. For each, record the assumption as a predicate, its owner, and whether a mechanism checks it. Report the fraction of assumptions that are checked, and the fraction that have no owner. Two independent auditors inventory the same system; report their agreement.

8.2 Break and catch

For each recorded assumption, inject a violation at that interface on a fixed seed set. Record whether the violation is refused at that interface, caught at a later level, or not caught. Proposition 1 predicts that checked assumptions are refused at their own interface. Proposition 2 predicts that unexamined assumptions, when caught at all, are caught later and are not located.

8.3 Compounding

Run a fixed workload for R rounds with and without one correction promoted at a chosen level. Record escalations, failures and energy per round. Proposition 3 predicts that cumulative escalations with the correction are no higher in any round, and that no property checked before the promotion fails after it.

8.4 Cross-body handoff

Choose a real handoff between organizations: a specification passed to an implementer, a library passed to downstream users, or a hardware revision passed to software. Express the handoff as a shared representation with explicit assumptions, run the break and catch protocol across it, and compare with the same handoff carried by prose documentation. Report failures located at the interface in each arm, with Wilson 95% intervals.

8.5 Generality under the gate

Build a task set that varies the conventions at interfaces: units, instruction contracts, library versions, encodings. Evaluate agents under a correctness gate. Proposition 4 predicts that success across the varied set requires detecting the convention, so agents that check interface assumptions explicitly should pass a larger share of the varied tasks than agents that assume a default, at matched success on the default tasks. Rank passing agents by joules to completion.

9. Falsifiers

Fixed before the runs they test.

ID	Observation	Defeats
TBP-F1	In the break and catch protocol, a violation of a checked assumption is accepted at its own interface	Proposition 1 as stated for the system under test
TBP-F2	In the compounding protocol, a correction that passes its own boundary check causes a previously passing check at another boundary to fail	Proposition 3
TBP-F3	In the cross-body protocol, the shared representation locates no more interface failures than prose documentation, within the intervals	The claim that explicit checked interfaces stabilize handoffs between bodies
TBP-F4	In the generality protocol, agents that assume defaults pass the varied tasks at a rate no lower than agents that check, at matched default performance	Proposition 4
TBP-F5	An interface inventory of a working system finds failures whose cause cannot be assigned to any interface assumption, with every body correct to its own model	The interface account of instability

10. Scope and method

The physics enters as structure. It supplies the vocabulary of bodies, relations and stable configurations, and every proposition is stated and tested in computing's own terms. Claims about sensitivity are claims about how outcomes depend on unexamined mismatches, and they are measured statistically at large N, in the manner of Stone and Leigh [7].

Explicitness is maintained, not declared. Assumptions surface as a system meets new environments, and the world a program serves keeps moving [35]. The method therefore treats each discovered assumption as a new interface to close: written as a predicate, given a check, and kept checked under Proposition 3. The stabilized configuration is held the way a spacecraft near a Lagrange point is held, by continuing station keeping.

The unit of analysis is the handoff between roles. One person can be theorist, maker and user in an afternoon, so the method applies wherever a handoff occurs, inside one team or across organizations. Every approach to building systems, from formal methods to large models with tests to hand-written code with review, is placed by the same measure: the fraction of its interface assumptions that are explicit and checked, read from the inventory protocol of Section 8.1.

11. Conclusion

The three-body problem taught physics that instability can come from the relations between simple parts, and that stability, where it exists, is a special arrangement of those relations in a shared frame. Computing has three bodies of its own, theorists, makers and users, and its instability comes from the interfaces between them, where assumptions cross unwritten and unchecked. A system that aims at general intelligence has to act across all three bodies and every level of the stack, so it cannot leave those assumptions implicit. Taking that aim seriously means writing every interface assumption down and checking it on every crossing. In a stack built that way, a correction stays where it is made and carries upward, and the system holds its shape as it changes. That is how the pursuit of AGI resolves the three-body problem in computing.

References

- [1] Liu Cixin, *The Three-Body Problem*, translated by Ken Liu. Tor Books, 2014. Original Chinese edition 三体, Chongqing Publishing House, 2008. ISBN 978-0-7653-7706-7.
- [2] H. Poincaré, "Sur le problème des trois corps et les équations de la dynamique," *Acta Mathematica* 13, 1890, pp. 1 to 270. Introduction: [doi:10.1007/BF02392506](https://doi.org/10.1007/BF02392506)

- [3] J. Barrow-Green, *Poincaré and the Three Body Problem*. History of Mathematics 11, American Mathematical Society, 1997. [doi:10.1090/hmath/011](https://doi.org/10.1090/hmath/011)
- [4] K. F. Sundman, “Mémoire sur le problème des trois corps,” *Acta Mathematica* 36, 1913, pp. 105 to 179. [doi:10.1007/BF02422379](https://doi.org/10.1007/BF02422379)
- [5] Q. Wang, “The global solution of the N-body problem,” *Celestial Mechanics and Dynamical Astronomy* 50, 1991, pp. 73 to 88. [doi:10.1007/BF00048987](https://doi.org/10.1007/BF00048987)
- [6] E. N. Lorenz, “Deterministic Nonperiodic Flow,” *Journal of the Atmospheric Sciences* 20(2), 1963, pp. 130 to 141. [doi:10.1175/1520-0469\(1963\)020<0130:DNF>2.0.CO;2](https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2)
- [7] N. C. Stone and N. W. C. Leigh, “A statistical solution to the chaotic, non-hierarchical three-body problem,” *Nature* 576, 2019, pp. 406 to 410. [doi:10.1038/s41586-019-1833-8](https://doi.org/10.1038/s41586-019-1833-8)
- [8] E. J. Routh, “On Laplace’s Three Particles, with a Supplement on the Stability of Steady Motion,” *Proceedings of the London Mathematical Society* s1-6, 1874, pp. 86 to 97. [doi:10.1112/plms/s1-6.1.86](https://doi.org/10.1112/plms/s1-6.1.86)
- [9] C. Moore, “Braids in classical dynamics,” *Physical Review Letters* 70, 1993, pp. 3675 to 3679. [doi:10.1103/PhysRevLett.70.3675](https://doi.org/10.1103/PhysRevLett.70.3675)
- [10] A. Chenciner and R. Montgomery, “A Remarkable Periodic Solution of the Three-Body Problem in the Case of Equal Masses,” *Annals of Mathematics* 152(3), 2000, pp. 881 to 901. [doi:10.2307/2661357](https://doi.org/10.2307/2661357)
- [11] M. Šuvakov and V. Dmitrašinović, “Three Classes of Newtonian Three-Body Planar Periodic Orbits,” *Physical Review Letters* 110, 114301, 2013. [doi:10.1103/PhysRevLett.110.114301](https://doi.org/10.1103/PhysRevLett.110.114301)
- [12] Mars Climate Orbiter Mishap Investigation Board, *Phase I Report*. NASA, 10 November 1999. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf
- [13] H. Wright, “Hyrum’s Law.” <https://www.hyrumslaw.com/>
- [14] J. Spolsky, “The Law of Leaky Abstractions,” *Joel on Software*, 11 November 2002. <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/>
- [15] J. L. Lions (chair), *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. European Space Agency, 19 July 1996. <https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>
- [16] G. Le Lann, “An analysis of the Ariane 5 flight 501 failure: a system engineering perspective,” *Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems*, 1997, pp. 339 to 346. [doi:10.1109/ECBS.1997.581900](https://doi.org/10.1109/ECBS.1997.581900)
- [17] RISC-V International, *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*, version 20250508, preface. <https://docs.riscv.org/reference/isa/v20250508/unpriv/colophon.html>
- [18] Apple, “Using Intel-based apps on a Mac with Apple silicon,” Apple Support, article 102527, published 21 September 2026. <https://support.apple.com/en-us/102527>
- [19] P. Zave and M. Jackson, “Four Dark Corners of Requirements Engineering,” *ACM Transactions on Software Engineering and Methodology* 6(1), 1997, pp. 1 to 30. [doi:10.1145/237432.237434](https://doi.org/10.1145/237432.237434)
- [20] S. Hooker, “The Hardware Lottery,” *Communications of the ACM* 64(12), 2021, pp. 58 to 65. [doi:10.1145/3467017](https://doi.org/10.1145/3467017)
- [21] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo and D. Dennison, “Hidden Technical Debt in Machine Learning Systems,” *Advances in Neural Information Processing Systems* 28, 2015. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcdf2674f757a2463eba-Abstract.html
- [22] M. E. Conway, “How Do Committees Invent?” *Datamation* 14(4), April 1968, pp. 28 to 31. https://www.melconway.com/Home/Committees_Paper.html
- [23] A. MacCormack, C. Baldwin and J. Rusnak, “Exploring the duality between product and organizational architectures: A test of the ‘mirroring’ hypothesis,” *Research Policy* 41(8), 2012, pp. 1309 to 1324. [doi:10.1016/j.respol.2012.04.011](https://doi.org/10.1016/j.respol.2012.04.011)
- [24] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM* 15(12), 1972, pp. 1053 to 1058. [doi:10.1145/361598.361623](https://doi.org/10.1145/361598.361623)

- [25] C. Y. Baldwin and K. B. Clark, *Design Rules, Volume 1: The Power of Modularity*. MIT Press, 2000. [doi:10.7551/mitpress/2366.001.0001](https://doi.org/10.7551/mitpress/2366.001.0001)
- [26] R. M. Henderson and K. B. Clark, “Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms,” *Administrative Science Quarterly* 35(1), 1990, pp. 9 to 30. [doi:10.2307/2393549](https://doi.org/10.2307/2393549)
- [27] J. H. Saltzer, D. P. Reed and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems* 2(4), 1984, pp. 277 to 288. [doi:10.1145/357401.357402](https://doi.org/10.1145/357401.357402)
- [28] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM* 12(10), 1969, pp. 576 to 580. [doi:10.1145/363235.363259](https://doi.org/10.1145/363235.363259)
- [29] B. Meyer, “Applying ‘design by contract,’” *Computer* 25(10), 1992, pp. 40 to 51. [doi:10.1109/2.161279](https://doi.org/10.1109/2.161279)
- [30] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems* 16(6), 1994, pp. 1811 to 1841. [doi:10.1145/197320.197383](https://doi.org/10.1145/197320.197383)
- [31] D. Garlan, R. Allen and J. Ockerbloom, “Architectural mismatch: why reuse is so hard,” *IEEE Software* 12(6), 1995, pp. 17 to 26. [doi:10.1109/52.469757](https://doi.org/10.1109/52.469757)
- [32] D. Garlan, R. Allen and J. Ockerbloom, “Architectural Mismatch: Why Reuse Is Still So Hard,” *IEEE Software* 26(4), 2009, pp. 66 to 69. [doi:10.1109/MS.2009.86](https://doi.org/10.1109/MS.2009.86)
- [33] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012. [doi:10.7551/mitpress/8179.001.0001](https://doi.org/10.7551/mitpress/8179.001.0001)
- [34] R. I. Cook, “How Complex Systems Fail,” Cognitive Technologies Laboratory, University of Chicago, 1998. <https://how.complexsystems.fail/>
- [35] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE* 68(9), 1980, pp. 1060 to 1076. [doi:10.1109/PROC.1980.11805](https://doi.org/10.1109/PROC.1980.11805)
- [36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, “seL4: Formal Verification of an OS Kernel,” *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009, pp. 207 to 220. [doi:10.1145/1629575.1629596](https://doi.org/10.1145/1629575.1629596)
- [37] S. Legg and M. Hutter, “Universal Intelligence: A Definition of Machine Intelligence,” *Minds and Machines* 17(4), 2007, pp. 391 to 444. [doi:10.1007/s11023-007-9079-x](https://doi.org/10.1007/s11023-007-9079-x)
- [38] M. R. Morris, J. Sohl-Dickstein, N. Fiedel, T. Warkentin, A. Dafoe, A. Faust, C. Farabet and S. Legg, “Levels of AGI for Operationalizing Progress on the Path to AGI,” *Proceedings of the 41st International Conference on Machine Learning*, 2024. arXiv:2311.02462
- [39] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM* 52(7), 2009, pp. 107 to 115. [doi:10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814)
- [40] X. Yang, Y. Chen, E. Eide and J. Regehr, “Finding and understanding bugs in C compilers,” *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283 to 294. [doi:10.1145/1993498.1993532](https://doi.org/10.1145/1993498.1993532)
- [41] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker and M. Deardouff, “How Amazon Web Services uses formal methods,” *Communications of the ACM* 58(4), 2015, pp. 66 to 73. [doi:10.1145/2699417](https://doi.org/10.1145/2699417)
- [42] A. Kennedy, “Types for Units-of-Measure: Theory and Practice,” in *Central European Functional Programming School*, Lecture Notes in Computer Science 6299, Springer, 2010, pp. 268 to 305. [doi:10.1007/978-3-642-17685-2_8](https://doi.org/10.1007/978-3-642-17685-2_8)