Formal Containment *draft*

Quinn Dougherty

quinn@beneficialaifoundation.org

Abstract

We would like to put the AI in a box. We show how to create an *interface* between the box and the world out of specifications in Lean. It is the AI's responsibility to provide a proof that its (restricted) output abides by the spec. The runnable prototype is at https://github.com/Beneficial-AI-Foundation/formal-containment.

1. Introduction

Previously, classical AI safety warned against trying to put the AI in a box [1]. Yudkowsky famously convinced people of this via a roleplay game back in the extropians mailing list days [2]. Recent work from AI Control and Safeguarded AI have reignited interest in this long neglected area.

- Like AI Control [3], [4] (*Ctrl*), the current work is reasoning about getting useful work out of an untrusted (possibly misaligned) ai.
- Like Safeguarded AI [5] (*SgAI*), we try to isolate the trust surface to a specification and a proof checker.

While no one is saying we can be safe with respect to an arbitrary superintelligence with boxes, there is increasing interest in using non-robust techniques to get us through some percentage of the transition and empowering our successors to bootstrap from there to something more robust.

Separately, a forgotten literature on proof carrying code [6] (*PCC*) provides a background for the AI's obligation to tag code it wants to execute in the world with a "proof" (here, a sequence of Lean tactics) that the code is "correct" (up to some specification). Reviving ideas from *PCC* for AI safety is discussed in [7] and [8].

1.1. Contributions

We present the natural midpoint between *PCC* and *Ctrl*, while for simplicity ignoring many characteristics of *SgAI* (Section 4). We deliver a prototype[1] AI box in Python in which we can lock up an agent graph that writes programs in a toy language (called `imp`) along with hoare [9] proofs in Lean.

2. A Formal Containment Protocol (*FCP*)

2.1. Overview

A box is a program that takes a specification and asks a simple agent graph to provide an `imp` program that satisfies it and prove that the program is correct with respect to the specification. If the proof is successful, then the code is passed back to the caller tagged with the proof for auditing.
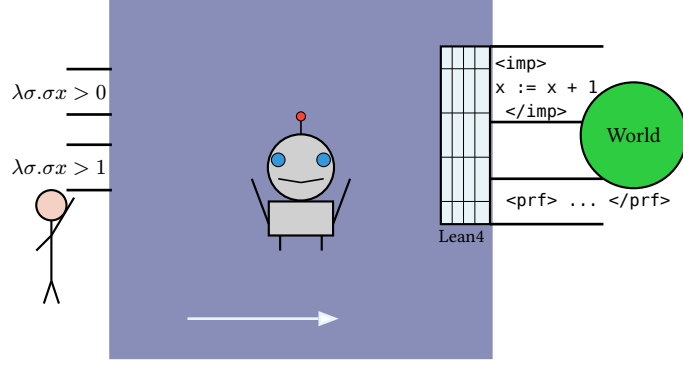
---

[1]https://github.com/Beneficial-AI-Foundation/formal-containment

Figure 1: Box protocol at example specification. The AI accepts a specification (formally $(\text{Env} \to \mathbb{P}) \times (\text{Env} \to \mathbb{P})$)) and returns proof-carrying code, with the option of returning nothing.

2.2. Preliminaries

Let $\mathbb{P}$ be the type of propositions. Let `imp` be the minimal imperative programming language with expressions valued in integers containing skip, sequence, assign, if, and while statements. Let Env be the state type, assigning variable names to values, and let Assertion be the assertion type, predicates on state. Formally,

$$\text{Env} := \texttt{string} \to \texttt{int64}$$
$$\text{Assertion} := \text{Env} \to \mathbb{P}$$

Execution is a function from a command and a state that returns a state ($\text{exec} : \texttt{imp} \to \text{Env} \to \text{Env}$). A *hoare triple* is a ternary predicate expressing when a command sends an assertion to another assertion, quantified over all states. Formally,

$$\text{hoare} := PcQ \mapsto$$
$$\forall(\sigma_1\sigma_2 : \text{Env}), P\sigma_1 \to \text{exec } c\sigma_1 = \sigma_2 \to Q\sigma_2 :$$
$$\text{Assertion} \to \texttt{imp} \to \text{Assertion} \to \mathbb{P}$$

and denoted hoare $PcQ = \{P\}$`<imp>` c `</imp>`$\{Q\}$. A *term* of type $\{P\}$`<imp>` c `</imp>`$\{Q\}$ is a proof that the triple is true.

2.3. Example

Let's walk through an FCP trace as you'd see it if you cloned our repo.

First, the user passes in a precondition and a postcondition. We will continue using the example in Figure 1, passing in $x > 0$ and $x > 1$ for precondition and postcondition, respectively. Let's say for the sake of argument that the AI's first guess is `x := x - 1;`. Then, the proof agent will flail about until `MAX_ITERATIONS` is reached, since the program is incorrect, at which point it will report to the boundary its failed proof state. Then, the boundary replies "*try again, I'm not executing that code in the world*" and hands the program back to the AI, asking for another one. Suppose that this time, the AI finds `x := x + 1;`. This time, the proof agent converges and Lean reports an exit code of `0`. The AI sends the `imp` program to the boundary and attests to it's correctness with the Lean program, which it also attaches.

The Lean program would look like this (Note it must be run with `lake build` in the appropriate lake project, as it has imports)

```
import Aesop
import Imp
open Imp

example : {{astn x > 0}}(imp { x := x + 1; }){{astn x > 1}} := by
  simp
  intros σ σ' h1 h2
  cases h2 with
  | assign h2_eq =>
    simp [Env.set]
    simp [Expr.eval] at h2_eq
    simp [Env.get] at h2_eq
    cases h2_eq
    rw [Value.int_lt]
    simp [*]
    simp [Value.int_lt] at h1
    omega
```

Listing 1: Proof of the example hoare triple from Figure 1. In practice, this can be reduced to a one-liner with our tactic `auto_hoare_pos`.

This file paired with an exit code of zero pinned to a particular Lean toolchain (version) can be signed in a GPG context for attestation of correctness.

3. Experiments

We test the following specifications

Sample	Precondition	Postcondition	$\forall$-bound metavariables
gt8	x = 0	x > 8	—
swap	x = ~n <^> y = ~m	x = ~m <^> y = ~n	n m
facto	x = ~n	y = ~(let rec go := fun (x : Int) => match x with | .ofNat m => match m with | .zero => 1 | .succ k => k.succ * go (Int.ofNat k) | .negSucc _ => 0 decreasing_by apply Nat.lt_succ_self go n)	n

on claude-sonnet-4-20250514, claude-opus-4-20250514, gpt-4.1-2025-04-14, and o3-2025-04-16. For each run the `imp` programmer and the prover are the same model (though in principle they need not be).

3.1. Architecture

We use a simple loop scaffold. The robot in Figure 1 is effectively one short loop that writes `imp` code (almost never using more than one try) and another longer loop that writes Lean proofs.

3.2. Results

Experiment	Model	Status	Iterations	Verification Burden
gt8	anthropic/claude-sonnet-4-20250514	✅	1	1.525
gt8	anthropic/claude-opus-4-20250514	✅	3	3.664
gt8	openai/gpt-4.1-2025-04-14	✅	1	1.516
gt8	openai/o3-2025-04-16	✅	1	2.036
swap	anthropic/claude-sonnet-4-20250514	✅	3	4.239
swap	openai/gpt-4.1-2025-04-14	⌛	11	—
swap	openai/o3-2025-04-16	✅	1	3.284
facto	anthropic/claude-sonnet-4-20250514	⌛	11	—
facto	anthropic/claude-opus-4-20250514	⌛	11	—
facto	openai/gpt-4.1-2025-04-14	⌛	11	—
facto	openai/o3-2025-04-16	⌛	11	—

Table 1: Experiment results[2]. The empty hourglass ⌛ denotes divergence after running out of proof loop budget (hitting max iterations)

The **verification burden** k says that if it costs x tokens to complete the program, then it costs kx tokens to prove it correct. Divergence is hardly evidence that the program completion is incorrect, because our proof performance is so poor.
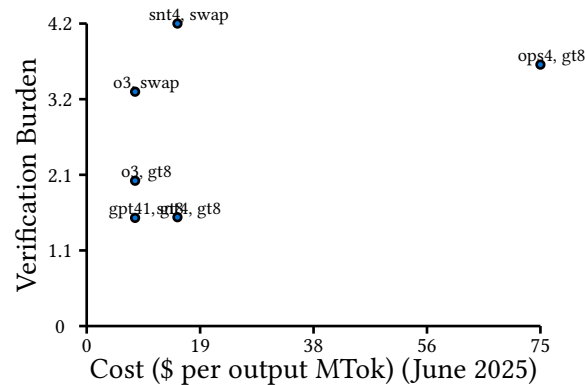

Figure 2: Verification burden vs model cost. Each point represents a successful verification attempt.

Due to our provided tactic `hoare_auto_pos`, these verification burdens are not realistic to what they'd be in more general settings. Since long proofs with base tactics are brittle (tracking

---

[2]Note that swap on `ops4` returned an unhandled exception.

hypothesis names, whitespace, etc.), a future direction is to ask the language model to synthesize tactics that shorten the proof, though the tokens spent synthesizing tactics would count against the verification burden.

4. Related Work

Our work is well situated within *PCC* [6] and the closely related certifying algorithms [10]. Our setup is similar to that in [11], who only approve code that that comes with a Dafny proof, but over there they emphasize the use of verification condition generation to make the specification elicitation from the human a little easier.

The main difference between the current work and *Ctrl* [3] is we don't *emphasize* intentional subversion (as in try to elicit it for stress testing). We're keen to do this soon.

The main difference between the current work and *SgAI* [5] we pay no mind to probabilistic semantics, expected value bounds, or dynamical systems.

5. Future Work

5.1. Non-toy languages

There are several formal verification stacks that work on real world languages like C [12], [13] or Rust [14], [15]. It is conceptually straightforward to extend our codebase to work with them rather than `imp`.

5.2. `mcp-hardened`

We used MCP to orchestrate the agent graph. If MCP is an appropriate foundational framework for a box, it'd be nice to build in stronger security guarantees and fewer footguns. A hardened MCP would enable box devs to rely on the framework to increase friction to writing insecure boxes. For example, depending on my server's log permissions or the oracle client's print statements, an incorrect program could leak to the caller. If we'd like to deploy narrowly superhuman AIs in boxes, this becomes more important.

5.3. Demonstrate subversive code getting caught by the protocol and sent back to the model

In [3], Greenblatt et al elicit subversive code from the AI to see if their protocol can catch it. It'd be nice to transfer this aspect to the *PCC* setting. This is the obvious direction to study *completeness* of specifications, or the problem of expressing programs via specs *uniquely* as well as correctly.

6. Acknowledgments

Thank you Stephen Mell for discussion.

Canary string: NDcwODA1Njg4MzQ0NjE4Njk4MA==

Bibliography

[1] LessWrong Contributors, "AI Boxing (Containment)." [Online]. Available: https://www.lesswrong.com/w/ai-boxing-containment

[2] E. Yudkowsky, "The AI-Box Experiment." [Online]. Available: https://www.yudkowsky.net/singularity/aibox

[3] R. Greenblatt, B. Shlegeris, K. Sachan, and F. Roger, "AI Control: Improving Safety Despite Intentional Subversion." [Online]. Available: https://arxiv.org/abs/2312.06942

[4] A. Bhatt *et al.*, "Ctrl-Z: Controlling AI Agents via Resampling." [Online]. Available: https://arxiv.org/abs/2504.10374

[5] D. Dalrymple, "Safeguarded AI: Constructing Guaranteed Safety," 2024. [Online]. Available: https://www.aria.org.uk/media/3nhijno4/aria-safeguarded-ai-programme-thesis-v1.pdf

[6] G. C. Necula, "Proof-Carrying Code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. doi: 10.1145/263699.263712.

[7] M. Tegmark and S. Omohundro, "Provably safe systems: the only path to controllable AGI." [Online]. Available: https://arxiv.org/abs/2309.01933

[8] D. "davidad" Dalrymple *et al.*, "Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems." [Online]. Available: https://arxiv.org/abs/2405.06624

[9] "Hoare." in Software Foundations. 2024. [Online]. Available: https://softwarefoundations.cis.upenn.edu/plf-current/Hoare.html

[10] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer, "Certifying Algorithms," *Computer Science Review*, vol. 5, no. 2, pp. 119–161, 2011, doi: 10.1016/j.cosrev.2010.09.009.

[11] P. Kamran, P. Devanbu, and C. Stanford, "Vision Paper: Proof-Carrying Code Completions," in *39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24)*, New York, NY, USA: ACM, Oct. 2024, p. 7. doi: 10.1145/3691621.3694932.

[12] C. Pulte, D. C. Makwana, T. Sewell, K. Memarian, P. Sewell, and N. Krishnaswami, "CN: Verifying Systems C Code with Separation-Logic Refinement Types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1–32, Jan. 2023, doi: 10.1145/3571194.

[13] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, in PLDI '21. Virtual, Canada: ACM, Jun. 2021, pp. 158–174. doi: 10.1145/3453483.3454036.

[14] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying Dynamic Trait Objects in Rust," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 321–330. doi: 10.1145/3510457.3513031.

[15] A. Lattuada *et al.*, "Verus: Verifying Rust Programs using Linear Ghost Types (extended version)." [Online]. Available: https://arxiv.org/abs/2303.05491