Computational Autonomy and the Halting Problem

Bradley D. Saucier
Southern New Hampshire University

November 24, 2025

Abstract

This note formalizes a simple but important link between computational autonomy and the classical limits of computability. Informally, a fully autonomous software agent is expected to decide how to respond to its environment without external supervision and without getting stuck. We model such autonomy as a nontrivial semantic property of program behavior over all inputs. Once stated at this level of generality, the main fact becomes immediate: any reasonable notion of full computational autonomy is algorithmically undecidable for Turing complete systems. Formally, the associated decision problem is an instance of Rice's Theorem and therefore inherits the same barrier as the halting problem. We give an explicit reduction from the halting problem to autonomy and then explain how practical engineering controls such as watchdog timers, bounded model checking, and static analysis operate under this theoretical ceiling by enforcing bounded or approximate forms of autonomy rather than perfection. From an engineering perspective, the message is that certification relies on architectural containment and bounded guarantees, not on a perfect autonomy oracle.

1 Introduction

As we hand more safety critical tasks to software agents, we naturally ask for guarantees. A Mars rover should explore on its own without continuous human input, but it must never drive off a cliff or sit in a deadlocked state while power runs out. Autonomous software systems are expected to operate without continuous human oversight, to make decisions based on their observations, and to avoid catastrophic failure modes such as livelock (state changes without useful progress) or unbounded waiting.

To keep the discussion concrete, we will return repeatedly to a Mars rover control program as a running example. A rover controller receives sensor readings and commands, and produces actuator outputs. A controller that always responds to commands, avoids marked hazard zones, and eventually either advances or reports a fault seems autonomous. A controller that falls into a state where it no longer processes commands or makes progress, even though the environment would allow it, does not.

It is tempting to ask for a definitive certification procedure: given a program P, decide whether P is fully autonomous in some well defined sense. For our running picture, imagine a rover control program that must always respond to commands, avoid hazard zones, and

eventually make forward progress whenever conditions allow. Can we build a tool that answers once and for all whether a given control program meets this standard of autonomy?

This paper explains why such a general procedure cannot exist for Turing complete systems. At a high level, the argument is straightforward. Any meaningful definition of full computational autonomy must speak about the observable behavior of a program over all inputs and histories. That makes autonomy a semantic property of the computed function or of the accepted language, not a syntactic property of the source code. Rice's Theorem then applies: every nontrivial semantic property of programs is undecidable. Equivalently, any algorithm that claimed to decide full autonomy would decide the halting problem.

The goals of this note are:

- to make this connection precise in a way that is accessible to readers familiar with basic computability theory,
- to show an explicit reduction from the halting problem to a natural autonomy property, and
- to explain what this impossibility means for practical engineering controls such as watchdog timers and bounded model checking.

We assume familiarity with Turing machines and the standard proof that the halting problem is undecidable [3, 2]. No background in formal verification is required beyond the notion that model checking and static analysis attempt to establish properties of programs without running them [1]. In what follows we use the terms "program", "Turing machine", and "agent" informally according to context, but all formal reasoning is carried out at the level of machine indices $e \in \mathbb{N}$.

2 Preliminaries

We fix a standard enumeration $(M_e)_{e\in\mathbb{N}}$ of Turing machines. Each machine M_e induces a partial computable function $\varphi_e \colon \Sigma^* \rightharpoonup \Sigma^*$ on finite strings over some alphabet Σ. We write

$$\varphi_e(x) \downarrow$$

if M_e halts on input x and produces an output, and

$$\varphi_e(x) \uparrow$$

if M_e does not halt on input x. It is common to read $\varphi_e(x) \downarrow$ as "phi e of x converges" or "halts" and $\varphi_e(x) \uparrow$ as "diverges".

We will work with encodings of machines and inputs as natural numbers. For a standard computable pairing function

$$\langle \cdot, \cdot \rangle \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N},$$

the code $\langle e, x \rangle$ represents the pair consisting of index e and input x.

Definition 1 (Halting problem)**.** The *halting set* is

$$H \;=\; \{\langle e, x\rangle \mid \varphi_e(x)\downarrow\},$$

where $\langle e, x\rangle$ is any standard computable encoding of a pair of natural numbers as a single natural number. The *halting decision problem* is

$$\mathsf{HALT} \;=\; \{\langle e, x\rangle \mid \langle e, x\rangle \in H\}.$$

It is classical that H is recursively enumerable (recognizable) but not recursive (decidable).

Theorem 1 (Turing [3])**.** *There is no total computable function*

$$D\colon \mathbb{N}\times\mathbb{N}\to\{0,1\}$$

such that

$$D(e,x) = \begin{cases} 1 & \text{if } \varphi_e(x)\downarrow, \\ 0 & \text{if } \varphi_e(x)\uparrow. \end{cases}$$

In other words, the halting problem is undecidable.

For the rover example, deciding HALT would amount to determining, for any controller code and any sensor or command input, whether the controller will eventually produce a response. The undecidability of HALT already hints at the impossibility of a perfect autonomy certifier.

We now recall the form of Rice's Theorem that we will use. It is convenient to state it explicitly in terms of indices of partial computable functions.

Definition 2 (Semantic property)**.** A set $P \subseteq \mathbb{N}$ is a *semantic property* of partial computable functions if membership depends only on the function computed. Formally, if $\varphi_e = \varphi_{e'}$ then

$$e \in P \quad \text{if and only if} \quad e' \in P.$$

Equivalently, P is a set of program indices that is closed under functional equivalence. For example, the property "M_e computes a total function" is semantic, while "M_e has fewer than one hundred states in its description" is not.

Definition 3 (Nontrivial property)**.** A semantic property $P \subseteq \mathbb{N}$ is *nontrivial* if

$$P \neq \emptyset \quad \text{and} \quad P \neq \mathbb{N}.$$

That is, there is at least one program that has the property and at least one that does not. Note that "nontrivial" here refers to the set of indices, not to the difficulty of checking the property in practice.

Theorem 2 (Rice [2])**.** *Let $P \subseteq \mathbb{N}$ be a nontrivial semantic property of partial computable functions. Then the decision problem*

$$\mathsf{RICE}_P \;=\; \{e \mid e \in P\}$$

is undecidable.

Rice's Theorem says that there is no algorithm which, given an arbitrary index e, decides whether the corresponding program has a nontrivial property defined in terms of its input output behavior.

3 Computational autonomy as a semantic property

Autonomy is often described informally in terms of self governance, independence from external control, or the ability to manage one's own behavior. For computability purposes we need an extensional definition: autonomy must be a property of observable behavior across all inputs, not of how the code is written.

To keep the discussion anchored, we continue to think of a Mars rover. The rover's control software receives sensor readings and commands, and produces actuator outputs. A rover that always responds to commands, avoids marked hazard zones, and eventually either advances or reports a fault seems autonomous. A rover that falls into a state where it no longer processes commands or makes progress, even though the environment would allow it, does not.

The following abstraction captures this picture while remaining compatible with standard computability theory.

3.1 Informal picture and observable behavior

We assume that each program index e induces an *observable behavior* B_e. Depending on the model, B_e might be

- a function from finite input traces to finite output traces,
- a set of infinite traces representing all possible executions, or
- a labeled transition system describing states and actions.

We do not fix one particular representation. What matters is that two implementations with the same observable behavior are considered equivalent from the point of view of autonomy. For the rover, B_e captures what it does in response to streams of sensor data and operator commands.

Note that B_e is always determined by the underlying partial function φ_e computed by M_e, though it may package that information differently. For example, B_e might be the graph of φ_e, a language of traces derived from φ_e, or a labeled transition system built from the states visited when φ_e is run on all inputs. From the autonomy perspective, two programs with the same B_e are indistinguishable.

At an intuitive level, we want computational autonomy to enforce two basic ideas:

- autonomy depends on what the agent does, not on how its code is written, and
- autonomy separates "good" agents that reliably manage their own behavior from "bad" agents that stall or violate basic safety conditions.

3.2 Abstract formalization

We now package these informal requirements from the rover scenario into a definition that applies to any reasonable notion of autonomy. The key insight is to characterize autonomy as a semantic property, one that depends only on observable behavior, and to require that it be nontrivial so that Rice's Theorem applies.

Definition 4 (Computational autonomy)**.** Fix a model in which each program index e determines an observable behavior B_e, such as a function from input streams to output streams or a language of input output traces. A set $A \subseteq \mathbb{N}$ of program indices is said to formalize *computational autonomy* if:

1. (Semantic) If $B_e = B_{e'}$ then $e \in A$ if and only if $e' \in A$. In other words, autonomy depends only on observable behavior, not on the particular implementation.

2. (Nontrivial) There exists at least one index e_{auto} such that $e_{\text{auto}} \in A$ and at least one index e_{non} such that $e_{\text{non}} \notin A$.

We call such a set A a *full computational autonomy property.*

Condition (1) enforces the usual software engineering intuition that two implementations with identical behavior are equally autonomous. Condition (2) rules out degenerate definitions in which every program is autonomous or no program is. The definition does not fix a particular list of requirements such as "eventually respond to every request" or "never enter a hazard state". Instead, it characterizes the shape that any reasonable full autonomy specification must have to live in the semantic world.

Example 1 (A concrete autonomy property)**.** Return to the rover example. Fix a set S of "acceptable" observable behaviors in which, for every admissible environment:

- every command is eventually acknowledged or explicitly rejected,

- the rover never drives into a region labeled as hazardous, and

- whenever the terrain permits safe movement, the rover eventually either moves or reports that it is stuck and requests help.

Define A to consist of all indices e such that:

1. for every input, the computation of M_e terminates in finite time at each step of the interaction, and

2. the resulting observable behavior B_e belongs to S.

Intuitively, such a program is autonomous because it never gets stuck internally and always maintains the desired safety and progress conditions on its own. This A is semantic and nontrivial, so it satisfies the definition above. It is one concrete instance of a full computational autonomy property and illustrates how a real world autonomy specification naturally yields a semantic, nontrivial set of indices.

The point of the abstraction is that we do not have to fix one particular S or one particular interaction model. Any autonomy property that respects functional equivalence and is nontrivial falls under Rice's Theorem.

4 Main result

We can now state the main theorem.

Theorem 3 (Undecidability of full computational autonomy)**.** *Let $A \subseteq \mathbb{N}$ be any set of program indices that formalizes full computational autonomy in the sense of the previous definition. Then the decision problem*

$$\mathsf{AUTONOMY}_A \;=\; \{e \mid e \in A\}$$

is undecidable.

Proof via Rice's Theorem. By definition, A is semantic and nontrivial. Therefore Rice's Theorem applies, and $\mathsf{AUTONOMY}_A$ is undecidable. □

For readers who prefer to see directly how the halting problem hides inside autonomy, we now give an explicit reduction argument. This is essentially Rice's Theorem carried out by hand for the particular property A.

Theorem 4 (Explicit reduction from the halting problem)**.** *Let $A \subseteq \mathbb{N}$ be a nontrivial semantic property formalizing computational autonomy. Then there is no computable function*

$$\mathit{IsAuto}\colon \mathbb{N} \to \{0,1\}$$

such that

$$\mathit{IsAuto}(e) = \begin{cases} 1 & \text{if } e \in A, \\ 0 & \text{if } e \notin A. \end{cases}$$

Proof. Assume, for contradiction, that such an IsAuto exists.

Because A is nontrivial, there exists at least one index in A. Let $e_{\text{yes}} \in A$ be any such index. In addition, we assume, consistent with the informal notion of autonomy used throughout, that a program which diverges on every input is not considered autonomous. Let e_{no} be an index of such an everywhere diverging machine, with $e_{\text{no}} \notin A$. Think of $M_{e_{\text{yes}}}$ as a well behaved rover controller that always manages its own behavior correctly, and $M_{e_{\text{no}}}$ as a clearly nonautonomous program that never produces a response on any input.

We will use IsAuto to decide the halting problem. The key idea is the standard "program as data" construction. Given a pair $\langle e, x\rangle$, we effectively build source code for a new machine that first simulates $M_e(x)$ and then, if that simulation ever halts, behaves like $M_{e_{\text{yes}}}$ on all future inputs. If the simulation never halts, the new machine never progresses beyond the simulation phase, so its observable behavior is divergence on every input, matching $M_{e_{\text{no}}}$. The computable function g defined below performs this code generation. The actual simulation takes place only when the resulting machine runs, not when g itself is computed.

Formally, given an arbitrary pair $\langle e, x\rangle$, we define a total computable function g that returns an index $g(e,x)$ of a machine $M_{g(e,x)}$ with the following behavior on any input y.

Construction of $M_{g(e,x)}$.

1. Simulate $M_e(x)$ step by step. If and when this simulation halts, record that fact and proceed to step 2.

2. After the simulation of $M_e(x)$ has halted, ignore x and emulate $M_{e_{\text{yes}}}$ on input y.

This construction is effective: there is a single total computable function g that, given $\langle e, x\rangle$, returns an index $g(e, x)$ of a machine that implements the above behavior. In summary, $M_{g(e,x)}$ runs $M_e(x)$ once as a test and, if that test ever finishes, permanently adopts the behavior of $M_{e_{\text{yes}}}$. If the test never finishes, $M_{g(e,x)}$ never produces any output on any input.

Now analyze $M_{g(e,x)}$ at the level of observable behavior $B_{g(e,x)}$.

If $\varphi_e(x) \downarrow$, then for every input y the simulation in step 1 halts after finitely many steps and $M_{g(e,x)}$ subsequently behaves like $M_{e_{\text{yes}}}$. Because A is semantic and $e_{\text{yes}} \in A$, we have $g(e, x) \in A$ in this case.

If $\varphi_e(x) \uparrow$, then the simulation in step 1 never halts, and $M_{g(e,x)}$ never reaches step 2. For every input y, the machine $M_{g(e,x)}$ diverges without producing output. By construction, $M_{e_{\text{no}}}$ also diverges on every input, so $B_{g(e,x)} = B_{e_{\text{no}}}$. Again by semanticity and $e_{\text{no}} \notin A$, we have $g(e, x) \notin A$.

Therefore

$$\langle e, x\rangle \in H \quad \text{if and only if} \quad g(e, x) \in A.$$

By assumption,

$$\text{IsAuto}(g(e, x)) = \begin{cases} 1 & \text{if } \langle e, x\rangle \in H, \\ 0 & \text{if } \langle e, x\rangle \notin H. \end{cases}$$

So we could decide the halting problem by computing $\text{IsAuto}(g(e, x))$, which contradicts the undecidability of H. Hence no such IsAuto exists. □

Remark (Recognizability)**.** The halting set H is recognizable but not decidable. If a notion of autonomy requires a program to detect and avoid nontermination on all inputs, then it implicitly asks for a decision procedure for the complement of H, which is not even recognizable. In that strong sense, perfect self monitoring for liveness faces an even stronger barrier than undecidability: the complement of H cannot even be semi decided.

5 Implications for engineering practice

We have shown that full computational autonomy, defined as a nontrivial semantic property, is undecidable by Rice's Theorem and, more concretely, that any hypothetical autonomy checker could be turned into a halting oracle. What does this mean for building real systems such as rover controllers in practice?

The short answer is that undecidability does not go away, but it is replaced in many cases by intractability: exact analysis is possible in principle but impossible at realistic scales. Practical techniques therefore enforce bounded or approximate forms of autonomy and delegate ultimate control to external enforcement mechanisms.

5.1 The finite state objection

A common objection is that real computers are finite state machines, so halting and other properties should be decidable. At a mathematical level this is correct: given a fully detailed finite state model with a fixed amount of memory, a search of the state space can in principle determine whether a particular state is reachable or whether the system eventually halts.

The problem is that the size of that state space is astronomical. A device with one gigabyte of memory has roughly $2^{8 \cdot 2^{30}}$ possible memory states, ignoring processor registers and external devices. Algorithms that are exhaustive at that scale are not usable. For a rover scale autonomous controller with sensors, actuators, and communication interfaces, the effective state space is even larger once we include environment variables. In that sense, undecidability at the Turing machine level and intractability at the finite state level describe the same operational reality: there is no general purpose procedure that can certify full autonomy for arbitrary software.

5.2 Watchdog timers and timeouts

A watchdog timer is a hardware or software mechanism that enforces a maximum time between "heartbeat" signals from a monitored task. If the task fails to refresh the watchdog in time, the watchdog assumes the task is stuck and forces a reset or transition to a safe state.

Formally, a watchdog converts the question

"Will this computation ever halt?"

into the weaker and decidable question

"Has this computation halted within T steps or T seconds?"

for some fixed bound T. This trades completeness for safety: a task that would have halted at $T + 1$ steps is treated as faulty. The situation mirrors the halting problem: the original property of eventual termination is undecidable in the idealized model, but its bounded approximation is decidable and implementable.

In the context of computational autonomy for the rover, a watchdog timer is an external enforcement mechanism. It does not make the rover controller itself more autonomous in the strong, undecidable sense captured by $\mathsf{AUTONOMY}_A$. Instead, it ensures that a controller which fails to make progress is preempted by a separate component whose behavior can be modeled and verified as a finite state system. The cost is the possibility of false positives: a rare but correct long running computation may be cut off as if it were a failure. This is a direct reflection of the main undecidability theorem: we do not solve the undecidable problem, we weaken the specification to a bounded variant that is decidable.

5.3 Bounded model checking

Bounded model checking (BMC) encodes the behavior of a finite state system over a fixed number k of steps into a propositional or satisfiability modulo theories (SMT) formula and

asks whether there is a counterexample of length at most k to a given property [**biere1999**, 1]. If a counterexample exists, it can often be produced automatically. If no counterexample exists up to depth k, BMC reports that the property holds for all executions of length at most k.

BMC is naturally suited to safety properties of the form "something bad never happens" within the bound. Liveness properties such as "every request is eventually answered" require reasoning about unbounded traces. Several techniques extend BMC in that direction:

- lasso shaped counterexamples, where a finite prefix leads into a loop that repeats forever,
- completeness thresholds, which bound the search depth needed to prove a property for a given finite state model, and
- k induction, which combines BMC style base cases with inductive invariants to prove unbounded safety properties.

Even with these extensions, BMC does not decide full autonomy for arbitrary code. Instead, it respects the undecidability barrier established above by restricting attention to executions of bounded length or to finite state models with known completeness thresholds, both of which avoid the full generality that triggers Rice's Theorem. Within these limits, it can give high assurance that a particular implementation of the rover controller satisfies a given specification.

5.4 Static analysis and abstract interpretation

Static analysis tools approximate program behavior without executing the program. Abstract interpretation frameworks compute over abstract domains that summarize sets of concrete states. Because autonomy is a semantic property, Rice type limitations still apply at the abstract level unless the abstraction is very coarse.

In practice, tools are usually designed to be sound but not complete with respect to a given model: if a warning is not reported, the property is guaranteed to hold within the abstraction, but reported warnings may be false positives. From the autonomy perspective, static analysis can establish that under certain assumptions on inputs and environment, an agent never reaches specified bad states and always follows certain progress patterns. It cannot provide a general decision procedure for full autonomy across all possible environments.

For the rover example, a static analyzer might prove that under specified sensor and terrain models, the controller never commands a move into a hazard region and always eventually either moves or signals that it is stuck. It will still either miss some subtle failures (if it is incomplete) or flag some safe controllers as possibly unsafe (if it is sound but imprecise). Both behaviors are direct reflections of the underlying undecidability of general semantic properties such as $\mathsf{AUTONOMY}_A$.

6 Conclusion

The intuitive idea that a fully autonomous system would need to understand and control its own future behavior runs directly into the classical undecidability results of computability

theory. Once autonomy is modeled as a nontrivial semantic property of program behavior, Rice's Theorem implies that there is no algorithm that decides, for arbitrary code, whether that code is fully autonomous. An explicit reduction from the halting problem makes this dependence concrete.

For engineering practice, the lesson is not that autonomy is impossible, but that perfect, once and for all certification of full autonomy for arbitrary software is impossible. Instead, systems engineers employ a layered approach:

- design autonomous components against clear specifications,
- use verification tools such as model checking and static analysis within their known limits, and
- deploy external enforcement mechanisms such as watchdog timers to contain failures when autonomy breaks down.

Returning to the rover example, we do not ask a single analysis tool to decide whether every possible controller is fully autonomous. We design and verify specific controllers against formal models, add watchdogs and recovery mechanisms to bound the impact of failures, and treat residual risk as an architectural and operational issue rather than as a missing theorem.

Understanding the underlying computability barriers clarifies what existing techniques can and cannot guarantee, and it highlights where remaining risk must be managed by design rather than by deeper analysis.

References

[1] Jr. Clarke Edmund M. et al. *Model Checking*. 2nd ed. The MIT Press, 2018. ISBN: 978-0262038836.

[2] Michael Sipser. *Introduction to the Theory of Computation*. 3rd ed. Cengage Learning, 2012. ISBN: 978-1133187790.

[3] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. s2 42.1 (1936), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.