AGENTIC CONTRACT-BASED REACTIVE SYNTHESIS FOR MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly being used for natural language (NL)-guided robot control, promising to help solve challenging design problems, such as automatically decomposing tasks and generating plans for multi-robot systems. However, existing LLM-based pipelines are mostly limited to either proposing and testing promising task decompositions, or performing verification of simple, single-agent controllers and plans. Rigorous verification layers that can ensure the correctness of multi-agent behaviors at scale in LLM-aided reactive system synthesis have been elusive. In this paper, we present an agentic *compositional* reactive synthesis framework for multi-agent systems. Given a system architecture and NL requirements, a *Translator* converts the requirements into linear temporal logic (LTL) specifications, while a *Decomposer* further structures them into a system-level assume-guarantee contract and a set of component-level contracts. The Decomposer is connected in a loop with a *contract-based verification* layer that performs a set of checks, including whether the generated contracts match the translated specification, whether the composition of component-level contracts refines the system contract, and whether each component-level contract is realizable by a reactive controller. Each failed check returns a traceable witness that localizes the responsible contracts for automatic rule-based or LLM-guided repair, until all checks pass. Our verification layer also includes the generation of runtime monitors from contracts, which can be used to further validate the synthesized controllers in their operational environment in simulation. Evaluation on three tasks motivated by multi-robot coordination problems in accessing limited resources shows the effectiveness of combining LLM guidance with hierarchical contract-based decomposition towards scalable and verified generation of multi-agent controllers from NL requirements.

1 INTRODUCTION

Large language models (LLMs) are increasingly being used for natural language (NL)-guided robot control, promising to help solve challenging design problems, such as automatically decomposing tasks and generating plans for multi-robot systems. Consider, for example, a team of robots aiming to locally make progress toward performing a task while respecting global constraints on shared resources. A natural-language requirement can state the desired team behavior compactly, e.g., "all robots shall keep delivering parts, but only one may occupy the dock at a time," by capturing the mission goal and a mutual exclusion constraint. Yet, such a requirement leaves the component-level coordination interfaces needed to realize the desired behavior implicit. Generating correct controllers for this mission will also require preserving exclusive access to the shared resource for every admissible interleaving of requests, while ensuring that neither robot is left in starvation.

Prior work addresses distinct stages of language-driven multi-robot control. SMART-LLM (Kannan et al., 2024) uses language models to propose viable task decompositions, coalition formations, and task allocations. However, it is not geared at providing formal guarantees on the correctness of the decompositions. Existing natural language (NL)-to-linear temporal logic (LTL) translation work (Bayat et al., 2026; Chen et al., 2023b) focuses on converting informal instructions to formal specifications. Reactive synthesis (Bloem et al., 2012; Kress-Gazit et al., 2009) can then be used to generate controllers from formal specifications. However, this pipeline cannot be directly applied to multi-agent control synthesis at scale. Contract-based design can offer effective mechanisms

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

for compositional reasoning about system behaviors (Benveniste et al., 2018; Nuzzo et al., 2018), including methods for provably correct stepwise refinement of a system-level assume-guarantee (A/G) contract to per-agent contracts. However, while contracts allow formally checking that the agent-level contracts are compatible and refine a system-level contract, devising an appropriate decomposition in the first place is a challenging task, usually performed manually based on heuristics and the designer's knowledge about the system. In this context, a fully automated framework would ideally be able to propose a promising decomposition of a system-level task specification into interdependent component-level A/G contracts and establish that the decomposition preserves the system requirements. If this is not the case, it would be able to identify which contract caused a violation and use this information to perform a targeted repair of the proposed decomposition.

This paper introduces an agentic *compositional* reactive synthesis framework for multi-agent systems that addresses the above gaps. Given a system architecture and NL requirements, an LLM-based *Translator* converts the requirements into linear temporal logic (LTL) specifications, while a *Decomposer* further structures them into a system-level A/G contract and a set of component-level contracts. The Decomposer is connected in a loop with a *contract-based verification* layer that performs a set of checks, including whether the generated contracts match the translated specification, whether the composition of component-level contracts refines the system contract, and whether each component-level contract is realizable by a reactive controller. Each failed check returns a traceable witness that localizes the responsible contracts for automatic rule-based or LLM-guided repair, until all checks pass. Our verification layer also includes the generation of runtime monitors from contracts, which can be used to further validate the synthesized controllers in their operational environment in simulation. Our contributions can be summarized as follows:

- An end-to-end agentic framework consisting of an LLM-based Translator and a Decomposer closed in a loop with a contract-based verification toolchain that turns NL requirements into verified component-level controllers and runtime monitors for multi-agent system design. Our framework combines NL-to-LTL translation, hierarchical contract decomposition, contract refinement checking, and reactive synthesis.
- A verification and repair loop in which every failed check returns a counterexample that allows identifying the responsible contracts and LTL formulas, so that they can be repaired by a rule-based or an LLM-guided procedure.

We evaluate our approach on three systems, in which multiple robots need to coordinate to access limited resources. We achieve verified generation of multi-agent controllers with no resource conflict from NL requirements in 83% of the experiments. Moreover, combining LLM guidance with hierarchical contract-based decomposition and reuse of verification artifacts from past iterations allows effectively addressing synthesis and validation problems with up to 60 robots.

2 PRELIMINARIES

Linear Temporal Logic and GR(1) Fragment. Linear Temporal Logic (LTL) (Pnueli, 1977) extends propositional logic to reason over infinite sequences of propositional interpretations, or *traces*. Let $\mathcal{AP}$ be a finite set of atomic propositions (APs). The class of LTL formulae over $\mathcal{AP}$ is defined by the grammar $\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \bigcirc\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi \,\mathcal{U}\, \varphi'$, where $P \in \mathcal{AP}$, with temporal operators $\bigcirc$ (next), $\Diamond$ (eventually), $\Box$ (always), and $\mathcal{U}$ (until) (De Giacomo & Vardi, 2013; Bauer et al., 2011; Leucker & Schallhart, 2009).

GR(1) (Generalized Reactivity(1)) is a fragment of LTL widely used for the synthesis of reactive systems (Bloem et al., 2012; Maoz & Ringert, 2015), and we synthesize controllers in this fragment. Let $\mathcal{AP} = \mathcal{X} \uplus \mathcal{Y}$ split into environment propositions $\mathcal{X}$ and system propositions $\mathcal{Y}$. A generalized reactivity of rank 1 (GR(1)) specification is an implication $\varphi^e \rightarrow \varphi^s$ from environment assumptions to system guarantees, i.e., for $\alpha \in \{e, s\}$,

$$\varphi^\alpha = \varphi^\alpha_{\text{init}} \wedge \bigwedge_i \Box\, \varphi^\alpha_{1,i} \wedge \bigwedge_i \Box\Diamond\, \varphi^\alpha_{2,i}, \tag{1}$$

where $\varphi^\alpha_{\text{init}}$ is a propositional formula characterizing the initial conditions, $\varphi^\alpha_{1,i}$ are transition relations characterizing the allowable moves (safety conditions), and $\varphi^\alpha_{2,i}$ are propositional formulas characterizing states that should be attained infinitely often (liveness conditions). A *strategy* M is a function that, given the trace so far and the environment's next valuation of $\mathcal{X}$, picks the next valuation of $\mathcal{Y}$; the traces M generates against every environment from its behaviors, denoted $[[M]]$. The

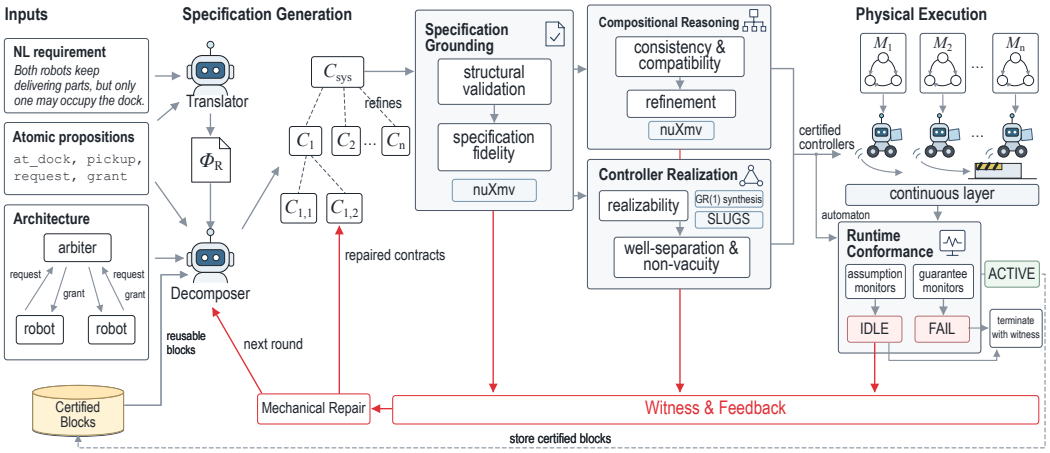


Figure 1 The proposed framework for multi-agent controller synthesis.

specification is *realizable* if some strategy satisfies it on all its traces, and GR(1) synthesis finds one in polynomial time in the number of valuations of $\mathcal{AP}$ (Bloem et al., 2012).

Assume-Guarantee (A/G) Contracts. Let M denote a component, i.e., an element of a system, characterized by a set of variables V_M and a set of behaviors $[[M]]$ over V_M. A strategy is a component whose variables are $\mathcal{AP}$ and whose behaviors are the traces it generates. A contract $C = (V_M, A, G)$ specifies two sets of behaviors over V_M: the assumptions A on the environment of M, and the guarantees G provided by M given that the assumptions are satisfied. M is a valid *implementation* of C, i.e., $M \models C$, if all its behaviors are included in the guarantees given the assumptions, $[[M]] \subseteq G \cup \overline{A}$. A component E is a valid *environment* of C if $[[E]] \subseteq A$. A contract is *consistent* if it admits a valid implementation, i.e., $G \cup \overline{A} \neq \emptyset$, and *compatible* if it admits a valid environment, i.e., $A \neq \emptyset$. The *refinement* relation supports reasoning about contract replaceability: a contract C' can be replaced by C when C refines C', written $C \preceq C'$, i.e., C has weaker assumptions and stronger guarantees. *Composition* ($\otimes$) builds a system-level contract out of multiple component-level contracts (Benveniste et al., 2018).

Architecture Modeling and Problem Formulation. We formalize the multi-agent coordination problem using assume-guarantee (A/G) contracts within the GR(1) fragment of LTL. Each component M_i has inputs I_i, outputs O_i, and internal variables X_i, with $V_{M_i} = I_i \cup O_i \cup X_i$ and disjoint output ownership. In our case studies, robots execute tasks and arbiters coordinate shared resources.

Definition 1 (System Architecture). *An architecture $\mathcal{T} = (\mathcal{M}, \mathcal{E})$ specifies $N = |\mathcal{M}|$ components and directed communication links $\mathcal{E} \subseteq \mathcal{M} \times \mathcal{M}$, determining which component outputs are available as inputs to others.*

Problem 1 (Reactive Contract Decomposition). *Given a natural-language requirement R, an architecture $\mathcal{T}$, and port vocabulary $\mathcal{AP} = \bigcup_i (I_i \cup O_i)$, the problem is to produce a system contract C_{sys} preserving the translated specification Φ_R, local GR(1) contracts C_k, reactive implementations M_k with state sets Q_k that respect the component interfaces, and a runtime monitor for each C_k, such that $\bigotimes_{k=1}^{N} C_k \preceq C_{\text{sys}}$ and $M_k \models C_k$ for every k. These relations ensure Φ_R on every execution whose environment satisfies the contract assumptions, and the monitors decide at run time whether it does (Section 3.6).*

3 Agentic Contract-Based Framework

Given tasks in NL requirements, a set of AP, and the system architecture, the proposed framework produces component-level controller strategies that complete the specified tasks, along with monitors for runtime verification, as shown in Fig. 1. We detail the framework's stages as follows.

3.1 Contract Formalization and Task Decomposition

We translate input requirements into LTL specifications and construct a hierarchy of contracts through two large language model (LLM) agents: the Translator and the Decomposer.

Translator. The Translator converts the NL requirement R into an LTL specification $\Phi_R = \bigwedge_{i=1}^{N_R} \varphi_i^R$ over $\mathcal{AP}$. We follow the approach of mapping each requirement sentence to a corresponding LTL template, with propositions in $\mathcal{AP}$ filling the template (Fuggitti & Chakraborti, 2023).

Decomposer. Given the LTL specification Φ_R, $\mathcal{AP}$ and the system architecture, the Decomposer produces a hierarchy of contracts, i.e., a tree of contracts from the system level recursively decomposing into component-level contracts $C_k = (V_k, A_k, G_k)$ where $V_k = I_k \cup O_k \cup X_k$. I_k and O_k are provided from $\mathcal{AP}$, with the O_k pairwise disjoint, and X_k are discovered by the Decomposer where the component must track its own progress, such as whether it holds a resource. The prompt provides the contract operations (Benveniste et al., 2018), the shapes that GR(1) synthesis admits (Bloem et al., 2012; Kress-Gazit et al., 2009) and common errors identified in practice (Ma'ayan & Maoz, 2023). The draft is only a candidate: the checks that follow determine whether its contracts are well formed, preserve Φ_R, compose correctly, and admit controllers.

The produced contract hierarchy is checked by a 3-stage verification pipeline that follows. Each pipeline stage performs a specific type of check on the contract hierarchy, with a mechanism to locate violations and a mechanical repair step to correct violations automatically. Unrepairable violations are reported to the Decomposer for correction. Passing the verification pipeline, certified controllers for the multi-agent system and corresponding runtime monitors are produced. The system is certified when all controllers work correctly in execution with the monitoring reports no violations. The Decomposer can also reuse blocks from previous certified systems when sharing the same atomic propositions. The reused blocks can be regarded as a sub-hierarchy of the target contract hierarchy that does not need to be reverified.

3.2 Specification Grounding

After contract generation, the specification grounding stage ensures that the generated contracts are correctly structured and semantically consistent with the system specification.

Structural Validation. Given the generated contracts, $\mathcal{AP}$ and the system architecture, we check that every component-level constraint is in GR(1) form (1), that no assumption constrains the component's outputs O_k or internal variables under the next operator, and that the contracts' variables are free of name conflicts.

Specification Fidelity. Given the generated system contract C_{sys} and the system specification Φ_R, we verify that $\forall \sigma,\ \sigma \models \varphi_{A_{\text{sys}}} \wedge \varphi_{G_{\text{sys}}} \implies \sigma \models \Phi_R$. We search for a counterexample trace $\sigma^{\text{cex}} \models \varphi_{A_{\text{sys}}} \wedge \varphi_{G_{\text{sys}}} \wedge \neg \Phi_R$ and, if one exists, localize the failure by evaluating the counterexample against each requirement and identify the violated requirements as $\Gamma_R(\sigma^{\text{cex}}) = \{\varphi_i^R \mid \sigma^{\text{cex}} \not\models \varphi_i^R\}$. If no such trace exists, the generated system contract preserves the system specification. For a set of constraints Ψ, let $\text{Sat}(\Psi)$ hold iff some trace satisfies every constraint in Ψ. We extract a minimal unsatisfiable core of assumptions $\Gamma_f \subseteq A_{\text{sys}}$:

$$\neg\text{Sat}(\Gamma_f \cup G_{\text{sys}} \cup \{\neg\Phi_R\}) \ \wedge\ \big(\forall \varphi \in \Gamma_f,\ \text{Sat}((\Gamma_f \setminus \{\varphi\}) \cup G_{\text{sys}} \cup \{\neg\Phi_R\})\big). \tag{2}$$

The assumptions not in Γ_f are added by the Decomposer.

3.3 Compositional Reasoning

We check if the composed component-level contracts refine the system contract, so that the system requirements can be satisfied by the multi-agent system. All contract operations below are applied to saturated contracts (Benveniste et al., 2018).

Consistency and Compatibility Checking. Given the component contracts $\{C_k = (A_k, G_k)\}_{k=1}^N$ and the system contract C_{sys}, we check whether the assumptions and guarantees of the system contract and of all the components' contracts can be satisfied simultaneously, ensuring that the architecture contract is both consistent and compatible. This verification problem is $\text{Sat}(\mathcal{F})$ for the set of all constraints $\mathcal{F}$. If it holds, the contracts are consistent and compatible. Otherwise, we extract

a minimal unsatisfiable core $\Gamma_c \subseteq \mathcal{F}$ as in (2),

$$\mathcal{F} = A_{\text{sys}} \cup G_{\text{sys}} \cup \bigcup_{k=1}^{N}(A_k \cup G_k), \qquad \neg\text{Sat}(\Gamma_c) \ \wedge \ \big(\forall \varphi \in \Gamma_c,\ \text{Sat}(\Gamma_c \setminus \{\varphi\})\big), \tag{3}$$

i.e., an irreducible set of assumptions and guarantees that cannot be satisfied together.

Refinement Checking. If all contracts are consistent and compatible, we proceed to verify whether the composite component contract $C_{\text{cmp}} = \otimes_{k=1}^{N} C_k$ refine the system contract. We separately check the assumption and guarantee sides by showing that

$$\varphi_{A_{\text{sys}}} \wedge \neg\varphi_{A_{\text{cmp}}}, \qquad \varphi_{G_{\text{cmp}}} \wedge \neg\varphi_{G_{\text{sys}}} \tag{4}$$

are both unsatisfiable, i.e., no trace satisfies either formula (Xiao et al., 2024; Xiao & Nuzzo, 2026b). Otherwise, if $\sigma_a \models \varphi_{A_{\text{sys}}} \wedge \neg\varphi_{A_{\text{cmp}}}$, we localize to the constraint set $\Gamma_a(\sigma_a) = \{\varphi \in A_{\text{cmp}} : \sigma_a \not\models \varphi\}$ that needs to be relaxed to admit σ_a. Similarly, if $\sigma_g \models \varphi_{G_{\text{cmp}}} \wedge \neg\varphi_{G_{\text{sys}}}$, we return $\Gamma_g(\sigma_g) = \{\varphi \in G_{\text{sys}} : \sigma_g \not\models \varphi\}$

3.4 Controller Realization

In parallel, for each component-level contract, we solve a GR(1) synthesis problem to extract a finite-state controller when the contract is realizable, and validate the extracted strategy.

Controller Synthesis. For each $C_k = (A_k, G_k)$, we check whether there exists a strategy π_k such that $\forall \sigma \in [[\pi_k]],\ \sigma \models \varphi_{A_k} \implies \sigma \models \varphi_{G_k}$, where $[[\pi_k]]$ is the set of traces π_k generates against every environment input sequence. A strategy meeting this condition is *winning*. We denote this realizability check as $\text{Real}(A_k, G_k)$. If it holds, we extract a finite-state machine controller M_k implementing π_k (Bloem et al., 2012). Several moves at a controller state may be winning. We prefer to extract a move that asserts an action output whenever one exists, so the controller acts instead of winning vacuously when the environment violates an assumption.

If C_k is unrealizable, the environment has a winning strategy π_k^{cex} which is a finite-state strategy of the environment over I_k such that every trace $\sigma \in [[\pi_k^{\text{cex}}]]$ satisfies $\varphi_{A_k} \wedge \neg\varphi_{G_k}$, whatever outputs the component chooses (Könighofer et al., 2013). Realizability can be restored by relaxing the guarantees or strengthening the assumptions of C_k. On the one hand, we compute a minimal unrealizable core $\Gamma_r \subseteq G_k$ as in (3):

$$\neg\text{Real}(A_k, \Gamma_r) \ \wedge \ \big(\forall \varphi \in \Gamma_r,\ \text{Real}(A_k, \Gamma_r \setminus \{\varphi\})\big). \tag{5}$$

On the other hand, following (Alur et al., 2013), an added assumption must be violated by π_k^{cex}, otherwise the counter-strategy keeps winning. Let $\mathcal{L}(I_k) = I_k \cup \{\neg x : x \in I_k\}$ be the literals over the inputs, and let $\Gamma_h = \{\ell \in \mathcal{L}(I_k) : \forall \sigma \in [[\pi_k^{\text{cex}}]],\ \sigma \models \Box\ell\}$ be the literals that the counter-strategy holds fixed on every trace. An assumption inconsistent with $\Box\ell$ for some $\ell \in \Gamma_h$ is violated by every trace of π_k^{cex} and therefore excludes the counter-strategy.

Well-Separation and Non-Vacuity. A realizable contract may still admit a winning strategy that satisfies G_k without the component acting. We exclude such contracts with three further realizability checks before a controller is extracted. First, following (Maoz & Ringert, 2016), C_k is *well-separated* if no strategy can force the environment to violate A_k. A strategy forces such a violation exactly when it wins against the unsatisfiable liveness guarantee $\Box\Diamond\bot$, so we check

$$\neg\text{Real}\big(A_k,\ \text{safe}(G_k) \cup \{\Box\Diamond\bot\}\big), \tag{6}$$

where $\text{safe}(G_k)$ denotes the safety conditions of G_k. Second, following the strengthening criterion of (Bloem et al., 2021), we check that no winning strategy leaves an output unused. Let y be an output of C_k that occurs positively in a liveness condition of G_k, or that C_{sys} requires on every trace, $\neg\text{Sat}(A_{\text{sys}} \cup G_{\text{sys}} \cup \{\Box\neg y\})$. We strengthen G_k in two ways: by requiring that y is never set, and by adding an output p that the component must eventually set, that remains set, and that forbids y once set. We check that neither strengthened contract is realizable, as formalized in Appendix B. If either is realizable, a winning strategy exists that never sets y or that eventually stops setting it, so the liveness condition is satisfied by A_k alone, and we say that y is *vacuous* in C_k.

Either failure is localized on the assumptions: dually to Γ_r, we compute a minimal set of assumptions $\Gamma_v \subseteq A_k$ that keeps the strengthened contract realizable (Appendix B), and a trace of a strategy that realizes it shows how the component wins without acting.

Table 1 The checks of the framework: what each one examines, the decision procedure that settles it, and what a failure returns.

Check	Object	Question	Decided by	Witness	Repair
Structural Validation	C_k	GR(1) form, ports	syntax	constraint, rule	rewrite
Specification Fidelity	C_{sys}	$C_{\text{sys}} \Rightarrow \Phi_R$	Sat	$\sigma^{\text{cex}}, \Gamma_R$	revise C_{sys} †
Consistency, Compatibility	C_k, C_{sys}	(3)	Sat	Γ_c	relax one
Refinement	$\bigotimes C_k, C_{\text{sys}}$	(4)	Sat	$\sigma_a, \Gamma_a; \sigma_g, \Gamma_g$	relax/strengthen†
Realizability	C_k	$\text{Real}(A_k, G_k)$	Real	$\Gamma_r, \pi_k^{\text{cex}}, \Gamma_h$	relax G_k, add A_k †
Well-Separation, Non-Vacuity	C_k	(6), (7)	Real	Γ_v, trace	relax A_k
Runtime Conformance	σ	*IDLE*, *FAIL*	monitors	prefix, constraint	weaken A_k, or terminate

† a mechanical repair (Section 3.5) is attempted first.

3.5 Mechanical Repair

Before a witness returns to the Decomposer, we attempt a repair without the Decomposer. Each repair below consumes the witness of one check, and the repaired contracts re-enter the pipeline.

Specification fidelity. A conjunct of Φ_R that C_{sys} merely omits costs no decomposer round: it is known, and only its placement is not. We add each omitted conjunct to A_{sys} if it constrains only inputs of the system, and to G_{sys} if it constrains one of its outputs, so every conjunct holds by construction and a misplacement is caught downstream.

Refinement. Intuitively, in refinement, every assumption of a component must be implied by the system assumptions together with the guarantees of the other components (Benveniste et al., 2018; Cimatti & Tonetta, 2015). Constraints in $\Gamma_a(\sigma_a)$ are those that neither supplies. Let $\varphi_a \in \Gamma_a(\sigma_a)$ be a constraint of the contract C_i and $\text{var}(\varphi_a)$ denote the set of variables constrained by φ_a. Let $\text{ctrl}(\varphi_a) = \{k \neq i : \text{var}(\varphi_a) \cap O_k \neq \emptyset\}$ be the other components that control a variable of φ_a. If $\text{ctrl}(\varphi_a) = \emptyset$, φ_a is an assumption on the system environment, and we propagate it to the system contract, which becomes $(A_{\text{sys}} \cup \{\varphi_a\},\ G_{\text{sys}})$. If $\text{ctrl}(\varphi_a) = \{j\}$ and $\text{var}(\varphi_a) \subseteq V_j$, only C_j can provide φ_a, and we add it to the guarantees of C_j, so that $C_j' = (A_j,\ G_j \cup \{\varphi_a\})$. In both cases σ_a falsifies φ_a, which removes it from the traces that (4) satisfies. A constraint with $|\text{ctrl}(\varphi_a)| \geq 2$ or with $\text{var}(\varphi_a) \not\subseteq V_j$ returns to the Decomposer.

Realizability. We instantiate candidate assumptions from two templates, i.e., *justice* and *persistence*, in the GR(1) families of (Li et al., 2011):

$$\square\lozenge\neg\ell,\ \ell \in \Gamma_h, \qquad \square\big((x \wedge y) \rightarrow \bigcirc x\big),\ x \in I_k \cap \text{var}(\Gamma_r),\ y \in O_k,$$

where $\text{var}(\Gamma_r)$ denotes the variables constrained by Γ_r. A justice candidate requires ℓ to be falsified infinitely often, so it excludes the counter-strategy. A persistence candidate applies when one guarantee in Γ_r keeps an output y asserted across several steps and another, such as $\square(y \rightarrow x)$, requires x while y holds. The two are unsatisfiable together if x drops during those steps, and the candidate assumes that it does not. A candidate φ_a repairs the contract to $C_k' = (A_k \cup \{\varphi_a\}, G_k)$.

3.6 Runtime Conformance

When all checks pass, the validated controllers M_k are deployed for execution in the simulators.

Physical Execution. Each state of M_k is implemented by a continuous controller that carries out the task operation selected by the state, and a control-barrier-function filter (Ames et al., 2017; Xiao & Belta, 2022) keeps the robots at a minimum distance from each other and from obstacles. Every proposition of $\mathcal{AP}$ is evaluated on the physical state, and M_k takes a transition when one changes, so $\mathcal{AP}$ is the only connection between the discrete and continuous layers and a validated M_k runs on any workspace in which the same propositions are evaluated.

Contract-Derived Monitors. The executed trace σ is the sequence of valuations of $\mathcal{AP}$ produced at each proposition change. Each constraint in A_k and G_k is monitored over prefixes of σ and reported violated as soon as a violating prefix appears (Bauer et al., 2011), yielding the three-valued contract status (Xiao & Nuzzo, 2026a): *IDLE* if an assumption is violated, *FAIL* if a guarantee is violated while assumptions hold, and *ACTIVE* otherwise. A liveness condition is violated when the required change exceeds its bound T (Kupferman et al., 2007). In *IDLE*, the violated assumption is traced via (4) to A_{sys}; if it lies in Γ_f of (2), the framework terminates with a witness, otherwise it returns

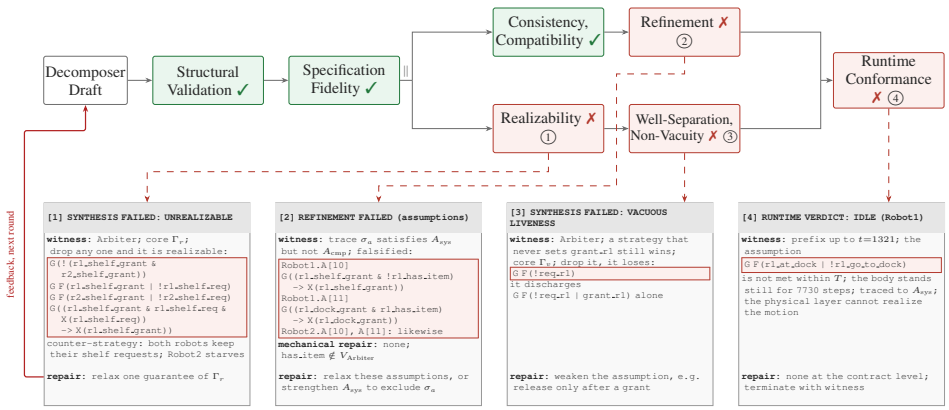


Figure 2 Feedback per failed check of Table 1, with the formulas to change boxed: (1) unrealizable core and counter-strategy, (2) refinement counterexample, (3) vacuous guarantee and assumption core, (4) runtime *IDLE* verdict.

to the Decomposer for weakening. *FAIL* indicates a continuous-layer fault, such as an incorrect proposition evaluation or an unexecuted task operation, and terminates with the witness.

Table 1 summarizes the verification pipeline where Sat checks fidelity, consistency, compatibility, and refinement; Real checks realizability, well-separation, and non-vacuity; monitoring checks whether the environment satisfies the contract assumptions. Fig. 2 illustrates each message type. Witnesses accumulate across rounds, and structural validation keeps rejected constraints from returning. The loop terminates when all checks pass or the round budget is exhausted.

4 Experimental Evaluation

Experimental setup. We use open-source tools throughout: nuXmv (Cavada et al., 2014) and CHASE (Nuzzo et al., 2018) for contract verification, Slugs GR(1) (Ehlers & Raman, 2016) for strategy synthesis, and our control-barrier-function (CBF) simulator for execution. Unless otherwise stated, all LLM components (Translator and Decomposer) use Claude Opus 4.6 (Anthropic, 2026). The repair loop is capped at nine iterations, with exhaustion counted as failure.

Environments. Three simulated worlds are used, and a task is a set of NL requirements and an architecture graph over one of them. The Charging station is an open floor with one charger, where robots sense their position and battery level and only one robot charges at a time. The Single-lane corridor is a wall with one gap between side A and side B, where robots sense their side and the gap holds one robot at a time. The Warehouse has shelves or pickup points and one dock, where robots sense their position and whether they carry an item, and the dock serves one robot at a time. Section 4.2 also uses a warehouse task with a shared shelf, so one arbiter controls two resources.

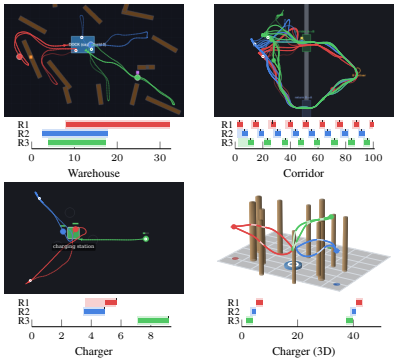
4.1 Verification and Task Completion

We evaluate each environment on a coordination ladder, where each rung adds a robot competing for a capacity-1 resource through an arbiter. We compare three decomposition methods using the same model, requirements, nine-round budget, and simulator. Fig. 3(a) reports the results and Fig. 3(b) shows the certified controllers at run time in the simulators, with non-overlapping request and grant intervals. Rounds gives the minimum–maximum decomposition attempts, while Items gives completed deliveries (Warehouse), crossings (Corridor), or recharges (Charger). No result means that synthesis or certification did not finish within the budget. *No verification* generates GR(1) specifications without checks; *Pass/fail* performs all checks but reports only the failed stage; and *Full* uses the complete pipeline. A run with a result completes the task when its controllers make progress without monitor violations, and is *stalled* when they deliver nothing.

The full pipeline completes 83% of runs, compared with 47% for *No verification* and 43% for *Pass/fail*; its 95% Wilson interval (66–93%) is also separated from theirs (30–64% and 27–61%). *No*

Env.	Method	Task completed	Stalled	Resource conflict	No result	Rounds	Items
Warehouse	No verif.	5/10	5	0	0	1–4	3–24
	Pass/fail	2/10	2	0	6	3	13
	Full	**8/10**	0	0	2	2–4	13–32
Corridor	No verif.	3/10	3	2	2	1–6	9–17
	Pass/fail	1/10	5	1	3	1–5	9–21
	Full	**7/10**	1	0	2	1–4	9–25
Charger	No verif.	6/10	0	0	4	1–5	2–8
	Pass/fail	10/10	0	0	0	1–2	2–10
	Full	**10/10**	0	0	0	1–2	2–10
All	No verif.	47%	27%	7%	20%	1–6	2–24
	Pass/fail	43%	23%	3%	30%	1–5	2–21
	Full	**83%**	3%	0%	13%	1–4	2–32

(a)



(b)

Figure 3 Verification and execution in the three environments. (a) Outcomes over 10 runs per environment and method. (b) Trajectories (top) and request (light) and grant (dark) intervals in seconds (bottom).

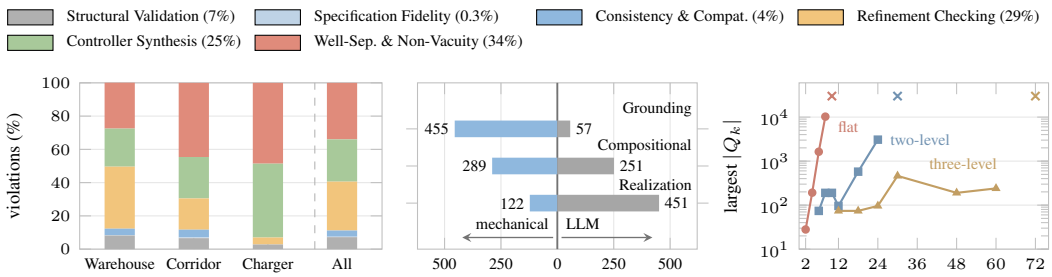
verification often synthesizes realizable but incorrect controllers: 27% of runs deliver nothing and 7% cause resource conflicts, as no interface checks enforce cross-component obligations. *Pass/fail* performs the checks but reports only the failed stage, causing the model to remove failed obligations rather than repair them, which leaves 23% of its runs stalled; 30% of runs exhaust the nine-round budget. *Full* provides both the failing core and trace, prompting targeted repairs. Only 13% of its runs exhaust the budget, and refinement checks prevent resource conflicts. Its sole stalled run exploits a liveness guarantee that the non-vacuity check does not detect. The methods differ most where components interact: on the charger both checked methods complete all ten runs, while on the warehouse and the corridor the full pipeline completes 8 and 7 of 10, against 2 and 1 for Pass/fail. The 3D charger in Fig. 3(b) is a forest water-collection scenario over the same propositions as the 2D charger, so it runs the same controllers with quadrotor dynamics and spatial CBF obstacle avoidance in place of the planar ones.

4.2 Violations and Repairs along the Pipeline

Fig. 4(a) classifies 759 violations in 446 drafts from 90 runs, 30 per environment, decomposed by Claude Sonnet 4.6 under a fixed NL-to-LTL translation. Pooled, non-vacuity catches the most (34%), followed by refinement (29%) and synthesis (25%), while structural, consistency and fidelity errors together account for 12%. The dominant class follows the structure of the task. In the warehouse, one arbiter controls a shelf and a dock, so every robot assumption about either grant must be discharged by an arbiter guarantee and every arbiter assumption about a request by a robot guarantee, and refinement catches 37% of the violations. The corridor and the charger contend for a single resource, the interface is smaller, and the most frequent defect is a contract that can be won without acting (45% and 49%). No single check would serve all three tasks. Fig. 4(b) shows which defects are repaired mechanically, 866 against 759 returned to the model. At grounding the rules repair 455 defects and return 57, since an omitted requirement has one correct placement. At composition the split is close, 289 against 251, since an assumption is propagated only when at most one component can provide it. At realization the rules repair 122 and return 451, since the two assumption templates cover a narrow class of unrealizable contracts and a vacuous contract needs a new design. Refinement is also the one check that no later check replaces: replayed offline on decompositions accepted with refinement disabled, 20 of 22 fail, against 0 of 23 for the full pipeline.

4.3 Scaling and Compositional Reuse

Fig. 4(c) compares three ways to obtain controllers for N robots that share one dock, and Appendix E lists the times. Synthesizing one flat arbiter for all robots works up to $N = 8$, where the largest controller has 10,240 states, and at $N = 10$ synthesis does not finish within 900 s. Reusing a certified three-robot group under a root arbiter carries the architecture to $N = 24$, because only the root is synthesized and it arbitrates groups rather than robots. A third level of arbiters bounds the number of children of each arbiter and carries it to $N = 60$, the largest team we tested, with the



(a) violations by check (b) who repairs (c) largest controller vs. team size

Figure 4 (a) Violations each check catches over 90 runs and 446 drafts, per environment and pooled. (b) Defects repaired mechanically or returned to the LLM. (c) Largest synthesized controller per team size N (horizontal axis), $\times$ marking the first N not finished within 900 s.

largest controller at or below 462 states. In every case the composition is checked against the flat requirement for N robots, so refinement is re-established at every size while the components are obtained once. Richer refinement rules and reuse across tasks would extend this further.

5 Related Work

Language models for robot planning. LLMs have been used for single-robot skill grounding and planning (ichter et al., 2023), multi-robot task allocation (Kannan et al., 2024), hierarchical temporal-logic planning (Xu et al., 2025), and the generation of specifications for motion solvers (Chen et al., 2023a). Another line of work translates natural language into temporal logic (Chen et al., 2023b; Bayat et al., 2026). These approaches produce plans, task structures, or system-level specifications, but do not generate and certify a component-level contract decomposition. Hierarchical finite-trace methods allocate tasks among robots (Xu et al., 2025; Luo et al., 2024; Luo & Liu, 2025); they address a different problem from synthesizing reactive strategies that guarantee recurring progress and mutual exclusion for every admissible environment behavior.

Reactive synthesis, contracts, and verified repair. Reactive synthesis produces controllers that satisfy their specifications against every admissible environment behavior (Bloem et al., 2012; Kress-Gazit et al., 2009), and we use the SLUGS synthesizer (Ehlers & Raman, 2016). A/G contracts support compositional reasoning (Benveniste et al., 2018; Nuzzo et al., 2018). Existing decomposition methods partition specifications (Iannopollo et al., 2018), allocate finite-trace tasks (Luo et al., 2024; Luo & Liu, 2025), or compose per-agent contracts (Liu et al., 2025). Our setting additionally requires generating coupled component interfaces that are not explicit in the team requirement. Well-separatedness (Klein & Pnueli, 2010; Bloem et al., 2021), counter-strategies (Alur et al., 2013), and unrealizable cores (Junker, 2004) provide relevant diagnostics. Related LLM-based systems repair violated assumptions (Meng et al., 2025), verify generated robot programs (Hu et al., 2024), or repair protocols from counterexamples (Xia et al., 2026). In contrast, our loop repairs an LLM-generated contract decomposition and verifies refinement before synthesizing the component controllers. Table 2 in Appendix A provides a detailed comparison.

6 Discussion

We presented a neuro-symbolic framework for the generation of multi-agent reactive controllers from natural-language requirements. Evaluation results show that the framework can generate formally verified controllers that successfully execute the intended task while coordinating access to shared resources. The framework is limited to discrete controllers (planners) obtained by reactive synthesis, and future work includes other contract implementation methods. The translation of the requirement into Φ_R is not certified, so a mistranslated requirement yields a verified controller for the wrong specification. The interface between the discrete controller and the continuous robot model is realized manually. A single arbiter limits scalability as the team grows (Fig. 4(c)), which recursive decomposition can address. We also plan to transfer contract templates and decomposition guidelines across domains and tasks.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, by writing and refactoring parts of the framework and experiment code; to design and provide feedback on the experiments, where the tools collected and organized information and every design decision was made by the authors; and to interpret results, by tracing failed runs to their causes and computing confidence intervals. We have not used generative AI tools to generate synthetic data sets, to develop theoretical models or conceptual frameworks, to formulate mathematical claims, to provide critical ingredients for proving them or assist in writing the proofs, or to propose or refine hypotheses. The formal framework and all mathematical derivations are the authors' own, and translation, dataset cleaning, and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to generate and modify scientific figures, to create and edit software code, to draft and edit parts of the paper for readability, to search for and identify relevant literature, and to format references. We have reviewed all AI-assisted work. AI-assisted code was tested against the unit and end-to-end test suites, and AI-assisted text and figures were checked against the underlying data. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work concerns the correct-by-construction control of robot teams in simulation. It involves no human subjects, personal data, or deployment on physical systems, and we are not aware of direct ethical risks beyond those common to automation of shared resources.

REPRODUCIBILITY STATEMENT

Section 4 lists the verification and synthesis tools, the LLM used for each component, the repair-loop cap, and the three environments. These tools are open source. Appendices B–E give the non-vacuity checks, the coordination-policy comparison, the fairness profiles, and the scaling results. The pipeline, the task specifications, the simulators, and the scripts that produce every table and figure are provided as anonymized supplementary material.

REFERENCES

Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of gr(1) temporal logic specifications. In *2013 Formal Methods in Computer-Aided Design*, pp. 26–33, 2013. doi: 10.1109/FMCAD.2013.6679387.

Aaron D. Ames, Xiangru Xu, Jessy W. Grizzle, and Paulo Tabuada. Control barrier function based quadratic programs for safety critical systems. *IEEE Transactions on Automatic Control*, 62(8): 3861–3876, 2017. doi: 10.1109/TAC.2016.2638961.

Anthropic. Claude opus 4.6. `https://www.anthropic.com/claude/opus`, 2026. Model Card.

Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4), September 2011. ISSN 1049-331X. doi: 10.1145/2000799.2000800. URL `https://doi.org/10.1145/2000799.2000800`.

Amir Bayat, Necmiye Ozay, Alessandro Abate, and Raphael M. Jungers. End-to-end abstraction-based control with LLM-enhanced NL-to-LTL translation, 2026. URL `https://arxiv.org/abs/2606.30163`.

Albert Benveniste, Benoît Caillaud, D. Ničković, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim Guldstrand Larsen. Contracts for system design. *Found. Trends Electron. Des. Autom.*, 12: 124–400, 2018. URL `https://api.semanticscholar.org/CorpusID:13828839`.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, May 2012. ISSN 0022-0000. doi: 10.1016/j.jcss.2011.08.007. URL `https://doi.org/10.1016/j.jcss.2011.08.007`.

Roderick Bloem, Hana Chockler, Masoud Ebrahimi, and Ofer Strichman. Vacuity in synthesis. *Form. Methods Syst. Des.*, 57(3):473–495, September 2021. ISSN 0925-9856. doi: 10.1007/s10703-021-00381-5. URL https://doi.org/10.1007/s10703-021-00381-5.

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pp. 334–342, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 9783319088662. doi: 10.1007/978-3-319-08867-9_22. URL https://doi.org/10.1007/978-3-319-08867-9_22.

Yong-Chao Chen, Jacob Arkin, Yang Zhang, Nicholas A. Roy, and Chuchu Fan. AutoTAMP: Autoregressive task and motion planning with llms as translators and checkers. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6695–6702, 2023a. URL https://api.semanticscholar.org/CorpusID:259138811.

Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. NL2TL: Transforming natural languages to temporal logics using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 15880–15903, 2023b. doi: 10.18653/v1/2023.emnlp-main.985.

Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97:333–348, 2015. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2014.06.011. URL https://www.sciencedirect.com/science/article/pii/S0167642314002901. Object-Oriented Programming and Systems (OOPS 2010) Modeling and Analysis of Compositional Software (papers from EUROMICRO SEAA'12).

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pp. 854–860. AAAI Press, 2013. ISBN 9781577356332.

Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible GR(1) synthesis. In Swarat Chaudhuri and Azadeh Farzan (eds.), *Computer Aided Verification*, pp. 333–339, Cham, 2016. Springer International Publishing. ISBN 978-3-319-41540-6.

Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pp. 234–254, 1979.

Francesco Fuggitti and Tathagata Chakraborti. NL2LTL - a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. ISBN 978-1-57735-880-0. doi: 10.1609/aaai.v37i13.27068. URL https://doi.org/10.1609/aaai.v37i13.27068.

Zichao Hu, Francesca Lucchetti, Claire Schlesinger, Yash Saxena, Anders Freeman, Sadanand Modak, Arjun Guha, and Joydeep Biswas. Deploying and evaluating LLMs to program service mobile robots. *IEEE Robotics and Automation Letters*, 9(3):2853–2860, 2024.

Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. Specification decomposition for synthesis from libraries of LTL assume/guarantee contracts. In *Design, Automation & Test in Europe (DATE)*, pp. 1574–1579, 2018. doi: 10.23919/DATE.2018.8342266.

brian ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes,

Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In Karen Liu, Dana Kulic, and Jeff Ichnowski (eds.), *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Ulrich Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pp. 167–172. AAAI Press, 2004. ISBN 0262511835.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. SMART-LLM: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 12140–12147, 2024. doi: 10.1109/IROS58592.2024.10802322.

Uri Klein and Amir Pnueli. Revisiting synthesis of gr(1) specifications. In *Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing*, HVC'10, pp. 161–181, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 9783642195822.

Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *Int. J. Softw. Tools Technol. Transf.*, 15(5–6):563–583, October 2013. ISSN 1433-2779. doi: 10.1007/s10009-011-0221-y. URL https://doi.org/10.1007/s10009-011-0221-y.

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009. doi: 10.1109/TRO.2009.2030225.

Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. From liveness to promptness. In Werner Damm and Holger Hermanns (eds.), *Computer Aided Verification*, pp. 406–419, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2008.08.004. URL https://www.sciencedirect.com/science/article/pii/S1567832608000775. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, pp. 43–50, 2011. doi: 10.1109/MEMCOD.2011.5970509.

Jonghan Lim, Mostafa Tavakkoli Anbarani, Rômulo Meira-Góes, and Ilya Kovalenko. Logic-based verification of task allocation for llm-enabled multi-agent manufacturing systems. In *2026 18th International Workshop on Discrete Event Systems (WODES)*, pp. 139–144, 2026. doi: 10.1109/WODES69290.2026.11598152.

Siyuan Liu, Adnane Saoud, and Dimos V. Dimarogonas. Controller synthesis of collaborative signal temporal logic tasks for multi-agent systems via assume-guarantee contracts. *IEEE Transactions on Automatic Control*, 2025. doi: 10.1109/TAC.2025.3549288. arXiv:2309.13499.

Xusheng Luo and Changliu Liu. Simultaneous task allocation and planning for multirobots under hierarchical temporal logic specifications. *IEEE Transactions on Robotics*, 41:5040–5059, 2025. doi: 10.1109/TRO.2025.3598139.

Xusheng Luo, Shaojun Xu, Ruixuan Liu, and Changliu Liu. Decomposition-based hierarchical task allocation and planning for multi-robots under hierarchical temporal logic specifications. *IEEE Robotics and Automation Letters*, 9(8):7182–7189, 2024. doi: 10.1109/LRA.2024.3412589.

Dor Ma'ayan and Shahar Maoz. Using reactive synthesis: An end-to-end exploratory case study. In *IEEE/ACM Int. Conf. on Software Engineering (ICSE)*, pp. 742–754, 2023. doi: 10.1109/ICSE48619.2023.00071.

Shahar Maoz and Jan Oliver Ringert. Gr(1) synthesis for LTL specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 96–106, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786824. URL https://doi.org/10.1145/2786805.2786824.

Shahar Maoz and Jan Oliver Ringert. On well-separation of GR(1) specifications. In *ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*, pp. 362–372, 2016. doi: 10.1145/2950290.2950300.

Qian Meng, Jin Peng Zhou, Kilian Q. Weinberger, and Hadas Kress-Gazit. INPROVF: Leveraging large language models to repair high-level robot controllers from assumption violations. In *IEEE Int. Conf. on Automation Science and Engineering (CASE)*, pp. 1680–1687, 2025. arXiv:2503.13660.

Pierluigi Nuzzo, Michele Lora, Yishai A. Feldman, and Alberto L. Sangiovanni-Vincentelli. CHASE: Contract-based requirement engineering for cyber-physical system design. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 839–844, 2018. doi: 10.23919/DATE.2018.8342122.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, 1977. doi: 10.1109/SFCS.1977.32.

Yongdong Wang, Runze Xiao, Jun Younes Louhi Kasahara, et al. DART-LLM: Dependency-aware multi-robot task decomposition and execution using large language models. *arXiv preprint arXiv:2411.09022*, 2024.

Shuren Xia, Qiwei Li, Taqiya Ehsan, and Jorge Ortiz. Tracefix: Repairing agent coordination protocols with tla+ counterexamples. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 181–196, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724152. doi: 10.1145/3786335.3813159. URL https://doi.org/10.1145/3786335.3813159.

Wei Xiao and Calin Belta. High-order control barrier functions. *IEEE Transactions on Automatic Control*, 67(7):3655–3662, 2022. doi: 10.1109/TAC.2021.3105491.

Yifeng Xiao and Pierluigi Nuzzo. Symbolic temporal supervision of LLM agents using contracts, 2026a. arXiv:2609.18128.

Yifeng Xiao and Pierluigi Nuzzo. Contract-based architecture exploration of cyber-physical systems via satisfiability modulo convex programming. In *Design, Automation & Test in Europe (DATE)*, 2026b. to appear.

Yifeng Xiao, Chanwook Oh, Michele Lora, and Pierluigi Nuzzo. Efficient exploration of cyber-physical system architectures using contracts and subgraph isomorphism. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024. doi: 10.23919/DATE58400.2024.10546764.

Shaojun Xu, Xusheng Luo, Yutong Huang, Letian Leng, Ruixuan Liu, and Changliu Liu. Nl2hltl2plan: Scaling up natural language understanding for multi-robots through hierarchical temporal logic task specifications. *IEEE Robotics and Automation Letters*, 10(10):10482–10489, 2025. doi: 10.1109/LRA.2025.3598648.

A Positioning Table

Table 2 summarizes the positioning argument of Section 5.

Table 2 Positioning against representative prior work by problem class. *All interl.*: correct against every environment interleaving. ✓: yes. ×: no. ∼: partial or secondary.

Approach	Output	Multi-robot	All interl.	From NL
LLM planners (Kannan et al., 2024; Wang et al., 2024; Xu et al., 2025; Chen et al., 2023a)	plan/traj.	✓	×	✓
LLM+verifier (Xia et al., 2026; Lim et al., 2026)	checked plan	✓	∼	✓
INPROVF (Meng et al., 2025)	repaired ctrl.	×	✓	✓
GR(1) synth. (Kress-Gazit et al., 2009)	strategy	∼	✓	×
A/G contracts (Iannopollo et al., 2018)	contracts	∼	✓	×
Ours	**contracts + strat.**	✓	✓	✓

B Non-Vacuity

Non-vacuity. The two strengthened contracts of Section 3.4 are

$$\neg\mathrm{Real}\big(A_k,\ G_k \cup \{\Box\neg y\}\big) \quad \text{and} \quad \neg\mathrm{Real}\big(A_k,\ G_k \cup \{\neg p,\ \Box(p \to \bigcirc p),\ \Box(p \to \neg y),\ \Box\Diamond p\}\big). \tag{7}$$

If the first is realizable, a winning strategy exists that never sets y. If the second is, a winning strategy exists that eventually stops setting y. The assumption core $\Gamma_v \subseteq A_k$ satisfies $\mathrm{Real}(\Gamma_v, G') \wedge \big(\forall \varphi \in \Gamma_v,\ \neg\mathrm{Real}(\Gamma_v \setminus \{\varphi\}, G')\big)$, where G' is the strengthened guarantee set of (6) or (7), so removing any assumption of Γ_v makes the strengthened contract unrealizable. When Γ_v is empty, G' is realizable under no assumption, and the guarantee that requires y is returned instead: the liveness condition of G_k in which y occurs or, if none, the guarantee of G_{sys} that requires y, with the direction of adding that obligation to G_k.

C Coordination Policy Comparison

We test synthesized arbiters against fixed coordination policies on the request streams a robot team can produce. Each arbiter serves one resource for $n \in \{2, 4, 5\}$ robots and is synthesized from a contract certified by an earlier version of the pipeline with Gemini 3.1 Pro on a generated task (Appendix D). Its environment is unconstrained, so every request stream is admissible, and its contract requires mutual exclusion, no grant without a request, and eventual service of a standing request. Greedy grants every requesting robot. Fixed priority grants the lowest-indexed requesting robot. Static TDMA grants the robots in a fixed cyclic order regardless of requests. The elicited schedule is the answer of Gemini 3.1 Pro when asked for a dock schedule for n robots, which is a fixed rotation. For $n = 2$ we enumerate all 2^{16} request streams of length 8. For $n = 4$ and $n = 5$ we sample 200,000 streams of length $4n$ and add simultaneous, persistent and off-schedule request patterns. A request is flagged as waiting when it stands unserved throughout the final $2n + 1$ steps.

Table 3 Share of request streams on which each policy violates a requirement of the arbiter contract.

Policy	Violated requirement	$n = 2$	$n = 4$	$n = 5$
Greedy	mutual exclusion	90.0%	100%	100%
Fixed priority	waiting	0.098%	0.081%	0.037%
Static TDMA	grant without request	99.6%	$> 99.9\%$	100%
Elicited schedule	grant without request	99.6%	$> 99.9\%$	100%
Synthesized arbiter	none	**0**	**0**	**0**

Every fixed policy violates one requirement on some stream, and the synthesized arbiters violate none. Fixed priority rarely triggers the waiting flag, but under repeated simultaneous requests it never serves the lower-priority robot. The guarantee of the synthesized arbiter over all admissible streams follows from synthesis. The finite waiting flag alone does not establish liveness.

D FAIRNESS PROFILES

The generated arbitration tasks vary the team size from 2 to 8 robots, the number of contended resources from 1 to 3, and the *fairness profile* that the requirement asks of the arbiter, i.e., the rule by which standing requests are served. Under justice, every standing request is eventually granted, $\square\lozenge(grant_i \vee \neg req_i)$, which is weak fairness (Bloem et al., 2012). Under round robin, grants rotate, and no robot is served twice while another robot's request stands. Under k-capacity, at most k robots hold the resource at once (Fischer et al., 1979). Under strict priority, a lower-indexed robot is served first, together with the no-starvation requirement. Table 4 reports the certified runs per profile over 77 runs of an earlier version of the pipeline with Gemini 3.1 Pro, excluding runs lost to infrastructure errors.

Table 4 Certified runs per fairness profile (77 runs).

Fairness profile	Certified / runs
Justice	41/47
Round robin	13/14
k-capacity	11/13
Strict priority	0/3, unrealizable

The first three profiles, each a classical coordination discipline (Dijkstra, 1965), are certified in most runs. Strict priority with no starvation is not: under jointly held requests, $\square(req_1 \rightarrow grant_1)$ starves robot 2, which contradicts its liveness goal, and SLUGS reports the specification unrealizable. These three runs are correct outcomes of the pipeline, since the requirement itself cannot be met.

E SCALING RESULTS

Table 5 lists the number of states of the largest synthesized controller and the end-to-end time behind Fig. 4(c) for each architecture and team size. Flat times are medians over certified Claude Sonnet 4.6 runs, and the tree columns use no LLM. A cell (> 900) did not finish within $900\,\mathrm{s}$.

Table 5 Largest synthesized controller and end-to-end time per architecture and team size N.

	flat, LLM loop		reuse, two levels		reuse, three levels	
N	$\lvert Q_k\rvert$	time (s)	$\lvert Q_k\rvert$	time (s)	$\lvert Q_k\rvert$	time (s)
2	**28**	33	–	–	–	–
4	**192**	218	–	–	–	–
6	1632	273	**74**	1.8	–	–
8	10240	338	**190**	3.3	–	–
10	–	(>900)	**190**	5.1	–	–
12	–	–	96	6.7	**74**	7.1
18	–	–	576	15.1	**74**	15.5
24	–	–	3072	33.5	**96**	34.2
30	–	–	–	(>900)	**462**	41
48	–	–	–	–	**190**	204
60	–	–	–	–	**240**	557
72	–	–	–	–	–	(>900)