

ACIES: A Self-Evolving Agentic Contract-Based Framework for Cyber-Physical System Control Synthesis

Yifeng Xiao¹, Zhiyu Ni¹, Dinghong Song², Mi Zhou³, Pierluigi Nuzzo¹

¹University of California, Berkeley ²University of California, Merced

³Georgia Institute of Technology

{yifeng_xiao, zhiyuni, pierluigi.nuzzo}@berkeley.edu; dsong15@ucmerced.edu; mzhou91@gatech.edu

Abstract

Large language models (LLMs) are increasingly being explored in cyber-physical system design as a promising tool to facilitate the conversion of natural-language requirements into formal specifications, and the generation of plans and controllers that provably satisfy them. Existing LLM-based pipelines, however, mostly validate the designed controllers by simulation, and carry no mechanism for improving their outcomes from past designs. We present ACIES, a self-evolving agentic contract-based framework for cyber-physical system control synthesis. Given a set of system requirements, ACIES automatically generates a control architecture that realizes the requirements using a sequence of synthesis episodes. Within each episode, a *Translator* agent formalizes the system requirements into temporal logic, and a *Decomposer* agent structures them into a system-level contract and proposes a decomposition into component-level contracts that can be independently implemented. A formal verification step is then integrated in the decomposition loop to check whether the component-level contracts are consistent, compatible, and provide a correct refinement of the system-level contract. Each failed check, at the verification or implementation step, returns a diagnostic signal that drives automated contract repair at the system or the component level. Across episodes, a *Summarizer* agent distills verification outcomes into reusable lessons stored in a memory. Evaluation results on five control synthesis tasks with eight LLM backbones show that the formalized system requirements are successfully decomposed and mapped to a synthesizable control architecture that realizes them, while self-evolution reduces the Decomposer token consumption by up to 65% and the end-to-end synthesis latency by up to 62%.

1 Introduction

Distinguished by the ability to process data and execute closed-loop control actions via the inte-

gration of physical processes with computation, cyber-physical systems (CPSs) have been deployed in various domains. CPS design remains, however, a challenging problem, calling for comprehensive frameworks that allow exploring complex design spaces, formalizing system specifications, and verifying heterogeneous system properties (Sangiovanni-Vincentelli et al., 2012). Large language model (LLM) agents are increasingly being explored as a promising tool for automating CPS design: they can help draft formal specifications from natural-language requirements, plan missions, and write control programs that satisfy them (English et al., 2025; Liang et al., 2023). However, in existing workflows, the proposed controllers are mostly validated by simulation, rather than proven to meet the specifications.

Assume-guarantee (A/G) contracts offer effective and rigorous mechanisms for formal and scalable requirement analysis and system verification via design abstraction and decomposition across heterogeneous domains (Benveniste et al., 2018; Nuzzo et al., 2018). A major bottleneck, however, is authoring: writing contracts demands joint expertise in the application domain and in formal logic. LLMs-based autoformalization shows the promise of automating parts of this path, by helping convert natural-language requirements into formal contracts and drive tasks such as automated planning or control synthesis (Wu et al., 2022; Chen et al., 2023; Liu et al., 2022; Chen et al., 2024). However, while verification methods can be effectively used to assess the correctness of an LLM’s output, systematic methods that can iteratively improve the outcome based on incorrect results from past designs are still missing.

In this paper, we address these challenges with ACIES, an agentic contract-based framework that can achieve automated formalization of system requirements and provably correct controller synthesis while learning from its mistakes. As shown

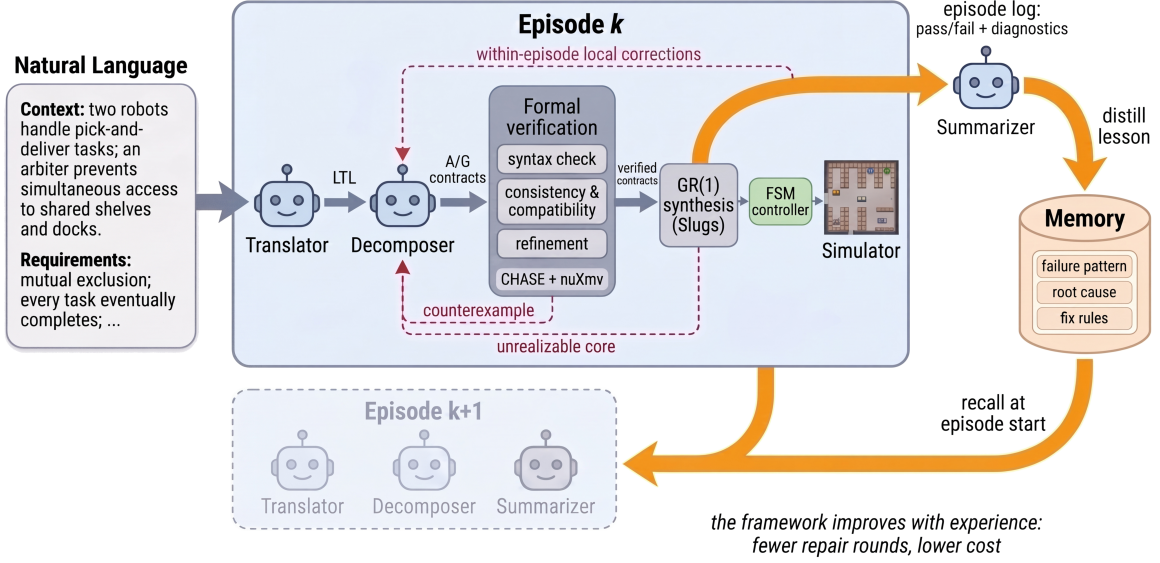


Figure 1: Overview of the ACIES framework.

in Figure 1, ACIES automatically generates a control architecture that realizes the requirements using a sequence of synthesis episodes. Within one episode, a *Translator* agent converts natural-language requirements into linear temporal logic (LTL) formulas and a *Decomposer* agent structures them into a system-level A/G contract and a set of component-level contracts that can be independently implemented and validated via formal verification and synthesis methods. Each verification or synthesis failure, whether because one or more contracts are inconsistent or incompatible, because they do not provide a correct refinement of the system-level contract, or because they are unrealizable, returns a diagnostic signal that the Decomposer can use to repair the contracts. Across episodes, a *Summarizer* agent condenses what verification revealed into a structured lesson that can be recalled from memory in the next episode.

Our contributions can be summarized as follows. We introduce contracts as a verification layer for LLM agents, enabling conversion of natural-language requirements into modular, verifiable specifications that can lead to provably correct implementations. We augment the agentic design framework with episodic memory, capable of distilling verification verdicts into reusable lessons. We evaluate the proposed approach on five control synthesis tasks and eight LLM backbones, showing that ACIES yields controllers that realize the

formalized system requirements, while cutting via self-evolution the Decomposer token consumption by up to 65% and the end-to-end (E2E) synthesis latency by up to 62%.

2 Related Work

From language to temporal logic. Methods for translating natural language (NL) into temporal logic have significantly progressed over the years, from techniques mostly based on hand-crafted grammars (Brunello et al., 2019) to fine-tuned translators such as NL2TL (Chen et al., 2023) and recent prompt-based methods (Cosler et al., 2023; Xu et al., 2024; Bayat et al., 2026). Lang2LTL (Liu et al., 2022) and VLTL-Bench (English et al., 2025) extend NL-to-LTL translation to account for concrete environments or contexts by grounding atomic propositions in environment-specific state spaces. Differently from these methods, focused on the direct translation of natural-language statements to standalone temporal-logic formulas, ACIES’s translations are also grounded in a system’s structure, in that they map the natural-language requirements to a contract architecture, including component-wise contracts together with a system-level contract.

Contract-based design and verification-guided LLMs frameworks. Contract-based design provides a compositional methodology for specifying

ing and verifying complex systems (Benveniste et al., 2018; Sangiovanni-Vincentelli et al., 2012; Nuzzo et al., 2018). CHASE (Nuzzo et al., 2018) offers an LTL contract language supporting consistency, compatibility, and refinement checking, via nuXmv (Cavada et al., 2014), and reactive synthesis (Piterman et al., 2006), via Slugs (Ehlers and Raman, 2016). ACIES augments CHASE with automated contract decomposition and repair.

Frameworks that combine LLMs with verifiers have been adopted in the context of theorem proving (Ospanov et al., 2025) and verified code generation (Gloeckle et al., 2026). Memory-based self-improvement is also well established for LLM agents (Shinn et al., 2023; Zhao et al., 2024). ACIES instantiates these approaches in the context of contract generation and repair by integrating an LLM-guided decomposition process with formal-verification workflows and further extending the verification-guided learning loop *across* episodes via memory.

3 The ACIES Framework

As illustrated in Figure 1, ACIES couples three LLM agents with a formal verification and synthesis toolchain. The input is a natural-language *context* (the system’s components, communication structure, and initial control architecture) and a set of natural-language *requirements* covering environment assumptions, coordination constraints (e.g., mutual exclusion), and system-level goals (e.g., every task eventually completes). The output is a set of reactive controllers represented by finite-state machines (FSMs) whose closed-loop execution with the environment satisfies the system-level requirements and achieves the system-level goals under every environment behavior consistent with the assumptions.

Problem statement. Given a system architecture $\mathcal{T} = (N, L)$, where N is the set of components and $L \subseteq N \times N$ represents their communication links, and a set of natural-language system requirements $\mathcal{R}$, synthesize a collection of local reactive controllers $\mathbf{K} = \{K_v \mid v \in N\}$ whose composition satisfies $\mathcal{R}$ under every environment behavior consistent with its assumptions.

3.1 Agentic Contract-Based Synthesis

Translator. The Translator agent converts each requirement into an LTL formula over a Boolean vocabulary shared with the execution environment

of the simulator. Requirement phrasings follow NL2LTL-style (Fuggitti and Chakraborti, 2023) patterns; the resulting formulas are passed to the contract decomposition and subsequent verification and synthesis steps.

Decomposer. The Decomposer agent structures the set of formulas into a set of per-component A/G contracts (examples shown in Appendix C). A contract $C = (V, A, G)$ is defined over a component’s input and output variables V and consists of a pair of LTL formulas in the generalized reactivity(1) (GR(1)) fragment: the *assumptions* A on the environment and the *guarantees* G that the component must deliver whenever A holds (Benveniste et al., 2018; Nuzzo et al., 2018). The agent identifies the components the requirements pertain to (e.g., robots, arbiter), assigns every Boolean variable as an input or output to each owner, classifies each formula as assumptions or guarantees, and allocates end-to-end objectives to a system-level contract. For a single-component task this reduces to deriving one controller contract from the system-level requirements. For multi-component tasks, the composition of the component-level contracts must additionally refine the system-level contract in terms of the contract algebra (Benveniste et al., 2018; Nuzzo et al., 2018).

Verification. The contract syntax is checked first (Appendix B), and well-formed contract drafts undergo semantic checks (Benveniste et al., 2018; Nuzzo et al., 2018). *Consistency* requires that a contract’s guarantees be satisfiable and *compatibility* that its assumptions be satisfiable, so that at least a valid implementation and environment trace exist. *Refinement* requires that the composition of the component-level contracts refine the system-level contract. Moreover, additional checks are needed to ensure that such a composition is also compatible and consistent. All these checks reduce to satisfiability and validity queries over LTL and can be discharged by a model checker. A failed consistency or compatibility check reports that a contract cannot be satisfied by any component or environment, respectively. A failed refinement check returns a counterexample trace showing which of the two conditions failed: either the composition’s guarantees do not imply the system-level guarantees, or the composition makes stronger assumptions than the system-level contract, excluding environments that the system permits (Xiao et al., 2024; Xiao and Nuzzo, 2026). Each contract’s LTL formulas

are then evaluated on the counterexample trace and the falsified formulas are used to identify candidate contracts for revision.

Synthesis. Each verified contract, expressed using the GR(1) fragment of LTL defines a two-player game between a controller and its environment (Piterman et al., 2006). Solving the game either extracts an FSM strategy that is correct under every valid environment (i.e., an environment satisfying the assumptions), or reports that the contract is unrealizable. In the latter case, the synthesizer returns an *unrealizable core*, a minimal subset of guarantee formulas whose conjunction remains unrealizable under all the assumptions (Könighofer et al., 2009). Overall, each violation has its own repair path (red in Figure 1), as the syntax check returns the location of the syntax error, verification returns counterexample traces, and synthesis returns unrealizable cores. Each of these paths flows back to the Decomposer as a structured repair prompt, and a new episode is initiated until all the checks pass.

3.2 Across-Episode Self-Evolution

Contract decomposition is an iterative process and often requires repair. However, the same failure modes (refinement violations, unrealizable cores) tend to recur across the synthesis episodes of a task. After each episode, ACIES emits a structured log $\mathcal{L} = \{(r_i, d_i)\}_{i=1}^n$, where i is the episode index, n is the maximum number of decomposition attempts, $r_i \in \{pass, fail\}$ its outcome, and d_i the diagnostic signals (counterexample traces, unrealizable cores, syntax errors). The Summarizer agent receives $\mathcal{L}$ and produces a *lesson* that is stored in memory and covers (i) *failure pattern*: the observable symptom, including the failed contract check and the recurring pattern in its diagnostic signals; (ii) *root cause*: the contract design decision that caused the failure, such as a missing assumption, an insufficient guarantee, or a task assigned to the wrong component; and (iii) *fix strategy*: a concrete revision rule specifying how future decompositions should avoid the same failure. Appendix D shows an example.

During the iterative decomposition loop, each task retains exactly one lesson in memory. When a lesson already exists, the Summarizer does not append a new entry. Instead, it uses a prompt containing the existing lesson and the new experience to *distill* them into a single revised lesson. This dis-

tillation step bounds memory usage while progressively refining the stored knowledge. The lesson stored for the current task is pre-pended to the Decomposer’s instructions at the start of a subsequent episode of that task, so that the Decomposer can reuse prior verification-grounded experience.

4 Evaluation Results

Setup. ACIES is implemented on the OpenAI Agents Software Development Kit (SDK) (OpenAI, 2025). The formal toolchain comprises the CHASE¹ logics_tool for syntax checking and the nuXmv model checker (Cavada et al., 2014) for consistency, compatibility, and refinement verification. The GR(1) synthesizer Slugs (Ehlers and Raman, 2016) is used to derive the controllers, and a Pygame-based 2D simulator with a control-barrier-function safety filter (Ames et al., 2019) to execute them (Appendix E). We evaluate eight recent LLMs as agent backbones, three open-source and five proprietary. Each configuration is run ten times with identical natural-language inputs, with and without memory (*with* vs. *w/o Lessons*). We report the E2E synthesis latency and the Translator and Decomposer token consumption.

Case studies. We evaluate ACIES on five robotic control synthesis tasks, named by the coordination pattern each exercises (Appendix E). **PickDeliver**, a single-robot warehouse, uses a sequential pick-and-deliver loop with no coordination; **Patrol** tests event-driven reactivity, where asynchronous alerts interrupt the robot’s patrolling and require inspection before it resumes. Both involve a single controller contract. **Arbitration**, a multi-robot warehouse scenario (Figure 3a), introduces resource contention: robots share shelves and docks over an open-ended task stream, and an arbiter contract must enforce mutual exclusion. **StaticDispatch** and **ReactiveDispatch** include two robots delivering three boxes (Figure 3b), posing a task-allocation problem: the robots must complete all the deliveries and terminate under a fixed round-robin dispatch policy or under dynamic assignment at runtime based on robot availability.

4.1 Memory Across Tasks

Figure 2 (left) compares the five tasks with GPT-5.5 as a backbone. The memory reduces both the latency and token consumption on every task, and

¹<https://chase-cps.github.io>

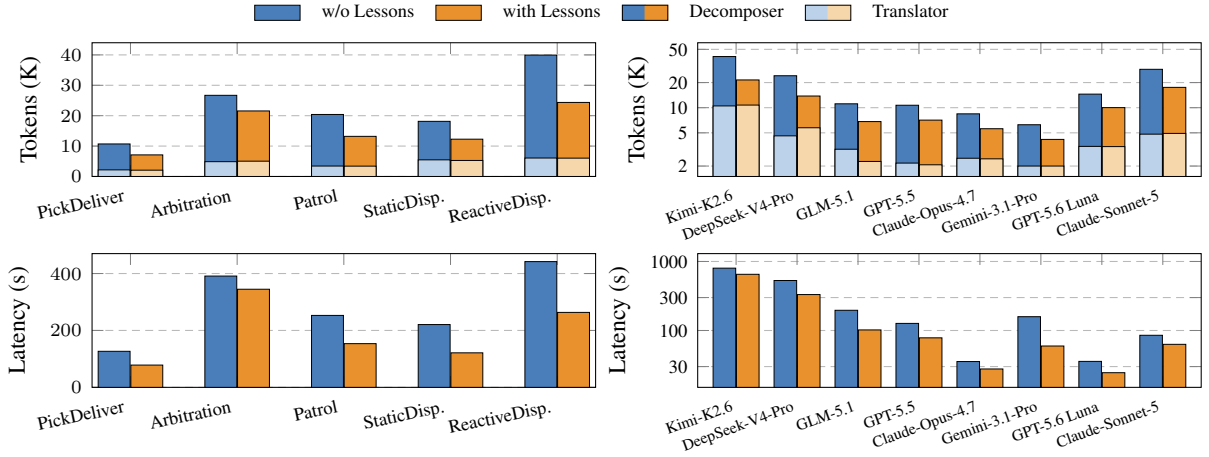


Figure 2: Token consumption (top) and E2E latency (bottom), with and without Lessons, for the five tasks with GPT-5.5 (left) and eight backbones on PickDeliver (right).

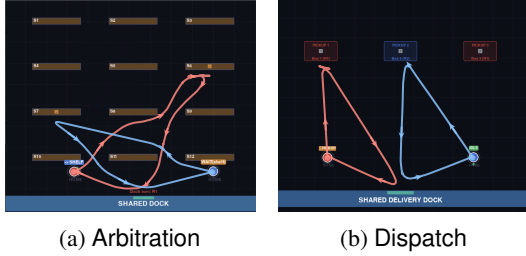


Figure 3: Multi-robot scenarios in the simulation environment: shared-resource arbitration (left) and task dispatch (right).

the benefit grows with the complexity of the coordination mechanism required by the task. On **ReactiveDispatch**, the Decomposer token consumption drops by 46% and the latency by 40%. The savings mostly concentrate in the Decomposer, which uses the recalled lessons to produce accurate decompositions and avoid repeated repairs. The Translator share stays flat, since the lessons mostly impact contract decomposition. These gains come from the lesson distilled on earlier episodes of the same task. In a leave-one-task-out control, we supply each task only with lessons learned from the other tasks (Appendix F); this leaves the success rate largely unchanged while increasing latency.

4.2 Memory Across Models

Figure 2 (right) sweeps all the eight backbones on **PickDeliver**, the least expensive task. The memory reduces the latency and token consumption for every model, cutting the Decomposer tokens by up to 65% (Kimi-K2.6) and the latency by up to 62% (Gemini-3.1-Pro). While the choice of the LLM backbone affects the generation cost and the num-

ber of repair rounds, all the deployed controllers are subject to the same verification criteria.

5 Conclusion

We presented **ACIES**, a self-evolving agentic contract-based design framework that turns natural-language system requirements into verified contracts and controllers. Across five control synthesis tasks and eight LLM backbones, we show that the synthesized controllers realize the formalized requirements. Moreover, the ability to recall lessons learnt from previous designs substantially cuts the overall synthesis costs. Future work includes investigating the scalability of the framework to larger designs via hierarchical contract architectures, the integration of design-time with runtime verification methods, and transferable lesson representations that enable cross-task memory reuse.

Limitations

This study has the following limitations: (1) *Scalability*: The tasks involve at most two robots and rely on centralized arbitration, whose state space grows exponentially with the number of robots. (2) *Domain coverage*: All five tasks come from one domain (discrete planning for warehouse-style robotics); broader coverage of cyber-physical systems remains to be demonstrated. (3) *Formalization fidelity*: Correctness is assessed against the formalized requirements; more analysis is needed to ensure the formalization aligns with the designer’s intent. (4) *Generality*: Lessons are distilled per task; reusing them across tasks showed no clear advantage (Appendix F), and cross-task generalization remains to be explored.

Acknowledgments

This work was supported in part by the National Science Foundation under awards 2514748 and 2448886, and by the Google Cloud Research Credits program.

References

- Aaron D. Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. 2019. Control barrier functions: Theory and applications. In *European Control Conference (ECC)*, pages 3420–3431.
- Anthropic. 2026a. Introducing Claude Opus 4.7. <https://www.anthropic.com/news/claude-opus-4-7>. Accessed: 2026-05-02.
- Anthropic. 2026b. Introducing Claude Sonnet 5. <https://www.anthropic.com/news/claude-sonnet-5>. Accessed: 2026-09-14.
- Amir Bayat, Necmiye Ozay, Alessandro Abate, and Raphael M. Jungers. 2026. End-to-end abstraction-based control with LLM-enhanced NL-to-LTL translation. *arXiv preprint arXiv:2606.30163*.
- Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. 2018. Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400.
- Andrea Brunello, Angelo Montanari, and Mark Reynolds. 2019. Synthesis of LTL formulas from natural language texts: State of the art and research directions. In *26th International Symposium on Temporal Representation and Reasoning (TIME 2019)*, pages 17–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer.
- Yongchao Chen, Jacob Arkin, Charles Dawson, Yang Zhang, Nicholas Roy, and Chuchu Fan. 2024. AutoTAMP: Autoregressive task and motion planning with LLMs as translators and checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.
- Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. 2023. NL2TL: Transforming natural languages to temporal logics using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15880–15903.
- Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer.
- DeepSeek. 2026. DeepSeek V4 release announcement. <https://api-docs.deepseek.com/news/news260424>. Accessed: 2026-05-02.
- Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) synthesis. In *Computer Aided Verification (CAV)*, volume 9780 of *Lecture Notes in Computer Science*, pages 333–339. Springer.
- William H English, Chase Walker, Dominic Simon, Sumit Kumar Jha, and Rickard Ewetz. 2025. Verifiable natural language to linear temporal logic (LTL) translation: A benchmark dataset and evaluation suite. *arXiv preprint arXiv:2507.00877*.
- Francesco Fuggitti and Tathagata Chakraborti. 2023. NL2LTL – a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In *AAAI. System Demonstration*.
- Samuele Germiniani, Daniele Nicoletti, and Graziano Pravadelli. 2025. A systematic literature review on mining LTL specifications. *IEEE Access*.
- Fabian Gloeckle, Mantas Baksys, Darius Feher, Kunhao Zheng, Amaury Hayat, Sean B. Holden, Gabriel Synnaeve, and Peter O’Hearn. 2026. WybeCoder: Verified imperative code generation. *arXiv preprint arXiv:2603.29088*.
- Google DeepMind. 2026. Gemini 3.1 Pro: A smarter model for your most complex tasks. <https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/>. Accessed: 2026-05-02.
- Robert Könighofer, Georg Hofferek, and Roderick Bloem. 2009. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.
- Jason Xinyu Liu, Ziyi Yang, Benjamin Schornstein, Sam Liang, Ifrah Idrees, Stefanie Tellex, and Ankit Shah. 2022. Lang2LTL: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*.
- Moonshot AI. 2026. Kimi K2.6: Advancing open-source coding. <https://www.kimi.com/blog/kimi-k2-6>. Accessed: 2026-05-02.

- Pierluigi Nuzzo, Michele Lora, Yishai A Feldman, and Alberto L Sangiovanni-Vincentelli. 2018. CHASE: Contract-based requirement engineering for cyber-physical system design. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 839–844. IEEE.
- OpenAI. 2025. OpenAI Agents SDK for Python. <https://github.com/openai/openai-agents-python>. Accessed: 2026-04-21.
- OpenAI. 2026a. GPT-5.6: Frontier intelligence that scales with your ambition. <https://openai.com/index/gpt-5-6/>. Accessed: 2026-09-14.
- OpenAI. 2026b. Introducing GPT-5.5. <https://openai.com/index/introducing-gpt-5-5/>. Accessed: 2026-05-02.
- Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. 2025. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. 2006. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer.
- Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57. IEEE.
- Pygame. 2026. Pygame. <https://github.com/pygame/pygame>. Accessed: 2026-09-09.
- Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. 2012. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 18(3):217–238.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.
- Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*, volume 35.
- Yifeng Xiao and Pierluigi Nuzzo. 2026. Contract-based architecture exploration of cyber-physical systems via satisfiability modulo convex programming. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.
- Yifeng Xiao, Chanwook Oh, Michele Lora, and Pierluigi Nuzzo. 2024. Efficient exploration of cyber-physical system architectures using contracts and subgraph isomorphism. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.
- Yilongfei Xu, Jincao Feng, and Weikai Miao. 2024. Learning from failures: Translation of natural language requirements into linear temporal logic with large language models. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, pages 204–215. IEEE.
- Z.ai. 2026. GLM-5.1: Overview and technical report. <https://z.ai/blog/glm-5.1>. Accessed: 2026-05-09.
- Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38.

Appendix

A Preliminaries

Linear temporal logic. LTL (Pnueli, 1977) specifies how system properties evolve along an execution trace $q_0, q_1, q_2, \dots$, where q_i is the state at step i . It extends propositional logic with the temporal operators **X** (*next*), **F** (*eventually*), **G** (*globally*), and **U** (*until*), which together allow expressing safety and liveness properties of practical interest (Germiniani et al., 2025).

Assume-guarantee contracts. In contract-based design (Nuzzo et al., 2018), a component M is described by a contract $C = (V, A, G)$ including a set of variables V , a set of assumptions A on the environment, and a set of guarantees G that M provides when the assumptions hold. In this paper, assumptions and guarantees are all specified in LTL. Specifically, A and G are expressed using lists of assumption and guarantee formulas, respectively, that need to be conjoined. M satisfies C if it satisfies $A \rightarrow G$. *Saturation* replaces G by $A \rightarrow G$, providing a “canonical” representation for the contract guarantees. *Composition* combines component-level contracts into a contract for the composite system, which is then checked to refine the system-level contract. *Refinement* $C \preceq C'$ holds if and only if C has weaker assumptions and stronger guarantees than C' , which for saturated contracts reduces to checking $A' \rightarrow A$ and $G \rightarrow G'$ hold.

B Example Contract

Figure 4 shows an A/G contract for a robot, including typed input/output variables, and environment assumptions and system guarantees in LTL. Robot1 assumes that once it requests a shelf/dock, it will eventually be granted access, and commits to making progress (pickup/dropoff, etc.) once granted. The Arbiter’s guarantees enforce mutual exclusion between the two robots’ grants and never grant an unrequested resource, thereby discharging Robot1’s assumption. The System contract guarantees that each robot eventually completes its task, with mutual exclusion preserved throughout.

C Example Contract Decomposition

Figure 5 shows representative excerpts of the contracts the Decomposer produces for Arbitration. The robot contracts specify the execution of the

```
1  CONTRACT Robot:
2    input boolean variable at_shelf;
3    input boolean variable has_item;
4    input boolean variable task;
5
6    output boolean variable go_to_shelf;
7    output boolean variable pickup;
8    output boolean variable done;
9
10   // Environment assumptions
11   Assumptions:
12     G F(task);
13     G(go_to_shelf -> F(at_shelf));
14     G(at_shelf -> F(has_item));
15
16   // System guarantees
17   Guarantees:
18     !go_to_shelf;
19     G(task -> F(go_to_shelf));
20     G(at_shelf -> F(pickup));
21     G(has_item -> F(done));
22     G F(done);
```

Figure 4: Example robot contract.

delivery task and the request-grant interaction with the environment; the arbiter contract enforces mutual exclusion over shared resources; and the system-level contract defines the global objectives.

D Example Memory Lesson

Figure 6 shows the lesson the Summarizer distilled for Arbitration. It records the symptoms of a failure, the root causes, and reusable repair rules in the three-part form in which lessons are stored in memory and pre-pended to the Decomposer’s prompt instructions in the next episode; the wording is lightly edited for readability.

E Simulation Environments

A real-time 2D simulator built on *Pygame* (Pygame, 2026) loads the synthesized GR(1) strategies as executable controller FSMs. At every tick, each component maps its continuous state, including its position, whether it is carrying an item, and its distance to a shelf or dock, to a Boolean environment vector, which is then fed to its controller FSM to determine the next action. A control-barrier-function-based safety filter (Ames et al., 2019) converts the resulting motion objective into a collision-free velocity command. Multiple robots execute this loop independently and interact through the shared world state. The multi-agent environments are shown in Figure 3. The evaluated backbones are DeepSeek-V4-Pro (DeepSeek, 2026), Kimi-K2.6 (Moonshot AI, 2026), and GLM-5.1 (Z.ai,

<pre> CONTRACT Robot1: input boolean variable ...; output boolean variable ...; Assumptions: G((r1_go_to_shelf & r1_shelf_grant) -> F(r1_at_shelf)); G(r1_pickup -> F(r1_has_item)); Guarantees: G((r1_task & !r1_has_item) -> F(r1_shelf_request)); G(r1_at_shelf -> F(r1_pickup)); G(r1_has_item -> F(r1_dock_request)); G(r1_task -> F(r1_done)); </pre>	<pre> CONTRACT Arbiter: input boolean variable ...; output boolean variable ...; Assumptions: G((r1_shelf_request & !r1_shelf_grant) -> X(r1_shelf_request)); G((r2_shelf_request & !r2_shelf_grant) -> X(r2_shelf_request)); Guarantees: G(!r1_shelf_grant & r2_shelf_grant); G(!r1_dock_grant & r2_dock_grant); G(r1_shelf_grant -> r1_shelf_request); </pre>	<pre> CONTRACT System: input boolean variable ...; output boolean variable ...; Assumptions: G(r1_shelf_request -> F(r1_shelf_grant)); G(r2_shelf_request -> F(r2_shelf_grant)); Guarantees: G(r1_task -> F(r1_done)); G(r2_task -> F(r2_done)); G(F(r1_done)); G(F(r2_done)); G(!r1_shelf_grant & r2_shelf_grant); G(!r1_dock_grant & r2_dock_grant); </pre>
(a) Robot1 contract	(b) Arbiter contract	(c) System contract

Figure 5: Excerpts of the contracts generated by the Decomposer for Arbitration.

<pre> LESSON Arbitration: FAILURE PATTERN: - Refinement check failed although syntax and compatibility passed. - Counterexample traces showed robot actions (pickup, dropoff) occurring without a prior request and grant. ROOT CAUSE: - The system-level guarantees were stronger than what the composition of the component contracts implies. - Rules on variables controlled by another component (requests, grants) were written as assumptions, so the robot and arbiter contracts each assumed the other's behavior. </pre>	<pre> FIX STRATEGY: - Each component guarantees only the variables it controls: robots guarantee actions and requests; the arbiter guarantees grants. - State protocol rules as guarantees of the controlling component, not as assumptions of its peers. - Add action preconditions: G(pickup -> at_shelf); G(dropoff -> (at_dock & has_item)); - Require request and grant before moving: G(go_to_shelf -> shelf_request); G(go_to_shelf -> shelf_grant); </pre>
---	---

Figure 6: Memory lesson distilled by the Summarizer for Arbitration.

2026) (open-source), and Claude-Opus-4.7 (Anthropic, 2026a), Gemini-3.1-Pro (Google DeepMind, 2026), and GPT-5.5 (OpenAI, 2026b) (proprietary), as well as the newer GPT-5.6 Luna (OpenAI, 2026a) and Claude Sonnet 5 (Anthropic, 2026b) (proprietary).

F Cross-Task Transfer Control

We test whether lessons transfer across tasks by running a leave-one-task-out control with GPT-5.6 Luna; the w/o Lessons baselines therefore differ from the GPT-5.5 runs of Figure 2 (left). For each

target task, the lesson prefix pre-pended to the Decomposer’s instructions (Section 3.2) contains one lesson from each of the other four tasks, kept fixed across runs, and excludes the target task’s lesson. Table 1 compares ten such runs with the corresponding w/o Lessons runs. Success rate is unchanged for three tasks, improves only for StaticDispatch, and decreases for ReactiveDispatch. On average across tasks, E2E latency increases by 36.9%.

These prefixes, built from the other four tasks, are $3\text{--}7\times$ longer than a single-task lesson, so each

Task	Success Rate	E2E (s)
	(w/o → cross)	(w/o → cross)
PickDeliver	10/10 → 10/10	35.7 → 76.4
Arbitration	2/10 → 2/10	218.4 → 286.8
Patrol	10/10 → 10/10	54.4 → 85.3
StaticDispatch	9/10 → 10/10	92.9 → 95.8
ReactiveDispatch	9/10 → 7/10	125.7 → 177.4

Table 1: Leave-one-task-out cross-task memory control with GPT-5.6 Luna. Each target receives only the other four fixed lessons; each condition has ten runs. E2E entries are means.

call that includes them must process substantially more input. In addition, they contain task-specific constraints (e.g., variable names) that the prompt marks as mandatory. Among successful runs, the average number of decomposition attempts rises from 3.33 to 3.65 for three out of five tasks. The longer per-call context and additional repair calls together are consistent with the observed latency overhead. These results indicate that lessons are task-specific: reused across tasks, they add context-processing overhead without improving the success rate, whereas a lesson mainly benefits the task it was distilled from. Establishing cross-task transfer will require retrieving lessons by task similarity and abstracting task-specific constraints into transferable rules, which we leave for future work.