RELSHIELD: MAXIMALLY PERMISSIVE SHIELDS FOR TOOL-USING AGENTS OVER RELATIONAL DATA

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using language agents can automate business processes without hard-coding a separate control flow for every user request, but they must still satisfy standing requirements over the data they modify. Existing guardrails typically enforce safety specifications over simplified views of application state, which may omit relationships needed to enforce them correctly. This can cause guardrails to permit unsafe calls or unnecessarily block safe ones. To alleviate this problem, we introduce `RelShield`, a novel shielding framework that synthesizes restrictions directly from a first-order safety requirement over the underlying relational application state, together with tool specifications, legal initial states, and uncontrollable environment actions. From these inputs, we provide the safety-game fixed point characterization of the *maximally permissive shield*, i.e., the largest set of grounded calls that can remain available at each state while guaranteeing that every execution preserves the requirement. We first prove that shield synthesis is undecidable in this relational setting, and then give a general sound symbolic synthesis procedure. For a class of instances whose reachable safe states have bounded size, we prove synthesis becomes decidable and give a sound and complete synthesis method. We implement both procedures in `RelShield` and evaluate them on relational instances constructed from three established tool-using agentic benchmarks. Under a fixed budget, `RelShield` successfully synthesizes the maximally permissive shield for 78.0% of instances. For 31.3% of solved instances, `RelShield` proves that no additional shield restrictions are needed, certifying that the environment itself enforces compliance. Finally, we compare against an LLM baseline and find that its synthesized shields fail to match those produced by `RelShield` on at least a third of instances.

1 INTRODUCTION

Tool-using language models are increasingly employed as autonomous agents in business processes. They act by issuing sequences of tool calls that update the application's persistent data in pursuit of a user's goal (Qin et al., 2025; Vu et al., 2025; Xu et al., 2025). Their general knowledge and planning capabilities let them handle different requests without requiring developers to specify beforehand a separate control flow for each one (Calvanese et al., 2026; Yao et al., 2023). However, that autonomy must remain within the business requirements of the process. For an important class of these requirements, safety is determined by the data the process operates over, with constraints concerning properties of multiple business objects and the relations among them (Tosatto et al., 2026; Malay et al., 2026; Patil et al., 2025). For example, Figure 1 shows one such requirement for a telecommunications agent: *no customer with an overdue bill can have an active phone line*.

In order to build compliant agents with a high degree of autonomy, these requirements must be enforced without prescribing how the agent should complete each task. Existing guardrails do this by sitting between the agent and its tools and restricting which calls it may execute (Deng et al., 2025; Yu et al., 2025). In Figure 1, blocking `resume_line(c,l)` when customer c has an overdue bill prevents that call from immediately violating the requirement. However, whether blocking this call is enough to guarantee safety throughout execution depends on how the rest of the process can evolve later on. At the same time, blocking more calls than necessary removes part of the agent's autonomy. Thus, the challenge is to restrict the available tool calls just enough to guarantee the requirement as the execution unfolds while preserving all other safe choices.

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.

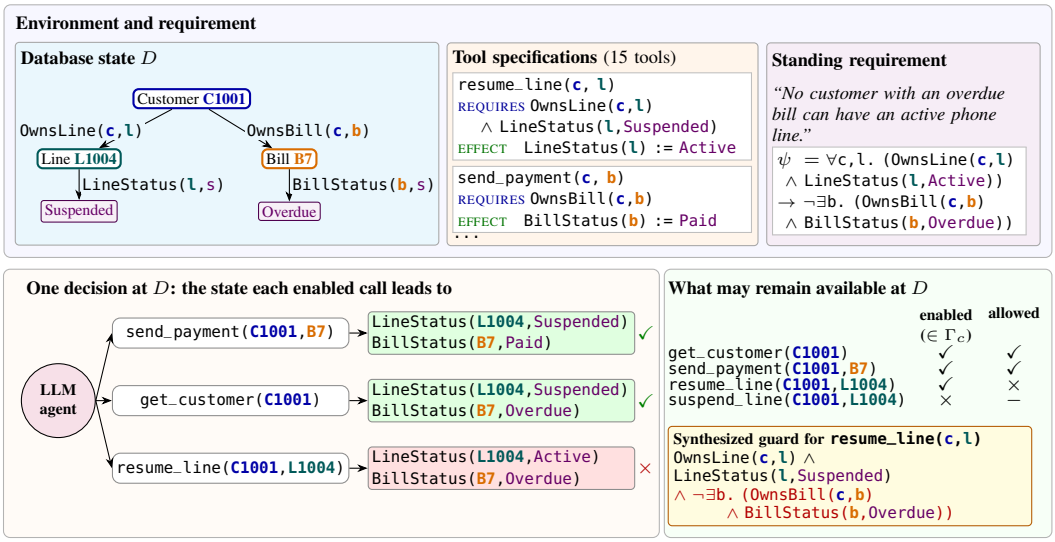


Figure 1: Example of the shield-synthesis problem adapted to the Telecom environment of τ^2-bench. The requirement is stated over the application's relational data, while the tool specifications determine which calls are enabled and how they update that data. The shield determines which grounded calls may remain available to the agent: `resume_line(C1001,L1004)` must be withheld.

However, today's guardrails do not typically operate with the application's data directly. They evaluate the specification they are given over something simpler: the arguments of the tool call (Shi et al., 2026b; Debenedetti et al., 2025), the sequence of calls made so far (Kamath et al., 2025), predicate abstractions written by the developer (Wang et al., 2025; Palumbo et al., 2026), rules extracted from policy text (Chen et al., 2025; Xiang et al., 2025; Luo et al., 2025b), or a model learned from past runs (Mou et al., 2026; Wang et al., 2026). As a consequence, complex relational requirements may not be expressible in the specification language a guardrail accepts, or would require hand-crafted state abstractions tailored to each new requirement.

In this work, we show that a safety requirement over relational application data can be enforced by restricting the tool calls offered to the agent without overblocking, and develop two procedures for synthesizing those restrictions automatically. We formulate the problem directly from the process description: a first-order safety requirement, initial states, the tool specifications, and any environment changes to the data. From these inputs, we define the synthesis target as the maximally permissive shield, namely the set of grounded calls that may remain available in each state while guaranteeing that every execution preserves the requirement. We leave the agent unspecified, making the guarantee hold regardless of its behaviour. We provide the safety-game fixed point characterization of this shield, show that the synthesis problem is undecidable and then develop two synthesis routes. First, we give a sound procedure that propagates the requirement symbolically through the tool specifications at each fixed-point iteration. Second, we identify a class of systems in which the reachable safe states have bounded size, show that synthesis is decidable, and give a sound and complete procedure that iterates the fixed-point over an exact finite abstraction of the system.

We implement both procedures in `RelShield`, a sound synthesis framework that is complete for the bounded class when supplied with a valid boundedness certificate. We evaluate `RelShield` on benchmark-derived environments with relational state from τ^2-bench (Barres et al., 2026), AppWorld (Trivedi et al., 2024), and WorkBench (Styles et al., 2024). Under a fixed budget, `RelShield` successfully synthesizes the maximally permissive shield for 78.0% of instances, with both synthesis procedures contributing. For 31.3% of solved instances, `RelShield` proves that no additional shield restrictions are needed, certifying that the environment itself enforces compliance. For the remaining instances, we find that the synthesized shield coincides with the one-step check. We then introduce variants of the same environments and find that small changes in uncontrollable dynamics can require safety-continuation reasoning. Finally, we compare `RelShield` against an

LLM baseline and find that the baseline fails to match the synthesized shields on at least a third of instances, often by unnecessarily blocking safe calls. Overall, we make the following contributions.

1. We formalize maximally permissive shielding for tool-using agents over relational application data, and we provide its safety-game fixed point characterization.
2. We prove that shield synthesis in this setting is undecidable in general, and provide a sound synthesis method that solves the fixed point symbolically. We then identify a class of bounded systems for which the problem becomes decidable and develop a sound and complete procedure that solves the fixed point extensionally over an exact finite abstraction while preserving the valid tool calls.
3. We implement both procedures in `RelShield` as a sound synthesis framework that is complete for the bounded class when supplied with a valid boundedness certificate.
4. We evaluate `RelShield` experimentally on instances constructed from τ^2-bench, AppWorld, and WorkBench. Our results demonstrate that the framework successfully synthesizes shields across varied deployment dynamics and can formally certify when an environment satisfies safety requirements natively. We show that frontier language models fail to match the certified shields on at least a third of instances.

2 Related work

Guardrails. A guardrail decides, when the agent proposes a tool call, whether the call may execute (Deng et al., 2025; Yu et al., 2025). Existing guardrails differ in what information about the application is available to that decision. Some operate primarily on the proposed call and its arguments (Shi et al., 2026b; Debenedetti et al., 2025). Others additionally use the call history (Kamath et al., 2025), or predicates and rules over selected application state (Wang et al., 2025; Palumbo et al., 2026). Such rules or checks may themselves be derived from user requests, policies, or past interactions (Xiang et al., 2025; Chen et al., 2025; Luo et al., 2025b), or learned from execution data (Mou et al., 2026; Li et al., 2026; Wang et al., 2026). What these methods can enforce therefore depends on the richness of the representation exposed to the guardrail. Deriving operational restrictions from higher-level process requirements with provable compliance has been identified as an open problem in agentic business processes (Calvanese et al., 2026; Acitelli et al., 2025).

Shielding. Shielding provides the formal background for some prior agent guardrails and for our work. A shield is synthesized from a safety specification and a model of the system so that unsafe actions can be withheld at runtime (Bloem et al., 2015; Könighofer et al., 2017); in its preemptive form, it restricts the actions available before the controller chooses (Alshiekh et al., 2018). Classical constructions compute the states from which safety can always be maintained and retain the actions that stay within them, yielding the least restrictive safe behavior (Ramadge & Wonham, 1987; Bernet et al., 2002). Later work extends this objective to richer symbolic and infinite-state representations (Rodríguez et al., 2025; Shi et al., 2026a; Samuel et al., 2021). First-order safety games and data-aware controller synthesis go further by reasoning over first-order or relational state and parameterized actions (Seidl et al., 2020; De Giacomo et al., 2022; de Leoni et al., 2020; Hofmann & Claßen, 2025). Our setting keeps the maximally permissive objective, but the agent's actions are the tool calls grounded in the objects of the current relational database. The set of available calls therefore changes from state to state and grows as objects are created, and each call has a set of possible outcomes chosen by the environment, which may be infinite.

Tool-using Agent Environments. We focus on stateful agent benchmarks relevant to the requirements studied in this work. In τ-bench, task success and policy compliance are distinct and evaluation depends on the resulting database state (Yao et al., 2025; Barres et al., 2026); AppWorld, WorkBench, Robotouille, and the function-calling leaderboard likewise evaluate the state left by agent actions (Trivedi et al., 2024; Styles et al., 2024; Gonzalez-Pumariega et al., 2025; Patil et al., 2025). These environments therefore support requirements over application data, but do not provide the set of calls such a requirement permits in each state. Safety benchmarks instead label actions or trajectories against a supplied policy (Levy et al., 2026; Luo et al., 2025a; Vijayvargiya et al., 2026; Ruan et al., 2024), while guardrail evaluations typically measure permissiveness indirectly through task utility (Debenedetti et al., 2024; Andriushchenko et al., 2025). Neither provides the tools to directly evaluate the maximally permissive shield synthesis problem natively

3 MAXIMALLY PERMISSIVE SHIELD FOR RELATIONAL ENVIRONMENTS

Environment model. We model the environment at the level of its relational application data, the enabled tool calls and uncontrollable actions that can modify that data, and a set of possible initial states. Leaving the agent unspecified makes guarantees hold for any actor. Figure 1 illustrates the environment and requirement at a decision point, and Appendix A.1 gives their full semantics.

State and requirement. We adopt the sorted relational-state model and first-order semantics of STEAD (Mercado & Lomuscio, 2026). A state D is a finite typed relational database containing the application facts represented in the model. In Figure 1, D contains a customer `C1001` that owns a suspended line with an overdue bill.

We write $D \models \varphi$ when a first-order formula φ holds in D, with quantifiers ranging over the active values of their respective sorts. The requirement is a closed first-order sentence ψ over the same relational schema (Figure 1, upper-right block). We write $Q_\psi = \{D : D \models \psi\}$ for the states that satisfy it. We focus on state invariants expressible in this language.

Calls and transitions. A tool schema becomes a *grounded call* when its arguments are bound to objects in the current state. At D, let $\Gamma_c(D)$ be the finite set of grounded calls enabled by the environment. Thus the "enabled" column in Figure 1 records membership in $\Gamma_c(D)$. For example, `resume_line(C1001,L1004)` is enabled although allowing it would violate the requirement.

For every enabled call m, the set $\varnothing \neq \tau(D, m)$ contains its possible successor states, allowing tool execution to be nondeterministic. The middle panel of Figure 1 depicts this semantics by connecting each enabled call to the facts in its successor state. The tool specifications shown above it are one finite representation from which Γ_c and τ are induced. Section 4.3 states the additional representation assumptions used for synthesis.

The state may also evolve independently of the agent's tool calls, for example through an external service, an asynchronous event, or a user directly modifying application data. We collect the enabled uncontrollable actions at D in $\Gamma_u(D)$ and use the same $\tau(D, m)$ notation for their possible successors. Unlike controllable calls in $\Gamma_c(D)$, these actions cannot be blocked by the shield.

Finally, let I be a nonempty set of initial states. We model the environment as a tuple

$$\mathcal{E} = \langle I, \Gamma_c, \Gamma_u, \tau \rangle. \tag{1}$$

Together with the requirement ψ, it defines a shield-synthesis instance. Figure 1 shows D, ψ, $\Gamma_c(D)$, and τ for one decision point.

Shielding. A *shield* is a state-dependent mapping S with $S(D) \subseteq \Gamma_c(D)$. Starting from $D_0 \in I$, an execution may proceed either through a call in $S(D)$ selected by the agent or through an uncontrollable action in $\Gamma_u(D)$. Let $\mathsf{Reach}_S(I)$ denote the states reachable under these choices. We say that S *enforces* the requirement when $\mathsf{Reach}_S(I) \subseteq Q_\psi$.

We call S *deadlock-free* when $S(D) \neq \varnothing$ at every $D \in \mathsf{Reach}_S(I)$ for which $\Gamma_c(D) \neq \varnothing$, and *correct* when it enforces the requirement and is deadlock-free. Deadlock freedom rules out satisfying the requirement by blocking every controllable call at a state where the environment enables one.

Maximally Permissive Shield: We define a correct shield S as *maximally permissive* if, for every correct shield S' and every $D \in \mathsf{Reach}_{S'}(I)$, $S'(D) \subseteq S(D)$. To characterize it, we start from a candidate set of states X and define

$$F_\psi(X) = \{D \in Q_\psi : (\Gamma_c(D) = \emptyset \vee \exists a \in \Gamma_c(D).\, \tau(D, a) \subseteq X) \wedge \forall u \in \Gamma_u(D).\, \tau(D, u) \subseteq X\}.$$

Thus, $F_\psi(X)$ contains exactly those states satisfying ψ from which every uncontrollable transition remains in X and, whenever controllable calls are enabled, at least one such call has all of its outcomes in X. The operator F_ψ is monotone, so it has a greatest fixed point. Define the *winning region* W and the shield $S^\star$ by

$$W = \nu X.\, F_\psi(X), \qquad S^\star(D) = \{\, a \in \Gamma_c(D) : \tau(D, a) \subseteq W \,\}. \tag{2}$$

Theorem 1 (Maximally Permissive Shield). *Given an environment $\mathcal{E}$ and requirement ψ, a correct shield exists exactly when $I \subseteq W$. In that case, $S^\star$ is correct and maximally permissive, and every correct shield S satisfies $S(D) \subseteq S^\star(D)$ for every $D \in \mathsf{Reach}_S(I)$. Moreover, a shield S is correct and maximally permissive exactly when $S(D) = S^\star(D)$ for every $D \in \mathsf{Reach}_{S^\star}(I)$.*

The theorem follows from the greatest-fixed-point property. The states reachable under any correct shield form a post-fixed point of F_ψ and therefore lie in W. Appendix A.2 gives the full proof and shows that allowing the shield to depend on execution history does not permit any additional calls. The construction of W and $S^\star$ does not depend on I. The initial states determine whether a correct shield exists and which part of $S^\star$ can affect execution. For a fixed instance, synthesis therefore only needs to recover $S^\star$ on $\mathsf{Reach}_{S^\star}(I)$.

Finally, to distinguish the future viability reasoning of S^*, we introduce the *one-step shield*:

$$S_1(D) = \{\, a \in \Gamma_c(D) : \tau(D, a) \subseteq Q_\psi \,\}. \tag{3}$$

Since $W \subseteq Q_\psi$, for every state D we have $S^\star(D) \subseteq S_1(D) \subseteq \Gamma_c(D)$. The one-step shield removes calls that can violate ψ immediately, while $S^\star$ may additionally remove calls that preserve ψ now but lose future enforceability. In the following section we study how to compute $S^\star$.

4 Shield Synthesis

The previous section characterizes the maximally permissive shield $S^\star$ on the theoretical level. In this section, we focus on computing it from the environment and requirement. We first show that this problem is undecidable in general.

Theorem 2 (Undecidability)**.** *Given a finitely specified environment $\mathcal{E}$, a requirement ψ, a state D_0, and a controllable call $a \in \Gamma_c(D_0)$, deciding whether $a \in S^\star(D_0)$ is undecidable.*

Since any procedure computing $S^\star$ would decide this membership question, the theorem shows that the synthesis problem is undecidable as well. Appendix B.1 proves the result by reduction from safety verification of relational data-centric systems over infinite data domains (Hariri et al., 2013), showing that the source of undecidability is the unbounded relational data itself.

In the rest of this section we develop two synthesis procedures. Each procedure starts from assumptions on how Γ_c, Γ_u and τ are specified. Closely related assumptions are standard in the reasoning-about-actions and data-aware process literatures (Reiter, 2001; Calvanese et al., 2013). Section 3 describes how `RelShield` instantiates these assumptions using first-order relational tool specifications with functional relational effects.

4.1 Symbolic Synthesis

Representation Assumptions. For this procedure, we assume the following properties for the tool specifications, which are then formalized in Appendix B.2. For every grounding $a(\bar{u})$, we require

$$D \models \mathrm{Pre}_a(\bar{u}) \iff a(\bar{u}) \in \Gamma_i(D),$$
$$D \models \mathrm{WP}_a(\varphi)(\bar{u}) \iff \forall D' \in \tau(D, a(\bar{u}))\,.\, D' \models \varphi,$$

where $i = c$ for controllable schemas and $i = u$ for uncontrollable schemas.

Symbolic Fixed-point: Because even Q_ψ may contain infinitely many relational states, explicit state enumeration is unavailable in general. We instead represent the descending fixed-point sequence $X_0 = Q_\psi, X_{k+1} = F_\psi(X_k)$ symbolically by first-order formulas φ_k. Let $\varphi_0 = \psi$. For a controllable tool schema a, define $G_a^k(\bar{x}) = \mathrm{Pre}_a(\bar{x}) \wedge \mathrm{WP}_a(\varphi_k)(\bar{x})$. The satisfying groundings of G_a^k are exactly the enabled calls of a whose outcomes all remain in X_k. In particular, G_a^0 defines the calls retained by the one-step check S_1. In Figure 1, $G^0_{\texttt{resume_line}}$ is the synthesized guard that blocks `resume_line` when the customer has an overdue bill.

Let $\mathsf{None}_c = \bigwedge_{a \in \mathcal{A}_c} \forall \bar{x}.\, \neg \mathrm{Pre}_a(\bar{x})$ denote that no controllable call is enabled, with each argument quantified over the active domain of its sort. One symbolic application of F_ψ is then

$$\varphi_{k+1} = \psi \wedge \underbrace{\Big(\mathsf{None}_c \vee \bigvee_{a \in \mathcal{A}_c} \exists \bar{x}.\, G_a^k(\bar{x})\Big)}_{\text{the agent can act safely for another step}} \wedge \underbrace{\bigwedge_{u \in \mathcal{A}_u} \forall \bar{y}.\, \Big(\mathrm{Pre}_u(\bar{y}) \to \mathrm{WP}_u(\varphi_k)(\bar{y})\Big)}_{\text{every enabled uncontrollable action remains in } X_k}. \tag{4}$$

Equation 4 is exactly the operator F_ψ evaluated on a first-order representation of X_k rather than on an explicit set of states.

Theorem 3 (Exact Symbolic Synthesis)**.** *Every φ_k is effectively constructible and satisfies $X_k = \{D : D \models \varphi_k\}$. If $\varphi_k \equiv \varphi_{k+1}$ for some k, then $X_k = W$, and G_a^k defines $S^\star$ for every $a \in \mathcal{A}_c$.*

The result follows by induction on the symbolic fixed-point iteration. Thus, whenever $\varphi_k \equiv \varphi_{k+1}$ can be certified, the procedure recovers $S^\star$ exactly (Appendix B.3). Since first-order equivalence is undecidable in general, this procedure is sound but incomplete. The next approach recovers completeness for a particular class of bounded instances.

4.2 Complete Synthesis

Representation Assumptions. For this procedure, we require the relational dynamics to be invariant under renaming of object identifiers. Let C contain the constants fixed by the relational specification. We adopt the C-isomorphism convention of STEAD (Mercado & Lomuscio, 2026): two states are *C-isomorphic*, written $D \simeq_C \bar{D}$, if they differ only by a sort-preserving renaming of identifiers outside C. We require such renamings to preserve the controllable and uncontrollable enabled calls and the possible outcomes of each grounded call. We, however, do not require the identifiers occurring in the initial states to belong to C. The formal conditions are given in Appendix B.4.

Restricted Fixed-point. The symbolic procedure targets $S^\star$ over the full state space. For a fixed instance, however, Theorem 1 shows that synthesis only needs $S^\star$ on $\mathsf{Reach}_{S^\star}(I)$. Since this set depends on the unknown shield, we instead consider the states reachable from I under Γ_c while remaining in Q_ψ. We call this the *safe reachable set* and denote it by $\mathcal{R}_\psi(I)$.

If a correct shield exists, then $\mathsf{Reach}_{S^\star}(I) \subseteq \mathcal{R}_\psi(I) \subseteq \mathsf{Reach}_{\Gamma_c}(I)$. Moreover, $\mathcal{R}_\psi(I)$ is closed under successors that remain in Q_ψ, so the winning-region computation can be restricted to it:

$$W_{\mathcal{R}} = \nu X.\big(\mathcal{R}_\psi(I) \cap F_\psi(X)\big) = W \cap \mathcal{R}_\psi(I). \tag{5}$$

Hence using $W_{\mathcal{R}}$ in place of W is enough to recover $S^\star$ on $\mathsf{Reach}_{S^\star}(I)$ (Appendix B.4).

Finite Abstraction. Suppose the safe reachable states contain only finitely many active values of each sort, up to renaming, and each action introduces only finitely many fresh values in one step. We will show that, when $R_\psi(I)$ is bounded in size, a finite universe can contain a C-isomorphic representative of every relevant state and successor. This allows us to solve the restricted fixed point over a finite abstraction and recover the original one and its corresponding S^* exactly.

Theorem 4 (Finite Abstraction)**.** *Assume that for every sort S there exists a corresponding finite bound n_S such that $|(\mathsf{ad}(D) \setminus C) \cap U_S| \leq n_S$ for every $D \in \mathcal{R}_\psi(I)$. Let f_S be the maximum number of fresh sort-S objects that any single action may create, and let $\mathcal{E}_f$ be the same environment interpreted over finite universes U_S^f satisfying $|U_S^f| \geq |C \cap U_S| + n_S + f_S$ for every sort S, with C-isomorphic representatives of the initial states.*

Then, the safe reachable set under $\mathcal{E}_f$ is finite, and every $D \in \mathcal{R}_\psi(I)$ has a C-isomorphic representative $\bar{D}$ in that set. Let $\bar{W}$ and $\bar{S}^\star$ be obtained by solving equation 5 over $\mathcal{E}_f$. For every such pair and every witnessing isomorphism γ, we have

$$D \in W \iff \bar{D} \in \bar{W}, \qquad \gamma\big(S^\star(D)\big) = \bar{S}^\star(\bar{D}).$$

The proof first uses the renaming invariance notions above to establish a one-step correspondence between enabled grounded calls and their possible outcomes in the concrete and finite environments. To match their shields, we lift this correspondence through the restricted safety-game fixed point. Consequently, Theorem 4 preserves not only winning-region membership but also $S^\star(D)$ under the witnessing isomorphism. Appendix B.4 gives the full formal proof, and the following section shows how `RelShield` turns this construction into a concrete synthesis procedure.

4.3 RelShield

`RelShield` applies the two synthesis procedures sequentially. It first attempts symbolic synthesis, which can obtain a global shield independent of the initial states. If stabilization is not certified within budget, it switches to the finite procedure for the given initial states. The latter is complete once a valid boundedness certificate is available, either supplied directly or proved automatically. `RelShield` satisfies the representation assumptions above using standard first-order relational tool

specifications from data-aware literature (Belardinelli et al., 2012). For the symbolic procedure, it uses the standard regression construction for functional relational updates (Calvanese et al., 2020). Because our formulas use active-domain semantics, regression additionally tracks the successor active domain when a call creates or deletes objects. Appendix B.2 gives this adaptation.

Symbolic certification. `RelShield` runs the iteration of Section 4.1 and at each round attempts to certify $\varphi_k \equiv \varphi_{k+1}$. Since the sequence is descending, it is enough to prove $\varphi_k \to \varphi_{k+1}$. At each step, it checks this implication with an SMT solver. A successful proof is a stabilization certificate and, by Theorem 3, yields $G^\star_a(\bar{x}) = \mathrm{Pre}_a(\bar{x}) \wedge \mathrm{WP}_a(\varphi_k)(\bar{x})$ for every controllable tool schema a.

Boundedness certification. If symbolic stabilization is not certified within budget, `RelShield` attempts the finite procedure of Section 4.2. This requires a valid bound on the number of identifiers of each sort in $\mathcal{R}_\psi(I)$. `RelShield` can search for such a certificate by expressing candidate cardinality bounds as first-order formulas and querying the SMT solver to prove sufficient invariance obligations. This search is incomplete because boundedness is undecidable in general. A known bound may instead be supplied directly as an invariant. When appropriate, a bound may also be included as an invariant in the requirement itself. Appendix B.5 gives the corresponding certificate encodings and proof obligations.

Finite synthesis and deployment. Once boundedness is certified, `RelShield` chooses finite per-sort universes with the sizes required by Theorem 4. It explores from the initial states using both controllable and uncontrollable actions, stopping expansion at successors that violate ψ. Each explored state is canonicalized by renaming identifiers outside C into the chosen finite universes and selecting a fixed representative of its C-isomorphism class. `RelShield` then evaluates equation 5 to convergence over these states, obtaining $\bar{W}$ and the table $\bar{D} \mapsto \bar{S}^\star(\bar{D})$.

The same canonicalization is used at runtime. For a state D, `RelShield` records a witness γ mapping to its canonical representative, queries the solved table, and maps the allowed calls back:

$$D \xrightarrow{\gamma} \bar{D} \xrightarrow{\bar{S}^\star} \bar{S}^\star(\bar{D}) \xrightarrow{\gamma^{-1}} S^\star(D).$$

This mapping is correct by Theorem 4. Finally, after either procedure synthesizes $S^\star$, `RelShield` checks whether it agrees with the one-step shield S_1 on the relevant synthesis scope. If so, it emits the simpler guard $\mathrm{Pre}_a \wedge \mathrm{WP}_a(\psi)$ and otherwise it retains the synthesized representation.

Thus, `RelShield` provides a novel offline framework for synthesizing maximally permissive shields for tool-using agents. The two procedures that form its backbone come with correctness guarantees, together with a completeness guarantee when the system is proved bounded. In the following section, we evaluate a concrete `RelShield` implementation *experimentally* over relational instances constructed from three established tool-using agentic benchmarks, to understand what maximally permissive shields look like in such deployments and to learn from the synthesis limitations where future research should focus.

5 Experimental Evaluation

In this section, we evaluate a `RelShield` implementation [1] with a fixed resource budget on instances constructed from three tool-using agentic benchmarks with relational states. On instances where it produces a result, we use the shield to study what type of safety reasoning these benchmarks require. However, prior work does not address our specific setting and the benchmarks do not provide ground-truth shields to compare with. Thus, we build an LLM baseline to study how difficult it is to derive these maximally permissive shields from the environment specification.

5.1 Setup

Many tool-using agent benchmarks expose the kind of relational application state that motivates our setting. However, they are designed primarily to evaluate task success or tool-use trajectories and do not directly provide the formal environment models needed to synthesize shields over operational data. We therefore construct relational environments from three complementary benchmarks. τ^2-bench provides relational customer-service workflows together with business policies written in

[1] The artifact and results are in `https://anonymous.4open.science/r/relshield-058C/`

Table 1: `RelShield` synthesis results. The upper block is the main census on the benchmark environments with percentages taken over it. The last row reports the controlled extensions separately.

			RelShield			Shield class		
Benchmark	Env.	Inst.	Symbolic	Finite	NC	$S^\star{=}\Gamma_c$	$S^\star{=}S_1$	$S^\star{\subsetneq}S_1$
τ^2-bench Telecom	2	7	7	0	0	2	5	0
τ^2-bench Airline	8	14	12	0	2	3	9	0
τ^2-bench Banking	4	5	1	3	1	0	4	0
AppWorld	2	7	2	0	5	2	0	0
WorkBench	3	8	7	0	1	3	4	0
Total	19	41	29	3	9	10	22	0
Share			70.7%	7.3%	22.0%	31.3%	68.8%	0.0%
Controlled extensions	8	9	0	8	1	1	2	5

natural language, from which we formalize requirements for its Telecom, Airline, and Banking domains (Barres et al., 2026). WorkBench provides office applications backed by relational tables and tool APIs, and we derive requirements from constraints exposed by its schemas and implementation (Styles et al., 2024). AppWorld provides SQL-backed applications with relations among objects and stateful object lifecycles, and we derive requirements from constraints expressed by its schemas and application endpoints (Trivedi et al., 2024).

We model each environment using the relational semantics of Section 3 and the first-order tool specifications of Section 4.3. For each benchmark, we retain the state needed to express the selected requirements and determine the enabledness and effects of the modelled tools. We compile those effects into relational transitions and select initial states from the benchmark data. This construction yields 41 instances over 19 environments and 32 requirements. Appendix C gives the modelled fragments with the tool-specifications, initial states and requirements.

5.2 Shield Synthesis

We run each instance once on a 64-core host with 251 GB of memory, allowing six symbolic rounds with a 30-second solver timeout, a 20,000-state cap for the finite procedure, and one hour of wall clock. Table 1 presents the results over the census.

Successful synthesis. `RelShield` successfully synthesizes $S^\star$ for 78.0% of the instances, with 70.7% solved through the symbolic procedure and 7.3% through the finite procedure. For 31.3% of the solved instances, `RelShield` proves that $S^\star = \Gamma_c$, certifying that the environment already preserves the safety requirement from the supplied initial states without any additional restriction, regardless of how an agent could choose among the available calls. In the remaining 68.8% of solved instances, `RelShield` proves that $S^\star = S_1$, showing that a one-step shield is sufficient to preserve the requirement throughout execution. This allows `RelShield` to deploy compact guards such as the one in Figure 1.

Controlled extensions. None of the solved main-census instances requires restrictions beyond S_1, so preserving safety in these benchmark-derived environments does not demand reasoning about how a currently safe call affects future enforceability. We therefore construct nine configurations by operationalizing suitable environment dynamics for Telecom and Banking (Table 1; Appendix D). In Telecom, we add an uncontrollable transition under which a past-due issued bill can become overdue, so resuming service can be safe immediately but expose a later violation. In Banking, we introduce clock progression, so allocating the last free dispute slot to a newer request can prevent an older request from meeting its deadline. On these configurations, Table 1 shows that `RelShield` can successfully derive restrictions beyond S_1 precisely when the added dynamics introduce future transitions that the agent cannot independently control.

Synthesis limits. The instances for which `RelShield` fails to synthesize a shield expose two different implementation limits. Eight of them exhaust the resource budget at the symbolic step. The remaining case exposes a proof-language limitation: certifying the Banking *card* bound requires richer relational invariants than the templates of Section 4.3 capture.

Table 2: LLM baseline results over five trials per model on the solved instances of Table 1, grouped by shield class. All metrics are percentages, and comparisons with $S^\star$ are over the $5n$ generations.

		`gpt-5.6-luna`				`deepseek-v4-pro`			
Shield	n	pass^1	pass^5	$\subsetneq S^*$	$\not\subseteq S^*$	pass^1	pass^5	$\subsetneq S^*$	$\not\subseteq S^*$
$S^\star{=}\Gamma_c$	10	84.0	50.0	16.0	–	68.0	60.0	30.0	–
$S^\star{=}S_1$	22	92.7	72.7	6.4	0.0	93.6	68.2	5.5	0.0
All	32	90.0	65.6	9.4	0.0	85.6	65.6	13.1	0.0
Controlled extensions									
$S^\star{=}S_1$	2	100.0	100.0	0.0	0.0	100.0	100.0	0.0	0.0
$S^\star{\subsetneq}S_1$	5	80.0	40.0	8.0	12.0	48.0	20.0	4.0	4.0

5.3 LLM Baselines

`RelShield` successfully synthesizes $S^\star$ for nearly 80% of the evaluated instances. Since every synthesized shield in the main census has a simple final form, we now focus on studying whether that is indeed the result of solving a simple problem, or a cheap operationalization of a previous harder reasoning step. Since existing benchmarks provide no ground-truth maximally permissive shields and there are no established synthesis baselines, we use frontier LLMs to test whether these synthesized shields can be reproduced reliably from the formal specification.

We evaluate `gpt-5.6-luna` and `deepseek-v4-pro` on the 32 solved census instances. Each model receives the schema, requirement, and full controllable and uncontrollable transition semantics, and generates a Python program deciding whether a grounded call is allowed in a given state. We evaluate these programs on fixed decision sets labelled by the $S^\star$ synthesized with `RelShield`. A mismatch therefore witnesses a concrete error, but agreement on all decisions does not prove correctness. We run five trials per model and report pass^1 and pass^5 as defined in τ-bench (Yao et al., 2025), together with the direction of mismatches. Appendix E.1 provides the full details.

Neither baseline reproduces the reference decisions reliably across trials, with pass^5 below two thirds overall. Among decision mismatches in the main census, every one is an over-restriction, meaning that the generated program blocks a call that $S^\star$ permits. Strikingly, even when `RelShield` certifies that no additional restriction is needed, pass^5 is at most 60.0%. Thus, even a simple final shield need not be straightforward to derive from the environment specification. On the extensions requiring $S^\star \subsetneq S_1$, pass^5 falls as low as 20.0%, and some generated programs instead admit calls that leave the winning region.

6 Conclusions

This work provides, to the best of our knowledge, the first formal framework for synthesizing maximally permissive shields for tool-using agents over relational operational data. Beyond its successful synthesis results, the formal guarantees provided by `RelShield` also allow it to serve as an auditing tool for agentic systems. For example, `RelShield` can prove when an environment already preserves a requirement natively, and our empirical evaluation shows that this case is common in the benchmarks we study. This way, developers could use `RelShield`'s certificate to avoid including the requirement in the agent's prompt, which could result in unnecessary conditioning. Furthermore, we observed that when shielding is needed, the benchmark instances required only immediate one-step checks rather than reasoning about future safety viability. Yet, we found that making explicit already-modelled environment dynamics is enough to make such reasoning necessary. This suggests that current benchmark dynamics may underexercise an important dimension of tool-agent safety. An immediate direction for future work is to study whether this pattern extends to other domains, and if so, to design benchmarks with more complex relational dynamics that require such reasoning over time. Although our modelling assumptions naturally accommodate the relational structure of current domains, expanding the framework to richer state representations also remains an important direction. Improving the scalability of the synthesis procedures so that they terminate on a broader range of instances will further expand the practical applicability of the approach.

AI USE STATEMENT

We used generative AI tools to assist with generating the relational models from the benchmarks codebases for the experimental evaluation, implementing and debugging the corresponding code, obtaining feedback on the methodology and presentation, polishing portions of the manuscript including the exposition of the appendix, polishing figures and identifying additional related work. The central problem formulation, theoretical claims, proofs, experimental design choices, interpretation of results, and final scientific conclusions are the authors' own. All AI-assisted improvements, including text and code artifacts, were reviewed, edited, and validated by the authors before inclusion. We take responsibility for the final content of this work, including all text, claims, code, data, figures, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The main contributions of this work are theoretical, with Sections 3 and 4 stating the formal setting and synthesis procedures, and Appendices A and B providing the complete definitions and proofs. Section 5 describes the experimental setup and evaluation protocol, while Appendices C,D and E provide the modelling of the benchmark-derived instances, controlled extensions and LLM evaluation details. The `RelShield` implementation for the experimental evaluation is in `https://anonymous.4open.science/r/relshield-058C/`, together with the experimental inputs and outputs, prompts, and scripts needed to reproduce the reported empirical results.

REFERENCES

Giacomo Acitelli, Anti Alman, Fabrizio Maria Maggi, and Andrea Marrella. Achieving framed autonomy in AI-augmented business process management systems through automated planning. *Inf. Syst.*, 133:102573, 2025.

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *AAAI Conference on Artificial Intelligence*, 2018.

Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, J. Zico Kolter, Matt Fredrikson, Yarin Gal, and Xander Davies. AgentHarm: A benchmark for measuring harmfulness of LLM agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik R Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. In *International Conference on Machine Learning (ICML)*, 2026. URL `https://openreview.net/forum?id=OC2z7iSQKa`.

Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. An abstraction technique for the verification of artifact-centric systems. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012.

Julien Bernet, David Janin, and Igor Walukiewicz. Permissive strategies: from parity games to safety games. *RAIRO Theor. Informatics Appl.*, 36(3):261–275, 2002.

Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: Runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.

Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: a database theory perspective. In *ACM Symposium on Principles of Database Systems (PODS)*, 2013.

Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. SMT-based verification of data-aware processes: a model-theoretic approach. *Math. Struct. Comput. Sci.*, 30(3):271–313, 2020.

Diego Calvanese, Angelo Casciani, Giuseppe De Giacomo, Marlon Dumas, Fabiana Fournier, Timotheus Kampik, Emanuele La Malfa, Lior Limonad, Andrea Marrella, Andreas Metzger, Marco Montali, Daniel Amyot, Peter Fettke, Artem Polyvyanyy, Stefanie Rinderle-Ma, Sebastian Sardiña, Niek Tax, and Barbara Weber. Agentic business process management: A research manifesto. *Inf. Syst.*, 140:102738, 2026.

Zhaorun Chen, Mintong Kang, and Bo Li. ShieldAgent: Shielding agents via verifiable safety policy reasoning. In *International Conference on Machine Learning (ICML)*, 2025.

Giuseppe De Giacomo, Paolo Felli, Brian Logan, Fabio Patrizi, and Sebastian Sardiña. Situation calculus for controller synthesis in manufacturing systems with first-order state representation. *Artif. Intell.*, 302:103598, 2022.

Massimiliano de Leoni, Paolo Felli, and Marco Montali. Strategy synthesis for data-aware dynamic systems with multiple actors. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2020.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025.

Zehang Deng, Yongjian Guo, Changzhou Han, Wanlun Ma, Junwu Xiong, Sheng Wen, and Yang Xiang. AI agents under threat: A survey of key security challenges and future pathways. *ACM Comput. Surv.*, 57(7):182:1–182:36, 2025.

Gonzalo Gonzalez-Pumariega, Leong Su Yean, Neha Sunkara, and Sanjiban Choudhury. Robotouille: An asynchronous planning benchmark for LLM agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *ACM Symposium on Principles of Database Systems (PODS)*, 2013.

Till Hofmann and Jens Claßen. LTLf synthesis on first-order agent programs in nondeterministic environments. In *AAAI Conference on Artificial Intelligence*, 2025.

Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Enforcing temporal constraints for LLM agents. *arXiv preprint arXiv:2512.23738*, 2025.

Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods Syst. Des.*, 51 (2):332–361, 2017.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, and Segev Shlomov. ST-WebAgentBench: A benchmark for evaluating safety and trustworthiness in web agents. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=MuCDzH0ctf.

Xu Li, Simon Yu, Minzhou Pan, Yiyou Sun, Bo Li, Dawn Song, Xue Lin, and Weiyan Shi. Unsafer in many turns: Benchmarking and defending multi-turn safety risks in tool-using agents. In *International Conference on Machine Learning (ICML)*, 2026. URL https://openreview.net/forum?id=iSuqYRmG4A.

Hanjun Luo, Shenyu Dai, Chiming Ni, Xinfeng Li, Guibin Zhang, Kun Wang, Tongliang Liu, and Hanan Salam. AgentAuditor: Human-level safety and security evaluation for LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a.

Weidi Luo, Shenghong Dai, Xiaogeng Liu, Suman Banerjee, Huan Sun, Muhao Chen, and Chaowei Xiao. AGrail: A lifelong agent guardrail with effective and adaptive safety detection. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025b.

Shiva Krishna Reddy Malay, Shravan Nayak, Jishnu Sethumadhavan Nair, Sagar Davasam, Aman Tiwari, Sathwik Tejaswi Madhusudhan, Sridhar Krishna Nemala, Srinivas Sunkara, and Sai Rajeswar. EnterpriseOps-Gym: Environments and evaluations for stateful agentic planning and tool use in enterprise settings. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*, 2026. URL https://openreview.net/forum?id=WQZfZjLHRa.

Alejandro J. Mercado and Alessio Lomuscio. Formal verification of agentic systems over operational data. *arXiv preprint arXiv:2608.03609*, 2026.

Yutao Mou, Zhangchi Xue, Lijun Li, Peiyang Liu, Shikun Zhang, Wei Ye, and Jing Shao. ToolSafe: Enhancing tool invocation safety of LLM-based agents via proactive step-level guardrail and feedback. In *Findings of the Association for Computational Linguistics: ACL*, 2026.

Nils Palumbo, Sarthak Choudhary, Jihye Choi, Guy Amir, Prasad Chalasani, and Somesh Jha. Formal policy enforcement for real-world agentic systems. *arXiv preprint arXiv:2602.16708*, 2026.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In *International Conference on Machine Learning (ICML)*, 2025.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, Chi Han, Yi R. Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Guoliang Li, Zhiyuan Liu, and Maosong Sun. Tool learning with foundation models. *ACM Comput. Surv.*, 57(4):101:1–101:40, 2025.

P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 07 2001.

Andoni Rodríguez, Guy Amir, Davide Corsi, César Sánchez, and Guy Katz. Shield synthesis for LTL modulo theories. In *AAAI Conference on Artificial Intelligence*, 2025.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *International Conference on Learning Representations (ICLR)*, 2024.

Stanly Samuel, Deepak D'Souza, and Raghavan Komondoor. GenSys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

Helmut Seidl, Christian Müller, and Bernd Finkbeiner. How to win first-order safety games. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2020.

Jieke Shi, Junda He, Zhou Yang, Dorde Zikelic, and David Lo. Synthesizing efficient and permissive programmatic runtime shields for neural policies. *ACM Trans. Softw. Eng. Methodol.*, 35(7): 210:1–210:38, 2026a.

Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Securing AI agents with privilege control. *arXiv preprint arXiv:2504.11703*, 2026b.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. WorkBench: a benchmark dataset for agents in a realistic workplace setting. In *Conference on Language Modeling (COLM)*, 2024.

Silvano Colombo Tosatto, Hannah M. S. Burke, Nick van Beest, and Heerko Groefsema. A first and fast symbolic approach for data-aware business process compliance checking. *Inf. Softw. Technol.*, 193:108047, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Sanidhya Vijayvargiya, Aditya Bharat Soni, Xuhui Zhou, Zora Zhiruo Wang, Nouha Dziri, Graham Neubig, and Maarten Sap. OpenAgentSafety: A comprehensive framework for evaluating real-world AI agent safety. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=xggSxCFQbA.

Hoang Vu, Nataliia Klievtsova, Henrik Leopold, Stefanie Rinderle-Ma, and Timotheus Kampik. Agentic business process management: Practitioner perspectives on agent governance in business processes. In *Business Process Management: Responsible BPM Forum*, 2025.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. *CoRR*, abs/2503.18666, 2025.

Haoyu Wang, Christopher M. Poskitt, Jiali Wei, and Jun Sun. ProbGuard: Proactive runtime monitoring for LLM agent safety via probabilistic prediction. *arXiv preprint arXiv:2508.00500*, 2026.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. GuardAgent: Safeguard LLM agents via knowledge-enabled reasoning. In *International Conference on Machine Learning (ICML)*, 2025.

Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Zhiruo Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025.

Miao Yu, Fanci Meng, Xinyun Zhou, Shilong Wang, Junyuan Mao, Linsey Pang, Tianlong Chen, Kun Wang, Xinfeng Li, Yongfeng Zhang, Bo An, and Qingsong Wen. A survey on trustworthy LLM agents: Threats and countermeasures. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2025.

APPENDIX

A SUPPLEMENT TO "MAXIMALLY PERMISSIVE SHIELD FOR RELATIONAL ENVIRONMENTS"

This supplement gives the full typed semantics for the environment model used in Section 3 and proves Theorem 1.

A.1 ENVIRONMENT MODEL

We work on the relational setting of STEAD (Mercado & Lomuscio, 2026). The definitions below make explicit the adaptation used by `RelShield`.

Definition A.1 (Relational state space)**.** Let $\mathscr{S}$ be a finite set of sorts and $U = \biguplus_{S\in\mathscr{S}} U_S$ a sorted universe. A finite typed relational schema with k relations has the form

$$\Sigma = \{R_i : S_{i,1} \times \cdots \times S_{i,r_i}\}_{i=1}^{k}.$$

A Σ-state over U is a map D such that

$$D(R_i) \in \mathcal{P}_{\text{fin}}\big(U_{S_{i,1}} \times \cdots \times U_{S_{i,r_i}}\big) \qquad (1 \le i \le k).$$

Thus every state has a finite active domain, i.e., finitely many values occur in its facts. We write $\mathcal{D}_\Sigma(U)$ for the set of all such states and define

$$\mathsf{ad}_S(D) = U_S \cap \bigcup_{i=1}^{k} \bigcup_{\bar{u}\in D(R_i)} \{u_1, \ldots, u_{r_i}\}, \qquad \mathsf{ad}(D) = \bigcup_{S\in\mathscr{S}} \mathsf{ad}_S(D).$$

We assume that first-order formulas over Σ are sorted and that constants denote fixed elements of their respective U_S. Satisfaction $D, \nu \models \varphi$ is otherwise the standard first-order satisfaction relation, with each relation symbol R interpreted by $D(R)$. Quantifiers use active-domain semantics. In particular, for x of sort S,

$$D, \nu \models \exists x\, \varphi \quad \Longleftrightarrow \quad \exists u \in \mathsf{ad}_S(D)\ D, \nu[x \mapsto u] \models \varphi.$$

with the remaining satisfaction clauses as usual. Let $\mathsf{FO}_0(\Sigma)$ denote the closed formulas of this language.

Grounded calls. Fix disjoint finite sets $\mathcal{A}_c$ and $\mathcal{A}_u$ of controllable tool schemas and uncontrollable action schemas, respectively. Each schema has a fixed sort signature $a : S_1 \times \cdots \times S_r$. For $\mathcal{A} \subseteq \mathcal{A}_c \cup \mathcal{A}_u$, let

$$\mathsf{Tools}_{\mathcal{A}}(U) = \{a(u_1, \ldots, u_r) \mid a : S_1 \times \cdots \times S_r \in \mathcal{A},\ u_i \in U_{S_i}\}$$

be the set of all well-sorted groundings of schemas in $\mathcal{A}$, and let

$$\mathsf{Tools}_{\mathcal{A}}(D) = \{a(u_1, \ldots, u_r) \in \mathsf{Tools}_{\mathcal{A}}(U) \mid u_i \in \mathsf{ad}_{S_i}(D)\}$$

be those whose arguments occur in the current state.

Definition A.2 (Relational environment)**.** A relational environment over $(\Sigma, U, \mathcal{A}_c, \mathcal{A}_u)$ is a tuple

$$\mathcal{E} = \langle I, \Gamma_c, \Gamma_u, \tau \rangle,$$

where

$$\varnothing \neq I \subseteq \mathcal{D}_\Sigma(U)$$

is the set of possible initial states,

$$\Gamma_c(D) \subseteq \mathsf{Tools}_{\mathcal{A}_c}(D), \qquad \Gamma_u(D) \subseteq \mathsf{Tools}_{\mathcal{A}_u}(D)$$

are respectively the controllable calls and uncontrollable actions enabled at each state $D \in \mathcal{D}_\Sigma(U)$, and

$$\tau : \{(D, m) \mid D \in \mathcal{D}_\Sigma(U),\ m \in \Gamma_c(D) \cup \Gamma_u(D)\} \longrightarrow 2^{\mathcal{D}_\Sigma(U)} \setminus \{\varnothing\}$$

assigns each enabled call or action a nonempty set of possible successor states.

For an enabled call or action m, we write

$$D \xrightarrow{m} D' \quad \Longleftrightarrow \quad D' \in \tau(D, m).$$

A requirement is a sentence $\psi \in \mathsf{FO}_0(\Sigma)$, with

$$Q_\psi = \{D \in \mathcal{D}_\Sigma(U) : D \models \psi\}.$$

An instance consists of an environment $\mathcal{E}$ together with a requirement ψ.

A *shield* is a map

$$S : \mathcal{D}_\Sigma(U) \to 2^{\mathsf{Tools}_{\mathcal{A}_c}(U)} \qquad \text{with} \qquad S(D) \subseteq \Gamma_c(D).$$

It induces

$$D \to_S D' \quad \Longleftrightarrow \quad \exists m \in S(D) \cup \Gamma_u(D).\ D' \in \tau(D, m),$$

and $\mathsf{Reach}_S(I)$ is the least subset of $\mathcal{D}_\Sigma(U)$ containing I and closed under $\to_S$. The shield enforces ψ when $\mathsf{Reach}_S(I) \subseteq Q_\psi$. It is *deadlock-free* when

$$\Gamma_c(D) \neq \varnothing \quad \Longrightarrow \quad S(D) \neq \varnothing$$

for every $D \in \mathsf{Reach}_S(I)$, and *correct* when it both enforces ψ and is deadlock-free.

A.2 Maximally Permissive Shield

Our shields are state-dependent. For comparison, we also consider the more general class of history-dependent shields.

Definition A.3 (History-dependent shield)**.** A finite history is a sequence $h = D_0 m_0 D_1 \cdots m_{k-1} D_k$ beginning in an initial state $D_0 \in I$ and following the transition semantics. A *history-dependent shield* H assigns to every history ending in D a set of allowed calls $H(h) \subseteq \Gamma_c(D)$. A history is *consistent* with H when every controllable call in it belongs to the set allowed after the preceding prefix (uncontrollable actions are never blocked). The shield is *deadlock-free* when

$$\Gamma_c(D) \neq \varnothing \quad \Longrightarrow \quad H(h) \neq \varnothing$$

for every reachable history consistent with H and ending in D. It enforces ψ when every state occurring in every such history satisfies ψ.

A memoryless shield S induces the history-dependent shield $H_S(h) = S(D)$ for the last state D of h.

Recall the monotone operator F_ψ from Section 3, its greatest fixed point $W = \nu X.F_\psi(X)$, and $S^\star(D) = \{a \in \Gamma_c(D) : \tau(D, a) \subseteq W\}$.

Lemma A.4 (Winning region)**.** *The following hold.*

1. $W \subseteq Q_\psi$.
2. *If* $D \in W$ *and* $\Gamma_c(D) \neq \varnothing$*, then* $S^\star(D) \neq \varnothing$.
3. *If* $D \in W$*, then every call in* $S^\star(D)$ *and every enabled uncontrollable action has all of its outcomes in* W*. That is,*
$$a \in S^\star(D) \Longrightarrow \tau(D, a) \subseteq W,$$
and
$$u \in \Gamma_u(D) \Longrightarrow \tau(D, u) \subseteq W.$$
4. *Let* H *be a deadlock-free history-dependent shield enforcing* ψ*. For every reachable history* h *consistent with* H *and ending in* D*,*
$$D \in W, \qquad H(h) \subseteq S^\star(D).$$

Proof. Because W is a fixed point of F_ψ, we have

$$W = F_\psi(W).$$

(1) By definition, every state in $F_\psi(W)$ satisfies ψ. Hence

$$W = F_\psi(W) \subseteq Q_\psi.$$

(2) Let $D \in W$ and suppose $\Gamma_c(D) \neq \varnothing$. Since $D \in F_\psi(W)$, the controllable clause in the definition of F_ψ gives some $a \in \Gamma_c(D)$ such that

$$\tau(D, a) \subseteq W.$$

By the definition of $S^\star$, this means $a \in S^\star(D)$. Therefore $S^\star(D) \neq \varnothing$.

(3) If $a \in S^\star(D)$, then $\tau(D, a) \subseteq W$ directly by the definition of $S^\star$. If $u \in \Gamma_u(D)$ and $D \in W = F_\psi(W)$, then the uncontrollable clause of F_ψ gives

$$\tau(D, u) \subseteq W.$$

Thus every transition permitted by $S^\star$, and every enabled uncontrollable transition, remains in W.

(4) Let R_H be the set of states reachable along histories consistent with H. Since H enforces ψ,

$$R_H \subseteq Q_\psi.$$

Take any $D \in R_H$, and let h be a reachable history consistent with H that ends in D. If $\Gamma_c(D) \neq \varnothing$, deadlock freedom gives some $a \in H(h)$. Every outcome of a is again in R_H, since executing an allowed call extends h to another history consistent with H. Likewise, every outcome of every enabled uncontrollable action is in R_H.

Therefore

$$R_H \subseteq F_\psi(R_H).$$

Since $W = \nu X.F_\psi(X)$ is the greatest fixed point of F_ψ, every post-fixed point of F_ψ is contained in W. Hence

$$R_H \subseteq W.$$

Finally, if $a \in H(h)$ and h ends in D, every outcome of a lies in $R_H \subseteq W$. Thus

$$\tau(D, a) \subseteq W,$$

and therefore $a \in S^\star(D)$. Hence

$$H(h) \subseteq S^\star(D).$$

□

Theorem 1 (Maximally Permissive Shield, restated). *Given an environment $\mathcal{E}$ and requirement ψ, a correct shield exists exactly when $I \subseteq W$. In that case, $S^\star$ is correct and maximally permissive, and every correct shield S satisfies*

$$S(D) \subseteq S^\star(D)$$

for every $D \in \mathsf{Reach}_S(I)$. Moreover, a shield S is correct and maximally permissive exactly when

$$S(D) = S^\star(D)$$

for every $D \in \mathsf{Reach}_{S^\star}(I)$.

Proof. Suppose first that $I \subseteq W$. By Lemma A.4(3), every transition allowed by $S^\star$, and every enabled uncontrollable transition, remains in W. Hence

$$\mathsf{Reach}_{S^\star}(I) \subseteq W.$$

By Lemma A.4(1), $W \subseteq Q_\psi$, so $S^\star$ enforces ψ. Moreover, whenever $D \in \mathsf{Reach}_{S^\star}(I)$ and $\Gamma_c(D) \neq \varnothing$, Lemma A.4(2) gives $S^\star(D) \neq \varnothing$. Thus $S^\star$ is deadlock-free and therefore correct.

Conversely, let S be any correct shield and set

$$R = \mathsf{Reach}_S(I).$$

Since S enforces ψ, we have $R \subseteq Q_\psi$. Take any $D \in R$. If $\Gamma_c(D) \neq \varnothing$, deadlock freedom gives some $a \in S(D)$. Since R is closed under transitions allowed by S,

$$\tau(D, a) \subseteq R.$$

Likewise, every enabled uncontrollable action has all of its outcomes in R. Hence

$$R \subseteq F_\psi(R).$$

Since $W = \nu X.F_\psi(X)$ is the greatest fixed point of F_ψ,

$$R \subseteq W.$$

In particular, $I \subseteq R \subseteq W$.

Finally, for any $D \in R$ and $a \in S(D)$, closure of R gives

$$\tau(D, a) \subseteq R \subseteq W.$$

Therefore $a \in S^\star(D)$, and hence

$$S(D) \subseteq S^\star(D).$$

It remains to prove the final characterization. Suppose first that S is correct and maximally permissive. Applying maximal permissiveness to the correct shield $S^\star$ gives

$$S^\star(D) \subseteq S(D)$$

for every $D \in \mathsf{Reach}_{S^\star}(I)$. These inclusions imply $\mathsf{Reach}_{S^\star}(I) \subseteq \mathsf{Reach}_S(I)$ by induction on executions. The containment proved above therefore gives the reverse inclusion $S(D) \subseteq S^\star(D)$ at every such state, and hence equality.

Conversely, suppose $S(D) = S^\star(D)$ for every $D \in \mathsf{Reach}_{S^\star}(I)$. Then S and $S^\star$ induce the same reachable states, so S is correct. For any correct shield S', the containment proved above implies that every execution of S' is also an execution of $S^\star$. Thus, for every $D \in \mathsf{Reach}_{S'}(I)$,

$$S'(D) \subseteq S^\star(D) = S(D),$$

so S is maximally permissive. □

Corollary 1 (History does not increase permissiveness)**.** *Let H be any correct history-dependent shield. For every reachable history h consistent with H and ending in D,*

$$H(h) \subseteq S^\star(D).$$

Consequently, a correct history-dependent shield exists if and only if a correct state-dependent shield exists, and $S^\star$ is maximally permissive even among history-dependent shields.

Proof. Lemma A.4(4) gives both $H(h) \subseteq S^\star(D)$ and $I \subseteq W$. Theorem 1 then gives a correct state-dependent shield $S^\star$. The converse is immediate, since a state-dependent shield is a special case of a history-dependent one. □

B Supplement to "Shield Synthesis"

A *finitely specified environment* is an environment whose schema, initial-state set, enabled-call maps, and transition semantics are given by finite effective descriptions. An instance additionally fixes a requirement ψ. We first show that exact shield synthesis is undecidable at this general level. We then introduce the additional structure used by the two constructive procedures.

B.1 Undecidability

Theorem 2 (Undecidability, restated). *Given a finitely specified environment $\mathcal{E}$, a requirement ψ, a state D_0, and a controllable call $a \in \Gamma_c(D_0)$, deciding whether*

$$a \in S^\star(D_0)$$

is undecidable. This remains true when $I = \{D_0\}$, there are no uncontrollable actions, and every state enables exactly one nullary controllable call.

Proof. Verification of finitely specified relational data-centric systems with nondeterministic services against propositional invariants is undecidable (Hariri et al., 2013, Theorem 5.1). Let $\mathcal{S}$ be such a system, with initial state D_0, transition relation $\Rightarrow_{\mathcal{S}}$, and propositional state formula p. Thus, deciding whether p holds at every state reachable from D_0 is undecidable.

From $\mathcal{S}$, construct an environment with the same relational state space, initial set

$$I = \{D_0\},$$

no uncontrollable actions, and a single nullary controllable call $\texttt{step}()$, enabled at every state. Define its outcomes by

$$\tau(D, \texttt{step}()) = \{D\} \cup \{D' : D \Rightarrow_{\mathcal{S}} D'\},$$

and take the requirement to be

$$\psi = p.$$

The construction is effective: the finite specification of $\Rightarrow_{\mathcal{S}}$, together with the identity outcome, gives a finite specification of τ. The identity outcome also ensures that $\tau(D, \texttt{step}())$ is nonempty at every state.

Because $\texttt{step}()$ is the only enabled controllable call, deadlock freedom forces every correct shield to allow it at every reachable state. Its transition relation is the reflexive closure of $\Rightarrow_{\mathcal{S}}$, which has exactly the same reachable states. Hence a correct shield exists exactly when p holds at every state reachable in $\mathcal{S}$.

By Theorem 1, a correct shield exists exactly when $D_0 \in W$. It remains to relate this to membership in $S^\star(D_0)$. If $D_0 \in W$, then Lemma A.4(2) gives $S^\star(D_0) \neq \varnothing$; since $\texttt{step}()$ is the only enabled call,

$$\texttt{step}() \in S^\star(D_0).$$

Conversely, if $\texttt{step}() \in S^\star(D_0)$, then

$$\tau(D_0, \texttt{step}()) \subseteq W.$$

Since $D_0 \in \tau(D_0, \texttt{step}())$, we obtain $D_0 \in W$.

Therefore

$$\texttt{step}() \in S^\star(D_0) \quad \Longleftrightarrow \quad D_0 \in W \quad \Longleftrightarrow \quad \mathcal{S} \models Gp.$$

Deciding membership in $S^\star(D_0)$ would therefore decide the undecidable source verification problem. □

B.2 First-Order Regression

The symbolic synthesis procedure represents sets of states by first-order formulas. To compute these sets backwards through a tool or uncontrollable action, we require first-order descriptions of when its groundings are enabled and of when all of their possible outcomes satisfy a given formula.

Definition B.1 (First-order regression)**.** For each schema $a \in \mathcal{A}_c \cup \mathcal{A}_u$, assume the following.

1. There is a first-order formula $\mathrm{Pre}_a(\bar{x})$ such that, for every state D and well-sorted tuple $\bar{u}$ from its active domain,
$$D \models \mathrm{Pre}_a(\bar{u}) \quad \Longleftrightarrow \quad a(\bar{u}) \in \Gamma_c(D) \cup \Gamma_u(D).$$
2. For every first-order sentence ϕ, one can effectively construct a first-order formula $\mathrm{WP}_a(\phi)(\bar{x})$ such that, for every state D and every $a(\bar{u}) \in \Gamma_c(D) \cup \Gamma_u(D)$,
$$D \models \mathrm{WP}_a(\phi)(\bar{u}) \quad \Longleftrightarrow \quad \forall D' \in \tau(D, a(\bar{u}))\,.\; D' \models \phi.$$

Thus Pre_a gives a first-order description of the enabled groundings already determined by Γ_c or Γ_u, while $\mathrm{WP}_a(\phi)$ describes those groundings whose every possible outcome satisfies ϕ. The definition does not prescribe how the transition semantics must be represented; it requires only that these formulas can be constructed effectively. For functional relational updates, this is the first-order preimage construction used in data-aware verification (Calvanese et al., 2020).

How `RelShield` constructs WP**.** `RelShield` instantiates this construction using the symbolic relational effects in its environment encodings. Each effect branch gives, for every relation R, a first-order successor-state formula $\rho_{a,e,R}(\bar{y}; \bar{x}, \bar{z})$ such that

$$R'(\bar{y}) \quad \Longleftrightarrow \quad \rho_{a,e,R}(\bar{y}; \bar{x}, \bar{z}),$$

where $\bar{x}$ are the call arguments and $\bar{z}$ collects any values selected by that branch. Choices from the current active domain remain ordinary active-domain variables; write $\bar{z}_f$ for the components that are fresh. The branch conditions and choices enumerate exactly the outcomes of the transition semantics (Appendix C).

The only additional bookkeeping comes from our active-domain semantics. Let C denote the finite set of constants occurring in the relational specification. For a fixed branch, write $\exists^+$ and $\forall^+$ for quantification over the finite temporary domain consisting of the current active domain, C, the call arguments, and the fresh placeholders $\bar{z}_f$. This is only a macro: it expands an active-domain quantifier together with finitely many substitutions for the additional named values. The active domain of the successor state is then first-order definable from the successor-state formulas. For a sort S, let

$$\mathsf{Adom}'_S(y) \;\equiv\; \bigvee_{R} \; \bigvee_{i:\,\mathrm{sort}(R,i)=S} \exists^+ \bar{y}_{-i}.\, \rho_{a,e,R}\big(\bar{y}[i \mapsto y]; \bar{x}, \bar{z}\big),$$

where the disjunction ranges over relation positions of sort S. Regression replaces each successor atom $R'(\bar{t})$ by $\rho_{a,e,R}(\bar{t}; \bar{x}, \bar{z})$, commutes with Boolean connectives, and relativizes post-state quantifiers to this successor active domain; for example,

$$\mathrm{Reg}_{a,e}(\exists y{:}S.\, \chi) = \exists^+ y{:}S.\, \big(\mathsf{Adom}'_S(y) \wedge \mathrm{Reg}_{a,e}(\chi)\big),$$

with the dual rule for universal quantification.

Lemma B.2 (Exact regression of relational effects)**.** *For a fixed effect branch and admissible choices producing successor D', the translation above satisfies, for every first-order sentence ϕ,*

$$D \models \mathrm{Reg}_{a,e}(\phi) \quad \Longleftrightarrow \quad D' \models \phi.$$

After eliminating the finite fresh placeholders and universally combining the possible branches and choices, the resulting formula is $\mathrm{WP}_a(\phi)$ of Definition B.1(2).

Proof. The first claim is by structural induction on ϕ. Relation atoms follow from the successor-state definitions, equality and Boolean connectives are immediate, and the quantified case follows because Adom'_S defines exactly $\mathsf{ad}_S(D')$. Fresh identifiers are opaque in the relational encodings, so admissible choices for $\bar{z}_f$ differ only by a C-preserving renaming. The fresh placeholders can therefore be eliminated using their equality and freshness constraints, yielding a formula in the original active-domain first-order language. This is the pure-equality instance of the data-aware preimage and quantifier-elimination construction (Calvanese et al., 2020). Conjoining the regressions of all possible outcomes gives the required universal preimage. □

For example, suppose `resume_line`(c, l) makes line l active. Its relevant effect can be written as

$$\mathsf{LineStatus}'(x, q) \leftrightarrow (x = l \wedge q = \mathsf{Active}) \vee (x \neq l \wedge \mathsf{LineStatus}(x, q)),$$

where the primed relation denotes the successor state and the unprimed relation denotes the current state. Let

$$\mathsf{Debt}(c) \equiv \exists b.\, \mathsf{OwnsBill}(c, b) \wedge \mathsf{BillStatus}(b, \mathsf{Overdue})$$

and let the requirement be

$$\psi_{\mathrm{bill}} = \forall c', l'.\, \big(\mathsf{OwnsLine}(c', l') \wedge \mathsf{LineStatus}(l', \mathsf{Active}) \rightarrow \neg \mathsf{Debt}(c')\big).$$

When ψ_{bill} is interpreted in the successor state, its $\mathsf{LineStatus}$ atom refers to the primed relation. Regression therefore replaces

$$\mathsf{LineStatus}'(l', \mathsf{Active})$$

by

$$l' = l \vee \big(l' \neq l \wedge \mathsf{LineStatus}(l', \mathsf{Active})\big).$$

The other relations in ψ_{bill} are unchanged by the tool, so the resulting pre-state formula is

$$\mathrm{WP}_{\texttt{resume_line}}(\psi_{\text{bill}})(c,l) = \forall c', l'.\, \Big(\mathsf{OwnsLine}(c', l') \wedge \big(l' = l \vee (l' \neq l \wedge \mathsf{LineStatus}(l', \mathsf{Active}))\big) \rightarrow \neg \mathsf{Debt}(c')\Big).$$

This formula is generated from the tool effect and the requirement, not from a particular state. The state enters only when the formula is evaluated, e.g. $D \models \mathrm{WP}_{\texttt{resume_line}}(\psi_{\text{bill}})(\bar{u})$. Creation and deletion require no separate rule: a created identifier is included in Adom'_S whenever it occurs in a successor fact, while a deleted identifier is excluded once no successor fact contains it.

B.3 Symbolic Synthesis

We now prove the symbolic-synthesis result of Section 4.1. Let

$$X_0 = Q_\psi, \qquad X_{k+1} = F_\psi(X_k),$$

and let ϕ_k be defined by equation 4, with G_a^k as in Section 4.1..

Theorem 3 (Exact symbolic synthesis, restated). *Every φ_k is effectively constructible and satisfies*

$$X_k = \{D : D \models \varphi_k\}.$$

If $\varphi_k \equiv \varphi_{k+1}$ for some k, then $X_k = W$, and G_a^k defines $S^\star$ for every $a \in \mathcal{A}_c$.

Proof. The first claim is by induction on k. For $k = 0$, $\varphi_0 = \psi$, so $\{D : D \models \varphi_0\} = Q_\psi = X_0$.

Suppose φ_k defines X_k. By Definition B.1, for every grounding $m = a(\bar{u})$,

$$D \models \mathrm{WP}_a(\varphi_k)(\bar{u}) \quad \Longleftrightarrow \quad \tau(D, m) \subseteq X_k,$$

while $\mathrm{Pre}_a(\bar{u})$ characterizes whether $m \in \Gamma_c(D) \cup \Gamma_u(D)$. Hence equation 4 expresses exactly the controllable and uncontrollable clauses of $F_\psi(X_k)$, and therefore

$$\{D : D \models \varphi_{k+1}\} = F_\psi(X_k) = X_{k+1}.$$

Effectiveness follows from Definition B.1.

Now suppose $\varphi_k \equiv \varphi_{k+1}$. Then $X_k = F_\psi(X_k)$, so $X_k \subseteq W$ because W is the greatest fixed point of F_ψ. Conversely, $W \subseteq X_0$, and monotonicity of F_ψ gives $W \subseteq X_j$ for every j. Hence $X_k = W$. Substituting W for X_k in G_a^k gives exactly the definition of $S^\star$. □

B.4 Complete Synthesis

This subsection proves the finite-abstraction result of Section 4.2. We make precise the renaming assumptions used there, show that the winning-region computation can be restricted to the safe-reachable states, and prove that a sufficiently large finite universe preserves both the winning region and the maximally permissive shield.

Fix a finite set $C \subseteq U$ containing every constant occurring in ψ or in the finite description of the environment.

Renaming assumptions. We adopt from STEAD (Mercado & Lomuscio, 2026, Definition 3) the notions of C-isomorphism and consistent renaming of transition pairs. Thus $\gamma : D \simeq_C E$ is a sort-preserving bijection that fixes C pointwise and preserves and reflects relational facts, and $D \oplus D' \simeq_C E \oplus E'$ means that the same renaming is used consistently for the pre- and post-states. We apply γ componentwise to grounded calls.

For shield synthesis, the renaming must preserve the action structure that appears in F_ψ: the controllable and uncontrollable enabled calls and the possible outcomes of each grounded call.

Definition B.3 (C-equivariant enabled calls)**.** The enabled-call maps of an environment are C-equivariant if, for every $\gamma : D \simeq_C E$,

$$\gamma(\Gamma_c(D)) = \Gamma_c(E), \qquad \gamma(\Gamma_u(D)) = \Gamma_u(E).$$

We also require τ to be C-tool-uniform, so that corresponding grounded calls have corresponding outcome sets under the same renaming (Mercado & Lomuscio, 2026, Definition 10). For first-order relational specifications whose constants lie in C, these conditions follow from invariance under isomorphism (Belardinelli et al., 2012); the relational encodings used by `RelShield` satisfy them.

The finite construction also needs enough unused values to match one successor step. Because the boundedness hypothesis of Theorem 4 applies only to $\mathcal{R}_\psi(I)$, we state this requirement separately as a per-step bound on newly introduced values.

Definition B.4 (Per-step fresh-value bound)**.** Let $\mathcal{E} = \langle I, \Gamma_c, \Gamma_u, \tau \rangle$ be an environment and let $C \subseteq U$ be finite. For a sort S, we say that $f_S \in \mathbb{N}$ is a per-step fresh-value bound if, for every state D, every $m \in \Gamma_c(D) \cup \Gamma_u(D)$, and every $D' \in \tau(D, m)$,

$$\big|\big(\mathsf{ad}(D') \setminus (\mathsf{ad}(D) \cup C)\big) \cap U_S\big| \leq f_S.$$

Lemma B.5 (Safe-reachable restriction)**.** *Let $\mathcal{E}$ be an environment and let ψ be a requirement, let $\mathcal{R} = \mathcal{R}_\psi(I)$, and define*

$$F_{\mathcal{R}}(X) = \mathcal{R} \cap F_\psi(X), \qquad W_{\mathcal{R}} = \nu X.F_{\mathcal{R}}(X).$$

Then

$$W_{\mathcal{R}} = W \cap \mathcal{R}.$$

Moreover, for every $D \in \mathcal{R}$,

$$S^\star(D) = \{a \in \Gamma_c(D) : \tau(D, a) \subseteq W_{\mathcal{R}}\}.$$

Proof. If $D \in W \cap \mathcal{R}$, every successor required by $W = F_\psi(W)$ lies in $W \subseteq Q_\psi$ and is therefore again in $\mathcal{R}$. Hence $W \cap \mathcal{R}$ is post-fixed for $F_{\mathcal{R}}$, so $W \cap \mathcal{R} \subseteq W_{\mathcal{R}}$. Conversely, every post-fixed point of $F_{\mathcal{R}}$ is contained in $\mathcal{R}$ and is also post-fixed for F_ψ, hence is contained in W. This proves the first claim.

For $D \in \mathcal{R}$, every successor in W satisfies ψ and is a one-step successor of a safe-reachable state, hence also lies in $\mathcal{R}$. Thus $\tau(D, a) \subseteq W$ iff $\tau(D, a) \subseteq W_{\mathcal{R}}$, which proves the second claim. □

Definition B.6 (Finite restriction)**.** Let $\mathcal{E} = \langle I, \Gamma_c, \Gamma_u, \tau \rangle$ be an environment, let $C \subseteq U$ be finite, and for every sort S let $U_S^f \subseteq U_S$ be finite with $C \cap U_S \subseteq U_S^f$. Put $U^f = \biguplus_S U_S^f$, and assume that every $D_0 \in I$ has a C-isomorphic representative over U^f. Let I_f contain one such representative for each D_0.

The finite restriction of $\mathcal{E}$ to U^f is

$$\mathcal{E}_f = \langle I_f, \Gamma_c^f, \Gamma_u^f, \tau_f \rangle,$$

where, for every state D over U^f and every $m \in \Gamma_c(D) \cup \Gamma_u(D)$,

$$\tau_f(D, m) = \tau(D, m) \cap \mathcal{D}_\Sigma(U^f),$$

and, for $i \in \{c, u\}$,

$$\Gamma_i^f(D) = \{m \in \Gamma_i(D) : \tau_f(D, m) \neq \varnothing\}.$$

This restriction differs from the one used for STEAD verification in one important respect: initial identifiers outside C need not be retained literally.

Lemma B.7 (One-step finite matching)**.** *Let $(\mathcal{E}, \psi)$ be an instance and let $C \subseteq U$ be finite. Assume that the enabled-call maps are C-equivariant and that τ is C-tool-uniform. For every sort S, let f_S be a per-step fresh-value bound and let $n_S \in \mathbb{N}$ satisfy*

$$|(\mathsf{ad}(D) \setminus C) \cap U_S| \leq n_S \qquad \textit{for every } D \in \mathcal{R}_\psi(I).$$

Let $\mathcal{E}_f$ be the finite restriction of $\mathcal{E}$ to universes U_S^f satisfying

$$|U_S^f| \geq |C \cap U_S| + n_S + f_S \qquad \textit{for every sort } S,$$

and write $\mathcal{R}_f$ for its safe-reachable set.

If $D \in \mathcal{R}_\psi(I)$, $\bar{D} \in \mathcal{R}_f$, and $\gamma : D \simeq_C \bar{D}$, then γ maps $\Gamma_c(D)$ bijectively onto $\Gamma_c^f(\bar{D})$ and $\Gamma_u(D)$ bijectively onto $\Gamma_u^f(\bar{D})$. Moreover, for every corresponding call m and $\gamma(m)$:

1. *every* $D' \in \tau(D, m)$ *has some* $\bar{D}' \in \tau_f(\bar{D}, \gamma(m))$ *such that*
$$D \oplus D' \simeq_C \bar{D} \oplus \bar{D}';$$
2. *every* $\bar{D}' \in \tau_f(\bar{D}, \gamma(m))$ *has some* $D' \in \tau(D, m)$ *satisfying the same correspondence.*

Proof. Let $\gamma : D \simeq_C \bar{D}$. In sort S, the image of $\mathsf{ad}(D) \cup C$ occupies at most $n_S + |C \cap U_S|$ values of U_S^f. An outcome D' introduces at most f_S additional values outside $\mathsf{ad}(D) \cup C$. Hence the universe bound leaves enough unused values to extend γ to the values occurring in D'. Let $\bar{D}'$ be the image of D' under this extension. Then
$$D \oplus D' \simeq_C \bar{D} \oplus \bar{D}',$$
and C-tool-uniformity gives $\bar{D}' \in \tau_f(\bar{D}, \gamma(m))$.

Applying the same argument to γ^{-1} gives the reverse matching. C-equivariance gives the bijections of controllable and uncontrollable enabled calls, and the forward matching ensures that every corresponding enabled call retains an outcome in the finite restriction. □

Under the standing renaming assumptions above, we can now restate the finite abstraction theorem.

Theorem 4 (Finite Abstraction, restated). *Assume that for every sort S there exists a corresponding finite bound n_S such that $|(\mathsf{ad}(D) \setminus C) \cap U_S| \leq n_S$ for every $D \in \mathcal{R}_\psi(I)$. Let f_S be the maximum number of fresh sort-S objects that any single action may create, and let $\mathcal{E}_f$ be the same environment interpreted over finite universes U_S^f satisfying $|U_S^f| \geq |C \cap U_S| + n_S + f_S$ for every sort S, with C-isomorphic representatives of the initial states.*

Then, the safe reachable set under $\mathcal{E}_f$ is finite, and every $D \in \mathcal{R}_\psi(I)$ has a C-isomorphic representative $\bar{D}$ in that set. Let $\bar{W}$ and $\bar{S}^\star$ be obtained by solving equation 5 over $\mathcal{E}_f$. For every such pair and every witnessing isomorphism γ, we have
$$D \in W \iff \bar{D} \in \bar{W}, \qquad \gamma\big(S^\star(D)\big) = \bar{S}^\star(\bar{D}).$$

Proof. The finite interpretation in the theorem is the finite restriction of Definition B.6. Its safe-reachable set is finite because the schema and all universes U_S^f are finite. Because the initial states need only have C-isomorphic representatives, the induction starts from the corresponding pair $D_0 \simeq_C \bar{D}_0$, rather than from an identical initial state.

Every $D \in \mathcal{R}_\psi(I)$ has a representative in the finite safe-reachable set. Follow a safe path from an initial state to D and apply Lemma B.7 at each step. Since every constant of ψ lies in C, satisfaction of ψ is preserved under C-isomorphism, so the matched path remains safe.

Write $\mathcal{R} = \mathcal{R}_\psi(I)$ and let $\mathcal{R}_f$ be the finite safe-reachable set. Let F_f be the restricted winning operator of equation 5 over $\mathcal{E}_f$, so $\bar{W} = \nu X.F_f(X)$. Let
$$Y = \{D \in \mathcal{R} : D \simeq_C \bar{D} \text{ for some } \bar{D} \in \bar{W}\}.$$
By Lemma B.7, the controllable and uncontrollable clauses of the fixed-point equation for $\bar{W}$ transfer to Y. Hence $Y \subseteq F_{\mathcal{R}}(Y)$, so $Y \subseteq W_{\mathcal{R}}$. Conversely, the states in $\mathcal{R}_f$ that are C-isomorphic to a state of $W_{\mathcal{R}}$ form a post-fixed point of F_f, again by Lemma B.7, and are therefore contained in $\bar{W}$. Thus, for every C-isomorphic pair $D \in \mathcal{R}$ and $\bar{D} \in \mathcal{R}_f$,
$$D \in W_{\mathcal{R}} \quad \iff \quad \bar{D} \in \bar{W}.$$
Lemma B.5 gives $W_{\mathcal{R}} = W \cap \mathcal{R}$, proving the winning-region equivalence in the theorem.

Finally, let $\gamma : D \simeq_C \bar{D}$. By Lemma B.7, a and $\gamma(a)$ are enabled together and their outcomes match in both directions. Using the winning-region equivalence,
$$\tau(D, a) \subseteq W \quad \iff \quad \tau_f(\bar{D}, \gamma(a)) \subseteq \bar{W}.$$
Therefore
$$a \in S^\star(D) \quad \iff \quad \gamma(a) \in \bar{S}^\star(\bar{D}),$$
and hence
$$\gamma\big(S^\star(D)\big) = \bar{S}^\star(\bar{D}).$$
□

B.5 Boundedness Certification

The finite synthesis procedure requires two kinds of bounds. The per-step fresh-value bounds f_S are given by Definition B.4. This subsection gives sufficient conditions for certifying the safe-reachable bounds n_S required by Theorem 4. These are the boundedness obligations used by the finite branch of `RelShield` described in the main text.

For a sort S and $n \in \mathbb{N}$, define

$$\mathsf{Cap}_{S,n} = \neg\exists x_0 \cdots x_n.\ \left(\bigwedge_{i=0}^{n} \bigwedge_{c \in C \cap U_S} x_i \neq c \ \wedge \bigwedge_{0 \le i < j \le n} x_i \neq x_j \right),$$

where $x_0, \ldots, x_n$ have sort S. For a family of bounds $\mathbf{n} = (n_S)_S$, let

$$\mathsf{Cap}_{\mathbf{n}} = \bigwedge_S \mathsf{Cap}_{S,n_S}.$$

Under the active-domain semantics of Appendix A.1, $\mathsf{Cap}_{S,n}$ states that at most n active values of sort S lie outside C.

Lemma B.8 (Boundedness certificate)**.** *Let $\mathcal{E} = \langle I, \Gamma_c, \Gamma_u, \tau \rangle$ be an environment, let ψ be a requirement, and let $\mathbf{n} = (n_S)_S$ be a family of finite bounds. Either of the following conditions is sufficient to guarantee*

$$|(\mathsf{ad}(D) \setminus C) \cap U_S| \le n_S$$

for every sort S and every $D \in \mathcal{R}_\psi(I)$.

1. $$\psi \models \mathsf{Cap}_{\mathbf{n}}.$$

2. *Every initial state satisfies $\mathsf{Cap}_{\mathbf{n}}$ and, for every $a \in \mathcal{A}_c \cup \mathcal{A}_u$,*

$$\psi \wedge \mathsf{Cap}_{\mathbf{n}} \wedge \mathrm{Pre}_a(\bar{x}) \ \models\ \mathrm{WP}_a(\neg\psi \vee \mathsf{Cap}_{\mathbf{n}})\,(\bar{x}).$$

Proof. Under the first condition, every state in $\mathcal{R}_\psi(I)$ satisfies ψ and therefore satisfies $\mathsf{Cap}_{\mathbf{n}}$. The claimed bounds follow from the definition of $\mathsf{Cap}_{\mathbf{n}}$.

For the second condition, consider any path from an initial state whose states all satisfy ψ. We prove by induction along the path that every state satisfies $\mathsf{Cap}_{\mathbf{n}}$. The initial state does so by assumption.

Suppose the current state satisfies both ψ and $\mathsf{Cap}_{\mathbf{n}}$. For any enabled controllable call or uncontrollable action, the displayed entailment and the defining property of WP_a imply that every successor either violates ψ or satisfies $\mathsf{Cap}_{\mathbf{n}}$. A successor belonging to a safe path satisfies ψ, so it must satisfy $\mathsf{Cap}_{\mathbf{n}}$. Hence every state in $\mathcal{R}_\psi(I)$ satisfies $\mathsf{Cap}_{\mathbf{n}}$, which gives the claimed bounds. □

C Supplement to the benchmark-derived census

This section specifies the benchmark-derived environments and requirements used in the main census of Section 5. For each domain we give the relational state retained by the model, the controllable and uncontrollable transitions over that state, the initial family, and the standing requirements paired with the environment. An *instance* consists of one environment and one requirement.

The models are constructed from benchmark application data and implementations. We retain the relations needed to state the selected requirements and to determine the enabledness and effects of the modeled calls. Any projection of state, restriction of the call interface, or initial-state construction is stated in the corresponding domain subsection. Requirements are formalized from benchmark policy or knowledge documents, implementation and schema constraints, or explicitly authored invariants over the modeled state.

Each domain ends with two grids over the environments and requirements defined immediately above it. The left grid reports the synthesis outcome; the right reports the synthesized shield class for each solved instance. Blank cells denote inapplicable environment–requirement pairs.



Notation. Controllable calls are the calls available to the agent before shielding; uncontrollable actions occur independently of the shield. Requirements are closed first-order formulas over the relational state. Benchmark-specific names are retained where they identify source objects or APIs. A *fixed* sort has exactly the finite carrier listed in its description. Its values are treated as environment constants that belong to C in the finite abstraction. An *opaque* sort is not exhaustively enumerated and its universe is potentially infinite, and identifiers not fixed by the specification may be renamed.

C.1 TELECOM (τ^2-BENCH)

Modeled fragment. We model the customer, line, and bill relations used by the selected Telecom policies together with the corresponding `TelecomTools` operations. Numeric data usage, monetary amounts, plan prices, and payment balances are outside this fragment because no modeled call reads or writes them; the tools whose semantics depend on those quantities are omitted.

Sorts. Opaque: c: customer, l: line, b: bill, p: plan, d: device, ph: phone. Fixed: line status $\{\mathsf{Active}, \mathsf{Suspended}, \mathsf{Pending}, \mathsf{Closed}\}$; bill status $\{\mathsf{Draft}, \mathsf{Issued}, \mathsf{Awaiting}, \mathsf{Paid}, \mathsf{Overdue}, \mathsf{Disputed}\}$; account status $\{\mathsf{Active}, \mathsf{Suspended}, \mathsf{PendingVerification}, \mathsf{Closed}\}$; date, the 19 dates occurring in the benchmark data together with the benchmark's constant "today", 2025-02-25. The only date any tool writes is today, so dates are values with a fixed order rather than a clock.

Schema.

Customer	$\mathit{AccountStatus}(c, \cdot)$, $\mathit{CustomerPhone}(c, ph)$, $\mathit{OwnsLine}(c, l)$, $\mathit{OwnsBill}(c, b)$.
Line	$\mathit{LineStatus}(l, \cdot)$, $\mathit{LinePlan}(l, p)$, $\mathit{LineDevice}(l, d)$, $\mathit{LinePhone}(l, ph)$, $\mathit{RoamingEnabled}(l)$, $\mathit{ContractEnd}(l, \mathit{date})$, $\mathit{SuspensionStart}(l, \mathit{date})$, $\mathit{LastPlanChange}(l, \mathit{date})$, $\mathit{LastSimReplacement}(l, \mathit{date})$.
Bill	$\mathit{BillStatus}(b, \cdot)$, $\mathit{BillIssued}(b, \mathit{date})$, $\mathit{BillDue}(b, \mathit{date})$.
Catalogue	$\mathit{Plan}(p)$, $\mathit{Device}(d)$.
Clock	$\mathit{DateBefore}(\mathit{date}, \mathit{date}')$, $\mathit{Today}(\mathit{date})$; frozen by every tool.

The status relations are total and functional on the objects that exist, so "the customer exists" is $\mathit{Cust}(c) := \exists a.\, \mathit{AccountStatus}(c, a)$, and likewise $\mathit{Line}(l)$ and $\mathit{Bill}(b)$ through $\mathit{LineStatus}$ and $\mathit{BillStatus}$.

Derived predicates.

$$\begin{aligned} \mathit{LineOf}(c, l) &:= \mathit{Cust}(c) \wedge \mathit{OwnsLine}(c, l) \wedge \mathit{Line}(l), \\ \mathit{Past}(d) &:= \exists t.\, \mathit{Today}(t) \wedge \mathit{DateBefore}(d, t), \\ \mathit{HasBill}(c, S) &:= \exists b.\, \mathit{OwnsBill}(c, b) \wedge \mathit{BillStatus}(b, S), \\ \mathit{BillsResolve}(c) &:= \forall b.\, \mathit{OwnsBill}(c, b) \rightarrow \mathit{Bill}(b). \end{aligned}$$

Tool semantics. Fifteen agent tools; the first five write.

Tool	Enabled when	Effect
`suspend_line`(c, l)	$\mathit{LineOf}(c, l) \wedge \mathit{LineStatus}(l, \mathsf{Active})$	$\mathit{LineStatus}(l) := \mathsf{Suspended}$; $\mathit{SuspensionStart}(l) := \mathit{today}$
`resume_line`(c, l)	$\mathit{LineOf}(c, l) \wedge (\mathit{LineStatus}(l, \mathsf{Suspended}) \vee \mathit{LineStatus}(l, \mathsf{Pending}))$	$\mathit{LineStatus}(l) := \mathsf{Active}$; $-\mathit{SuspensionStart}(l, \cdot)$
`send_payment_request`(c, b)	$\mathit{Cust}(c) \wedge \mathit{BillsResolve}(c) \wedge \neg \mathit{HasBill}(c, \mathsf{Awaiting}) \wedge \mathit{OwnsBill}(c, b) \wedge \mathit{Bill}(b)$	$\mathit{BillStatus}(b) := \mathsf{Awaiting}$
`enable_roaming`(c, l)	$\mathit{LineOf}(c, l)$	$+\mathit{RoamingEnabled}(l)$
`disable_roaming`(c, l)	$\mathit{LineOf}(c, l)$	$-\mathit{RoamingEnabled}(l)$

Tool	Enabled when	Effect
`get_customer_by_id`(c)	$Cust(c)$	read
`get_customer_by_phone`(ph)	$\exists c.\ CustomerPhone(c, ph) \vee \exists c, l.\ OwnsLine(c, l) \wedge LinePhone(l, ph)$	read
`get_bills_for_customer`(c)	$Cust(c) \wedge BillsResolve(c)$	read
`get_data_usage`(c, l)	$LineOf(c, l) \wedge \exists p.\ LinePlan(l, p) \wedge Plan(p)$	read
`get_details_by_id_line`(l)	$Line(l)$	read
`get_details_by_id_bill`(b)	$Bill(b)$	read
`get_details_by_id_customer`(c)	$Cust(c)$	read
`get_details_by_id_device`(d)	$Device(d)$	read
`get_details_by_id_plan`(p)	$Plan(p)$	read
`transfer_to_human_agents`()	true	read

Uncontrollable actions. One, native: the simulated user's payment, which the benchmark applies to the server outside the agent's tools.

Environment action	Fires when	Effect
`payment_received`(b)	$BillStatus(b, \mathsf{Awaiting})$	$BillStatus(b) := \mathsf{Paid}$

Initial families. *C1002*: we retain customer C1002's benchmark-provided component, including its five lines with their plans, devices, and phone numbers, together with its bills. C1002 is the smallest benchmark-provided customer for which both the overdue-bill and line-status relations are non-vacuous. In the benchmark state, C1002 has an overdue bill (B1005) and five active lines. To obtain an initial state satisfying the standing requirement, we initialize those lines as suspended, consistent with the benchmark policy governing service for customers with overdue bills.

Environments.

Environment	Initial family	Agent tools	Environment
C1002, all 15 tools	C1002	the 15 tools	none
C1002 + payment event	C1002	the 15 tools	`payment_received`

Requirements. Five formalizations of Telecom policy clauses. The first is the state-invariant formulation used in Figure 1; it strengthens the source rule that service must not be resumed while an overdue bill remains unpaid.

- **no active line with an overdue bill** (`no_service_with_overdue_bill`): $\forall c, l.\ OwnsLine(c, l) \wedge LineStatus(l, \mathsf{Active}) \rightarrow \neg HasBill(c, \mathsf{Overdue})$.
- **no active line past contract end** (`no_service_past_contract_end`): $\forall l, d.\ \neg(LineStatus(l, \mathsf{Active}) \wedge ContractEnd(l, d) \wedge Past(d))$.
- **both of the above** (`suspension_discipline`): the conjunction of the two.
- **one bill awaiting payment** (`at_most_one_awaiting_payment`): $\forall c.\ \exists^{\leq 1} b.\ OwnsBill(c, b) \wedge BillStatus(b, \mathsf{Awaiting})$.
- **every suspension has a cause** (`suspension_has_cause`): $\forall l.\ LineStatus(l, \mathsf{Suspended}) \rightarrow (\exists c.\ OwnsLine(c, l) \wedge HasBill(c, \mathsf{Overdue})) \vee \exists d.\ ContractEnd(l, d) \wedge Past(d)$.

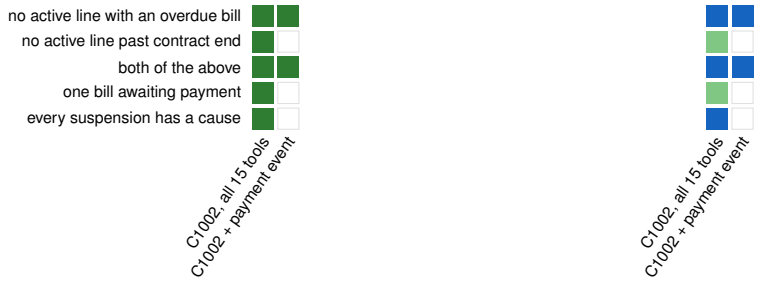


Figure 2: Telecom environment–requirement pairs: synthesis outcome (left) and synthesized shield class (right).

C.2 AIRLINE (τ^2-BENCH)

Modeled fragments. We use two relational fragments of the reservation domain. The *native* fragment retains the state needed by the cancellation requirements and includes the cancellation tool together with three read-only tools. The *projected* fragment additionally retains itinerary endpoints, membership tier, payment-method kinds, baggage and passenger counts, and minted certificates, enabling the modeled baggage, flight, passenger, and certificate operations. A payment-extended native fragment adds reservation payment methods for the payment-source requirement. Prices, balances, certificate amounts, passenger names, and dates of birth are omitted where they do not affect a retained call. We omit booking because determining whether the call is enabled requires reproducing the benchmark's exact price calculation.

Sorts. Opaque: r: reservation, f: flight, u: user, p: passenger (a line of the reservation's passenger list, not a person), t: instant. Fixed: flight date, the dates the loaded flights run on; flight status $\{\mathsf{available}, \mathsf{ontime}, \mathsf{flying}, \mathsf{landed}, \mathsf{cancelled}, \mathsf{delayed}\}$; cabin $\{\mathsf{basic}, \mathsf{economy}, \mathsf{business}\}$. The projected model adds the opaque m: payment method and a: airport, and the fixed kind $\{\mathsf{credit}, \mathsf{gift}, \mathsf{certificate}\}$, tier $\{\mathsf{regular}, \mathsf{silver}, \mathsf{gold}\}$, bag count $\{\mathsf{B0}, \ldots, \mathsf{B}k\}$ with k declared per environment, passenger count $\{\mathsf{P1}, \ldots, \mathsf{P4}\}$, and certificate slot, the three certificate identifiers the benchmark's `send_certificate` can ever mint. The payment-extended fragment adds the opaque m: payment method and the fixed source $\{\mathsf{credit}, \mathsf{gift}, \mathsf{certificate}\}$.

Schema.

Native	$ResUser(r,u)$, $ResCabin(r,\cdot)$, $ResSegment(r,f,date)$, $ResPassenger(r,p)$, $Cancelled(r)$, $ResInsurance(r)$, $CreatedAt(r,t)$, $InstantBefore(t,t')$, $FlightStatus(f,date,\cdot)$, $Flight(f)$.
Projected additions	$ResOrigin(r,a)$, $ResDest(r,a)$, $FlightOrigin(f,a)$, $FlightDest(f,a)$, $UserPayment(u,m)$, $PayKind(m,\cdot)$, $Member(u,\cdot)$, $MintedCert(u,\cdot)$, $TotalBags(r,\cdot)$, $NonFreeBags(r,\cdot)$, $PassCount(r,\cdot)$, $BagsOK(tier,cabin,np,n,n')$, $EnoughSeats(f,date,cabin,r)$.
Payment-extended fragment additions	$ResPayment(r,m)$, $PayMethodSource(m,\cdot)$.

$BagsOK$ is the policy's free-bag allowance table: it holds when n is at most the free allowance of the tier and cabin per passenger times np plus n'. $EnoughSeats$ compares the flight's free seats with the reservation's passenger count. Both relations are frozen by every modeled tool.

Derived predicates.

$$
\begin{aligned}
Res(r) &:= \exists u.\, ResUser(r,u),\\
User(u) &:= \exists r.\, ResUser(r,u),\\
Flown(f,d) &:= FlightStatus(f,d,\mathsf{flying}) \vee FlightStatus(f,d,\mathsf{landed}),\\
Usable(u,m) &:= UserPayment(u,m) \wedge \neg PayKind(m,\mathsf{certificate}) \wedge \neg PayKind(m,\mathsf{gift}).
\end{aligned}
$$

Gift-card payments are outside the modeled set because their acceptance compares two balances.

Tool semantics. The native menu is the first four rows; the projected menu adds the other nine.

Tool	Enabled when	Effect
`cancel_reservation`(r)	$Res(r)$	$+Cancelled(r)$
`list_all_airports`()	true	read
`get_reservation_details`(r)	$Res(r)$	read
`get_user_details`(u)	$User(u)$	read
`search_direct_flight`(), `search_onestop_flight`(), `get_flight_status`(), `calculate`(), `transfer_to_human_agents`()	true	read (the native arguments shape only the returned value)
`update_reservation_passengers`(r)	$Res(r)$	read (the native tool refuses any change to the passenger count; names are not modeled)
`send_certificate`(u)	$User(u) \wedge \exists s.\, \neg MintedCert(u, s)$	$+MintedCert(u, s)$ for the first free slot s of the three
`update_reservation_flights`(r, cab, f, d, m)	$Res(r) \wedge \exists u.\, ResUser(r, u) \wedge Usable(u, m) \wedge \big((ResSegment(r, f, d) \wedge ResCabin(r, cab)) \vee (FlightStatus(f, d, \mathsf{available}) \wedge EnoughSeats(f, d, cab, r))\big)$	$ResSegment(r, \cdot, \cdot) := \{(f, d)\}$; $ResCabin(r) := cab$ (itineraries modeled at length one)
`update_reservation_baggages`(r, n, n', m)	$Res(r) \wedge \exists u.\, ResUser(r, u) \wedge Usable(u, m)$	$TotalBags(r) := n$; $NonFreeBags(r) := n'$

Uncontrollable actions. None; τ^2-bench Airline provides no autonomous transition.

Initial families. All are the benchmark-provided database projected onto the schema and restricted to reservations with their users and the flights their segments use. The selected records make the requirement-relevant relations present. For environments that exercise flight changes, we add candidate flights to which the reservation can be moved.

Four reservations	The two benchmark-provided reservations with a segment already flown, together with 4WQ150 and VAAOXJ.
Whole database	All 2,000 reservations and 500 users.
Common	I6M8JQ with candidate HAT001 and baggage counts up to B1.
Certificate	K1NW8N.
Baggage	T1QOBS with baggage counts up to B3.
Itinerary	T1QOBS with candidates HAT001 and HAT002 and baggage counts up to B2.
Two certificates	AQHB3C and 66NJCE with baggage counts up to B2.

No benchmark-provided reservation is cancelled and no benchmark-provided user holds a mintable certificate, so the cancellation and certificate clauses hold at every initial state and every violation is agent-caused.

Environments.

Environment	Initial family	Agent tools
4 reservations, 4 tools	four reservations	native menu (cancel and three reads)
payments, 4 reservations	four reservations, payment-extended fragment	native menu
payments, whole DB	whole database, payment-extended fragment	native menu
common / certificate / baggage / itinerary / two certificates, 13 tools	the projected family of that name	all 13 tools

No environment has an uncontrollable action.

Requirements. Seven formalizations of Airline policy clauses.

- **no cancellation after a flown segment** (`no_cancel_after_flown`): $\forall r.\ \mathit{Cancelled}(r) \rightarrow \neg\exists f, d.\ \mathit{Flight}(f) \wedge \mathit{ResSegment}(r, f, d) \wedge \mathit{Flown}(f, d)$.
- **cancellation only on stated grounds** (`cancellation_grounds`): $\forall r.\ \mathit{Cancelled}(r) \rightarrow \neg\exists f, d.\ (\mathit{Flight}(f) \wedge \mathit{ResSegment}(r, f, d) \wedge \mathit{Flown}(f, d)) \wedge \big(\mathit{AllCancelled}(r) \vee \mathit{Within24h}(r) \vee \mathit{ResCabin}(r, \mathsf{business}) \vee \mathit{ResInsurance}(r)\big)$, where $\mathit{AllCancelled}(r) := \neg\exists f, d.\ \mathit{Flight}(f) \wedge \mathit{ResSegment}(r, f, d) \wedge \neg\mathit{FlightStatus}(f, d, \mathsf{cancelled})$ and $\mathit{Within24h}(r) := \exists t.\ \mathit{CreatedAt}(r, t) \wedge \mathit{InstantBefore}(t_{24}, t)$ for the policy's cutoff instant t_{24}.
- **at most five passengers** (`at_most_five_passengers`): $\forall r.\ \exists^{\leq 5} p.\ \mathit{ResPassenger}(r, p)$.
- **payment-source limits** (`payment_source_limits`, payment-extended fragment): $\forall r.\ \exists^{\leq 1} m.\ (\mathit{ResPayment}(r, m) \wedge \mathit{PayMethodSource}(m, \mathsf{certificate})) \wedge \exists^{\leq 1} m.\ (\ldots \mathsf{credit}) \wedge \exists^{\leq 3} m.\ (\ldots \mathsf{gift})$.
- **itinerary endpoints preserved** (`itinerary_endpoints_preserved`, projected): $\forall r.\ \mathit{Res}(r) \rightarrow (\exists a, f, d.\ \mathit{ResOrigin}(r, a) \wedge \mathit{ResSegment}(r, f, d) \wedge \mathit{FlightOrigin}(f, a)) \wedge (\exists a, f, d.\ \mathit{ResDest}(r, a) \wedge \mathit{ResSegment}(r, f, d) \wedge \mathit{FlightDest}(f, a))$.
- **baggage within allowance** (`baggage_allowance_consistency`, projected): $\forall r, u, \mathit{tier}, \mathit{cab}, \mathit{np}, n, n'.\ \mathit{ResUser}(r, u) \wedge \mathit{Member}(u, \mathit{tier}) \wedge \mathit{ResCabin}(r, \mathit{cab}) \wedge \mathit{PassCount}(r, \mathit{np}) \wedge \mathit{TotalBags}(r, n) \wedge \mathit{NonFreeBags}(r, n') \rightarrow \mathit{BagsOK}(\mathit{tier}, \mathit{cab}, \mathit{np}, n, n')$.
- **certificate eligibility** (`certificate_eligibility`, projected): $\forall u.\ (\exists s.\ \mathit{MintedCert}(u, s)) \rightarrow \mathit{Member}(u, \mathsf{silver}) \vee \mathit{Member}(u, \mathsf{gold}) \vee \exists r.\ (\mathit{ResUser}(r, u) \wedge \mathit{ResInsurance}(r)) \vee \exists r.\ (\mathit{ResUser}(r, u) \wedge \mathit{ResCabin}(r, \mathsf{business}))$.

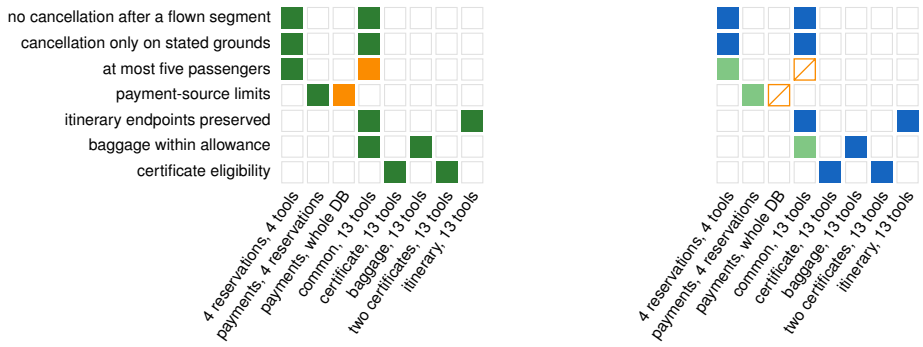


Figure 3: Airline environment–requirement pairs: synthesis outcome (left) and synthesized shield class (right).

C.3 Banking (τ^2-bench)

Modeled fragments. We use two Banking fragments. The *native* fragment retains the relations needed by the selected knowledge-base clauses. Because the modeled tools never change the monetary and age conditions used by these policies, we encode their truth values as fixed predicates such as $\mathit{Closable}$, $\mathit{Bal25}$, $\mathit{Aged14}$, and $\mathit{Aged30}$; transfer and credit operations are outside the modeled interface. The *dispute* fragment projects the initial state of benchmark task 084 onto its account, transactions, open dispute, and filing dates. The main-census version keeps the benchmark's current date fixed; advancing-time variants are specified separately in Appendix D.

Sorts (native fragment). Opaque: u: user, a: account, k: debit card, x: credit-card account, q: credit-limit request, o: replacement order. Fixed: account status $\{\mathsf{Open}, \mathsf{Closed}\}$; card status $\{\mathsf{Pending}, \mathsf{Active}, \mathsf{Frozen}, \mathsf{Closed}, \mathsf{Deactivated}\}$; account class

{checking, saving, bchecking, bsavings}; request status {Pending, Approved, Denied}; order status {Outstanding, Settled}; date.

Schema (native fragment).

Accounts	$Account(a)$, $AccOwner(a, u)$, $AccClass(a, \cdot)$, $AccStatus(a, \cdot)$, and the projected facts $Closable(a)$, $Bal25(a)$, $Aged14(a)$, $Aged30(a)$, $NegBal(a)$.
Debit cards	$Card(k)$, $CardAcc(k, a)$, $CardStatus(k, \cdot)$, $CardNew(k)$, $PendingOrder(a)$.
Credit cards	$CCAcct(x)$, $CCOwner(x, u)$, $CCStatus(x, \cdot)$, $CCOrder(o)$, $CCOrderAcct(o, x)$, $CCOrderStatus(o, \cdot)$.
Requests	$CliReq(q)$, $CliAcct(q, x)$, $CliStatus(q, \cdot)$, $CliDate(q, date)$, $CooldownAfter(date, date')$.

Derived predicate. $OpenAcc(u, cl) := \exists a.\, AccOwner(a, u) \wedge AccClass(a, cl) \wedge AccStatus(a, \mathsf{Open})$.

Tool semantics (native fragment). The eleven tools on the compiled menus.

Tool	Enabled when	Effect
`freeze_debit_card`(k)	$Card(k) \wedge CardStatus(k, \mathsf{Active})$	$CardStatus(k) := \mathsf{Frozen}$
`unfreeze_debit_card`(k)	$Card(k) \wedge CardStatus(k, \mathsf{Frozen}) \wedge \exists a.\, CardAcc(k, a) \wedge AccStatus(a, \mathsf{Open})$	$CardStatus(k) := \mathsf{Active}$
`close_debit_card`(k)	$Card(k) \wedge (CardStatus(k, \mathsf{Active}) \vee CardStatus(k, \mathsf{Pending}))$	$CardStatus(k) := \mathsf{Closed}$
`close_bank_account`(a)	$Account(a) \wedge \neg AccStatus(a, \mathsf{Closed}) \wedge Closable(a)$	$AccStatus(a) := \mathsf{Closed}$
`order_debit_card`(a, u)	$Account(a) \wedge AccClass(a, \mathsf{checking}) \wedge AccStatus(a, \mathsf{Open}) \wedge AccOwner(a, u) \wedge Bal25(a) \wedge \neg PendingOrder(a) \wedge \neg\exists k.\, CardAcc(k, a) \wedge CardStatus(k, \mathsf{Active})$	**new** $k{:}card$; $+Card(k)$, $+CardAcc(k, a)$, $CardStatus(k) := \mathsf{Pending}$, $+CardNew(k)$, $+PendingOrder(a)$
`activate_debit_card`(k)	$Card(k) \wedge CardNew(k) \wedge CardStatus(k, \mathsf{Pending}) \wedge \exists a.\, CardAcc(k, a) \wedge AccStatus(a, \mathsf{Open})$	$CardStatus(k) := \mathsf{Active}$; every other active card on the same account $:= \mathsf{Deactivated}$
`get_debit_cards_by_account`(a)	$Account(a)$	read
`open_bank_account`(u, cl)	$cl = \mathsf{checking} \vee (cl = \mathsf{saving} \wedge \exists a.\, AccOwner(a, u) \wedge AccClass(a, \mathsf{checking}) \wedge AccStatus(a, \mathsf{Open}) \wedge Aged14(a)) \vee (cl = \mathsf{bchecking} \wedge \neg\exists a.\, (AccOwner(a, u) \wedge AccStatus(a, \mathsf{Closed})) \wedge OpenAcc(u, \mathsf{checking})) \vee (cl = \mathsf{bsavings} \wedge \neg\exists a.\, (AccOwner(a, u) \wedge NegBal(a)) \wedge \exists a.\, AccOwner(a, u) \wedge AccClass(a, \mathsf{bchecking}) \wedge AccStatus(a, \mathsf{Open}) \wedge Aged30(a))$	**new** $a{:}account$; $+Account(a)$, $+AccOwner(a, u)$, $AccClass(a) := cl$, $AccStatus(a) := \mathsf{Open}$, $+Closable(a)$
`get_all_user_accounts`(u)	$\exists a.\, AccOwner(a, u)$	read
`close_cc_account`(x, u)	$CCAcct(x) \wedge CCOwner(x, u)$	$CCStatus(x) := \mathsf{Closed}$
`get_cli_history`(x)	$CCAcct(x)$	read

Sorts (dispute fragment). Opaque: a: account, t: transaction. Fixed: day, the calendar from the benchmark's fixed date 11/14/2025 to one day past the last deadline; transaction date, the dates of the raised transactions; level, the seven checking-account levels; tier {Entry, Mid, Premium, Elite}.

Schema (dispute fragment).

Transactions	$TxnAccount(t, a)$, $Raised(t)$, $Disputed(t)$, $OpenDispute(t)$, $TxnDate(t, \cdot)$.
Account policy	$AccountLevel(a, \cdot)$, $TierOf(level, tier)$.
Calendar	$ResolvesOn(t, day)$, $Today(day)$, $NextDay(day, day')$, $SixtyDaysPast(tdate, day)$.

The calendar relations are frozen in the main-census model; the controlled extensions add $NextDay$ where time advances and, for alternating turns, a nullary $AgentTurn$.

Tool semantics (dispute fragment). Two tools, with the clock frozen at the benchmark's date.

Tool	Enabled when	Effect
`file_dispute`(t)	$\neg Disputed(t)$	$+Disputed(t)$, $+OpenDispute(t)$
`get_current_time`()	true	read

Uncontrollable actions. None on either fragment.

Initial families. The native fragment uses one benchmark-provided user per requirement; the dispute fragment uses the task-084 state projected onto the requirement-relevant account and transactions.

Cards	User kj93a7b2e1, for the one-active-card clauses.
Accounts	User ar72c5d8e3, who holds exactly four personal checking accounts, for the four-accounts clause.
Credit cards	User e3f4a5b6c7, for the replacement clause.
Dispute	Task 084 projected onto the Light Blue account (Entry tier, cap two), with one open dispute expected to resolve on 12/22/2025 and the two transactions raised on it, dated 10/05/2025 and 11/11/2025. Their filing windows close on 12/04/2025 and 01/10/2026; today is 11/14/2025.

Environments.

Environment	Initial family	Agent tools	Environment
cards, with ordering	user kj93a7b2e1	freeze, unfreeze, close card, close account, order, activate, get cards	none
accounts, with opening	user ar72c5d8e3	open, close account, get accounts	none
credit cards	user e3f4a5b6c7	close cc account, get cli history	none
dispute, clock frozen	task 084	file, get time	none

Requirements. Four formulas yield five main-census requirement specifications: the one-active-card formula is evaluated once directly and once with a declared capacity hint for the finite procedure. The dispute requirement combines the documented per-tier cap with a standing obligation that a raised, unfiled transaction remain within the documented 60-day filing window. With the clock frozen in the main census, only the cap can become newly violated; the time component becomes operative in Appendix D.

- **one active card per account** (`at_most_one_active_card_per_account`): $\forall a.\ \exists^{\leq 1} k.\ CardAcc(k, a) \wedge CardStatus(k, \mathsf{Active})$. The **capped** variant uses the same requirement and differs only by an auxiliary capacity bound supplied to the finite synthesis procedure.
- **at most four checking accounts, capped** (`at_most_four_personal_checking+cap`): $\forall u.\ \exists^{\leq 4} a.\ (AccOwner(a, u) \wedge AccClass(a, \mathsf{checking}) \wedge AccStatus(a, \mathsf{Open})) \wedge \exists^{\leq 5} a.\ Account(a)$.
- **no closing with a pending replacement** (`no_close_cc_with_pending_replacement`): $\forall x.\ CCStatus(x, \mathsf{Closed}) \rightarrow \neg\exists o.\ CCOrderAcct(o, x) \wedge CCOrderStatus(o, \mathsf{Outstanding})$.
- **dispute cap and 60-day window** (`dispute_discipline`): $\bigwedge_{(tier,n)} \forall a, lv.\ AccountLevel(a, lv) \wedge TierOf(lv, tier) \rightarrow \exists^{\leq n} t.\ (OpenDispute(t) \wedge TxnAccount(t, a))$ for $(\mathsf{Entry}, 2), (\mathsf{Mid}, 3), (\mathsf{Premium}, 4), (\mathsf{Elite}, 5)$, conjoined with

$\forall t, td, now.\ \ Raised(t) \ \wedge\ \neg Disputed(t) \ \wedge\ TxnDate(t, td) \ \wedge\ Today(now) \ \rightarrow\ \neg SixtyDaysPast(td, now).$

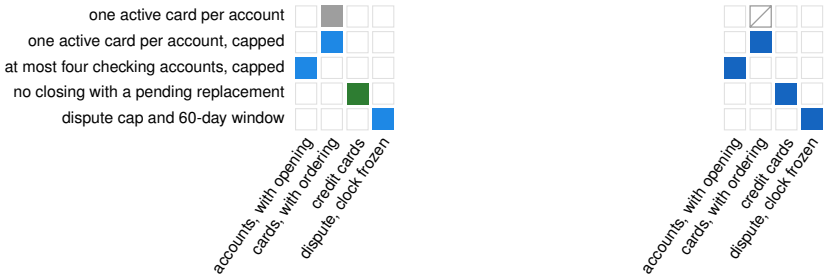


Figure 4: Banking environment–requirement pairs: synthesis outcome (left) and synthesized shield class (right).

C.4 AppWorld

Modeled fragments. We model AppWorld's note-taking and task-management applications, explicitly representing the currently logged-in user in the relational state. The projection retains ownership, collaboration, project, section, task, invitation, title, and pin relations used by the modeled APIs and requirements; fields such as bodies, timestamps, descriptions, due dates, priorities, labels, comments, and attachments are omitted because no retained call depends on them. The application clock is fixed within an episode, and neither fragment has an uncontrollable action.

Sorts and schema (notes). Opaque: u: user, n: note, ti: title. Relations: $Me(u)$, $User(u)$ (the account still exists), $Note(n)$, $NoteOwner(n, u)$, $NTitle(n, ti)$, $Pinned(n)$, and the declared title vocabulary $AvailNTitle(ti)$, which no tool writes. Abbreviations: $Alive := \exists m.\ Me(m) \wedge User(m)$ and $Mine(n) := \exists m.\ Me(m) \wedge User(m) \wedge NoteOwner(n, m)$.

Tool semantics (notes).

Tool	Enabled when	Effect
`create_note`(ti)	$Alive \wedge \neg\exists n.\ NTitle(n, ti) \wedge Mine(n)$	**new** n:$note$; $+Note(n)$, $+NoteOwner(n, me)$, $+NTitle(n, ti)$
`delete_note`(n)	$Note(n) \wedge Mine(n)$	$-Note(n)$, $-NoteOwner(n, \cdot)$, $-NTitle(n, \cdot)$, $-Pinned(n)$
`pin_note`(n) / `unpin_note`(n)	$Note(n) \wedge Mine(n)$	$+Pinned(n)$ / $-Pinned(n)$
`retitle_note`(n, ti)	$Note(n) \wedge Mine(n) \wedge \neg\exists n' \neq n.\ NTitle(n', ti) \wedge Mine(n')$	$NTitle(n) := ti$
`delete_account`$()$	$Alive$	$-User(me)$; every note owned by me is removed with its owner, title and pin facts (the app cascades)
`show_note`(n)	$Note(n) \wedge Mine(n)$	read
`search_notes`$()$	$Alive$	read

Sorts (tasks). Opaque: u: user, p: project, t: task, s: section, i: invite, and project and section names.

Schema (tasks).

Principal/project	$Me(u)$, $User(u)$, $Project(p)$, $Owner(p,u)$, $Inbox(p)$, $PName(p,\cdot)$, $Collab(p,u)$.
Sections/tasks	$Section(s)$, $SecProj(s,p)$, $SName(s,\cdot)$, $Task(t)$, $TaskProj(t,p)$, $TaskSec(t,s)$, $TaskCreator(t,u)$, $Assignee(t,u)$, $Assigner(t,u)$, $Done(t)$.
Invites	$Invite(i)$, $InvProj(i,p)$, $InvInvitee(i,u)$, $InvInvitor(i,u)$.
Name vocabularies	$AvailPName$, $AvailSName$.

Derived predicates.

$$Auth(p) := \exists m.\, Me(m) \wedge User(m) \wedge Collab(p,m),$$
$$TaskAuth(t) := \exists p.\, TaskProj(t,p) \wedge Auth(p).$$

The first is the app's authorization check.

Tool semantics (tasks). The fifteen tools of the compiled menu, none of which creates an object.

Tool	Enabled when	Effect
`delete_project`(p)	$Project(p) \wedge Auth(p) \wedge \neg Inbox(p)$	remove p with its name, owner, inbox flag and collaborator links, every section of p with its name, every task of p with its section, creator, assignee, assigner and done facts, and every invite to p
`delete_section`(s)	$Section(s) \wedge \exists p.\, SecProj(s,p) \wedge Auth(p)$	remove s with its name and project, and every task in s with its facts
`delete_task`(t)	$Task(t) \wedge TaskAuth(t)$	remove t with its facts
`assign_task`(t,u)	$Task(t) \wedge TaskAuth(t) \wedge User(u) \wedge \exists p.\, TaskProj(t,p) \wedge Collab(p,u)$	$Assignee(t) := u$; $Assigner(t) := me$
`unassign_task`(t)	$Task(t) \wedge TaskAuth(t)$	$-Assignee(t,\cdot)$, $-Assigner(t,\cdot)$
`complete_task`(t) / `uncomplete_task`(t)	$Task(t) \wedge TaskAuth(t)$	$+Done(t)$ / $-Done(t)$
`remove_collaborator`(p,u)	$Project(p) \wedge Auth(p) \wedge \neg Inbox(p) \wedge User(u) \wedge \neg Owner(p,u) \wedge Collab(p,u)$	$-Collab(p,u)$ (the assignments of u are left in place)
`accept_project_invite`(i)	$Invite(i) \wedge InvInvitee(i,me) \wedge Alive \wedge \exists p.\, InvProj(i,p) \wedge Project(p) \wedge \neg Inbox(p) \wedge \neg Collab(p,me)$	$+Collab(p,me)$ for the invited project; remove i with its facts
`delete_project_invite`(i)	$Invite(i) \wedge InvInvitor(i,me) \wedge Alive$	remove i with its facts
`show_projects`(), `search_users`()	$Alive$	read
`show_project`(p), `show_sections`(p)	$Project(p) \wedge Auth(p)$	read
`show_task`(t)	$Task(t) \wedge TaskAuth(t)$	read

Uncontrollable actions. None.

Initial families.

Notes	Starting from the benchmark-provided database of one user, we use the application's own deletion operation to retain that user's first three notes; the title vocabulary is those three titles plus one unused title.
Tasks	Starting from the benchmark-provided database around one user, we use the application's own deletion operations to retain the inbox and one further project, one task, and no sections. The retained task is assigned to a non-owner collaborator so that the assignee requirement can be violated. We retain exactly the users referenced by the remaining projects, tasks, assignments, and invitations, together with the remaining names and one unused project and section name.

Environments.

Environment	Initial family	Agent tools
notes M0 + both	notes	delete, pin, unpin, retitle, show, search, `create_note`, `delete_account`
todoist, 15 tools	tasks	the fifteen tools above

Requirements. Seven standing invariants over the two application schemas. Notes title uniqueness is the app's native per-user uniqueness constraint; the dangling-owner invariant is stated over the ownership schema. The five task-app invariants capture consistency conditions enforced by particular endpoints but not uniformly across all operations.

- **every note has an existing owner** (`SN1_no_dangling_owner`): $\forall n, u.\ NoteOwner(n, u) \rightarrow User(u)$.
- **titles unique per user** (`SN2_unique_title_per_user`): $\forall n, n', u, ti.\ NoteOwner(n, u) \land NoteOwner(n', u) \land NTitle(n, ti) \land NTitle(n', ti) \rightarrow n = n'$.
- **assignee is a collaborator** (`TD1_assignee_is_collaborator`): $\forall t, u.\ Assignee(t, u) \rightarrow \exists p.\ TaskProj(t, p) \land Collab(p, u)$.
- **section belongs to the task's project** (`TD2_section_in_project`): $\forall t, s.\ TaskSec(t, s) \rightarrow \exists p.\ TaskProj(t, p) \land SecProj(s, p)$.
- **owner is a collaborator** (`TD3_owner_is_collaborator`): $\forall p, u.\ Owner(p, u) \rightarrow Collab(p, u)$.
- **no dangling user** (`TD4_no_dangling_user`): $\forall t, u.\ (Assignee(t, u) \rightarrow User(u)) \land \forall p, u.\ (Owner(p, u) \rightarrow User(u)) \land \forall p, u.\ (Collab(p, u) \rightarrow User(u))$.
- **inbox is not shared** (`TD5_inbox_not_shared`): $\forall p, u.\ Inbox(p) \land Collab(p, u) \rightarrow Owner(p, u)$.

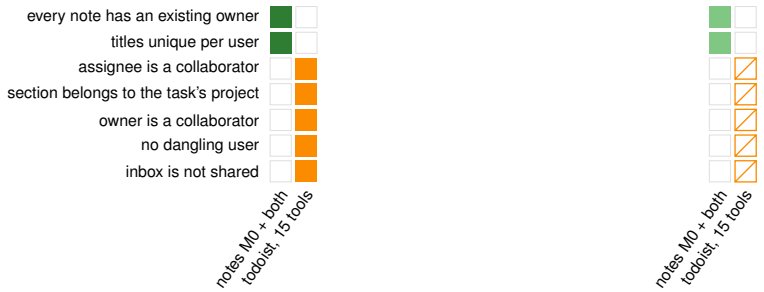


Figure 5: AppWorld environment–requirement pairs: synthesis outcome (left) and synthesized shield class (right).

C.5 WorkBench

Modeled fragments. We model WorkBench's project-management, calendar, and CRM toolkits over their CSV-backed tables and shared e-mail directory. All columns of the three modeled tables are retained. The e-mail and analytics toolkits are outside the interface because their free-text and numeric aggregate semantics are not used by the selected requirements. When WorkBench rejects a call, it returns a message and leaves the database unchanged. We therefore keep such calls available in the model: when the tool's internal checks succeed, the corresponding update is applied; otherwise the state is unchanged.

Sorts. Opaque: t: task, e: event, c: customer, m: e-mail address, x: free text (names, notes, phone numbers, dates, start times and durations share one sort, since the native tools accept any string in any of those columns), l: list, b: board, s: status, i: product interest. The last four are opaque rather than fixed on purpose: the native tools accept values outside their catalogues, and a fixed sort would make that inexpressible.

Schema.

Tasks	$Task(t)$, $TaskName(t,x)$, $TaskAssignee(t,m)$, $TaskList(t,l)$, $TaskDue(t,x)$, $TaskBoard(t,b)$.
Events	$Event(e)$, $EventName(e,x)$, $EventParticipant(e,m)$, $EventStart(e,x)$, $EventDuration(e,x)$.
Customers	$Customer(c)$, $CustName(c,x)$, $CustAssignee(c,m)$, $CustEmail(c,m)$, $CustPhone(c,x)$, $CustLastContact(c,x)$, $CustInterest(c,i)$, $CustStatus(c,s)$, $CustFollowUp(c,x)$, $CustNotes(c,x)$.
Frozen catalogues	$Directory(m)$, $ValidList(l)$, $ValidBoard(b)$, $ValidStatus(s)$, $ValidInterest(i)$.

Derived predicate. $Known(m) := \exists t.\ TaskAssignee(t,m)$, the project manager's own assignee test, which is against the task table and not against the directory.

Tool semantics. Thirty write tools and six reads. For calls used to test referential constraints, we also consider variants in which the relevant argument is a new value not already present in the modeled state.

Tool	Succeeds when	Effect
`create_task`(x,m,l,x',b)	$Known(m) \wedge ValidList(l) \wedge ValidBoard(b)$	**new** t:$task$; $+Task(t)$, name x, assignee m, list l, due x', board b
`delete_task`(t)	$Task(t)$	remove t with its facts
`update_task_name`(t,x), `update_task_due`(t,x)	$Task(t)$	$TaskName(t) := x$ / $TaskDue(t) := x$
`update_task_assignee` (t,m)	$Task(t) \wedge Known(m)$	$TaskAssignee(t) := m$
`update_task_list`(t,l)	$Task(t) \wedge ValidList(l)$	$TaskList(t) := l$
`update_task_board`(t,b)	$Task(t) \wedge ValidBoard(b)$	$TaskBoard(t) := b$
`create_event` (x,m,x',x''); variant with a fresh m	true	**new** e:$event$; $+Event(e)$, name x, participant m, start x', duration x''
`delete_event`(e)	$Event(e)$	remove e with its facts
`update_event_` `name/start/duration` (e,x)	$Event(e)$	the named column of $e := x$
`update_event_` `participant`(e,m); variant with a fresh m	$Event(e)$	$EventParticipant(e) := m$
`add_customer` (x,m,s,m',i,x'); variants with a fresh s, i or m	true	**new** c:$customer$; $+Customer(c)$, name x, assignee m, status s, e-mail m', interest i, notes x'
`delete_customer`(c)	$Customer(c)$	remove c with its facts
`update_customer_` `name/phone/last_` `contact_date/follow_` `up_by/notes`(c,x)	$Customer(c)$	the named column of $c := x$
`update_customer_` `assignee`(c,m); variant with a fresh m	$Customer(c)$	$CustAssignee(c) := m$
`update_customer_email` (c,m)	$Customer(c)$	$CustEmail(c) := m$
`update_customer_` `status`(c,s)	$Customer(c) \wedge ValidStatus(s)$	$CustStatus(c) := s$
`update_customer_` `product_interest`(c,i)	$Customer(c) \wedge ValidInterest(i)$	$CustInterest(c) := i$
`search_tasks`(), `get_` `task_information_by_` `id`(t), `search_events`(), `get_event_` `information_by_id`(e), `search_customers`(), `find_email_address`()	true	read

Uncontrollable actions. None.

Initial families. Both are small slices of the benchmark-provided tables that retain the create, update, and delete mechanics together with the relevant catalogues.

Full slice	The benchmark-provided tables cut to their first four tasks, four events, and four customers, with the whole 20-address directory and the four catalogues.
CRM slice	One customer, no task or event, and the directory cut to the addresses used by that customer's row plus one spare; the selected requirements hold initially.

Environments. The three toolkits, each with all of its modeled tools.

Environment	Initial family	Agent tools
projects, full	full slice	the 7 task writes, `search_tasks`, `get_task_information_by_id`, `find_email_address`
calendar, full	full slice	the 8 event writes, `search_events`, `get_event_information_by_id`, `find_email_address`
CRM, full	CRM slice	the 15 customer writes, `search_customers`, `find_email_address`

Requirements. Eight referential constraints over the native table and catalogue schema. These requirements capture referential constraints in the WorkBench schema that are not uniformly enforced by the modeled write tools. Each requirement is paired with the toolkit that writes its constrained table: one with calendar, four with CRM, and three with project management.

- **event participant in directory** (`WB1`): $\forall e, m.\ \mathit{EventParticipant}(e, m) \rightarrow \mathit{Directory}(m)$.
- **customer assignee in directory** (`WB2`): $\forall c, m.\ \mathit{CustAssignee}(c, m) \rightarrow \mathit{Directory}(m)$.
- **customer e-mail unique** (`WB3`): $\forall c, c', m.\ \mathit{CustEmail}(c, m) \wedge \mathit{CustEmail}(c', m) \rightarrow c = c'$.
- **task assignee in directory** (`WB4`): $\forall t, m.\ \mathit{TaskAssignee}(t, m) \rightarrow \mathit{Directory}(m)$.
- **status in catalogue** (`WB5`): $\forall c, s.\ \mathit{CustStatus}(c, s) \rightarrow \mathit{ValidStatus}(s)$.
- **product interest in catalogue** (`WB6`): $\forall c, i.\ \mathit{CustInterest}(c, i) \rightarrow \mathit{ValidInterest}(i)$.
- **list in catalogue** (`WB7`): $\forall t, l.\ \mathit{TaskList}(t, l) \rightarrow \mathit{ValidList}(l)$.
- **board in catalogue** (`WB8`): $\forall t, b.\ \mathit{TaskBoard}(t, b) \rightarrow \mathit{ValidBoard}(b)$.

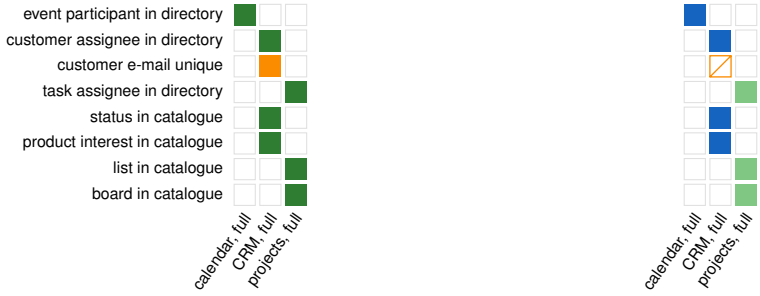


Figure 6: WorkBench environment–requirement pairs: synthesis outcome (left) and synthesized shield class (right).

D SUPPLEMENT TO THE CONTROLLED EXTENSIONS

This section specifies the nine controlled configurations reported separately from the benchmark-derived census in Section 5.2. The extensions keep the corresponding relational application model as their base and make the semantic changes stated below. The configurations vary specific environment dynamics while leaving the requirement and evaluation procedure unchanged

D.1 TELECOM: OVERDUE-BILL LIFECYCLE

The Telecom extension adds one uncontrollable lifecycle transition that is absent from the benchmark simulator: an issued bill whose due date is before the fixed current date may become overdue. The native payment event is retained in one arm and omitted in the other.

Environment action	Fires when	Effect
`payment_received`(b)	$BillStatus(b, \mathsf{Awaiting})$	$BillStatus(b) := \mathsf{Paid}$
`bill_falls_overdue` (b)	$BillStatus(b, \mathsf{Issued}) \wedge \exists d.\ BillDue(b, d) \wedge Past(d)$	$BillStatus(b) := \mathsf{Overdue}$

The extension uses C1001's benchmark-provided component as its initial family. Its issued bill is already past due but not overdue; the customer's active lines are initialized as suspended so the standing requirement holds and the added lifecycle transition is enabled.

D.2 BANKING: ADVANCING TIME

The Banking extensions start from the task-084 dispute model of Appendix C.3. The finite calendar begins on the benchmark's current date, 11/14/2025, and extends to one day after the last modeled filing deadline. Once the final day is reached, the date no longer advances. Advancing the clock moves $Today$ to the next date according to $NextDay$. In the day-per-call and turn-taking variants, reaching a dispute's $ResolvesOn$ date also closes that dispute.

The two variants differ in who controls this advancement. In *day per call*, every agent call also advances the clock by one day. In *turn-taking day*, agent actions alternate with an uncontrollable `next_day` action; `agent_idle` allows the agent to yield its turn without otherwise changing the state.

Tool	Enabled when	Effect
`file_dispute`(t)	$\neg Disputed(t)$ [and $AgentTurn$ under turn-taking]	$+Disputed(t)$, $+OpenDispute(t)$ [then a tick under day-per-call; $-AgentTurn$ under turn-taking]
`get_current_time`()	true [$AgentTurn$ under turn-taking]	read [a tick under day-per-call; $-AgentTurn$ under turn-taking]
`agent_idle`()	$\neg AgentTurn$	read (turn-taking only)
`next_day`(), environment	$\neg AgentTurn$ under turn-taking	a tick; $+AgentTurn$ under turn-taking

The C1–C3 comparison uses the same task-084 source state with two additional modeling choices held fixed across the three arms: only the designated TechWorld request remains in $Raised$, and the shipped $ResolvesOn$ fact is removed. The filing tool is additionally guarded by the documented 60-day filing window. C1 keeps the clock frozen, C2 makes the one-day tick controllable by the agent, and C3 uses a bounded uncontrollable tick under turn-taking. Thus C1–C3 compare three clock-control regimes while keeping their obligation set and filing guard fixed.

D.3 TELECOM: KEY SCHEDULING

The key-scheduling extension uses the whole Telecom database. We add a unary relation $Req(c)$ to mark the customer currently being served. Customer-specific write tools may be called only for that customer. After each such call, the next customer is chosen nondeterministically from the benchmark-provided customers. The initial family ranges over each possible requested customer. Thus, the extension changes which calls are available at each step and how customers are scheduled, without adding a separate environment action.

D.4 OUTCOMES

Table 13 reports the nine configurations. The two overdue-bill variants and the three Banking regimes in which time can advance without an agent choice require $S^\star \subsetneq S_1$; the frozen and agent-controlled clock arms remain one-step.

Table 13: Controlled extensions: synthesis outcome and synthesized shield class.

Environment	Requirement	Outcome	Shield class
Telecom			
C1001 + overdue + payment events	no active line with an overdue bill	finite	beyond one step
C1001 + overdue event	no active line with an overdue bill	finite	beyond one step
whole DB, key-scheduled	no active line with an overdue bill	no cert.	—
	one bill awaiting payment	finite	no restriction
Banking			
dispute arm C1, frozen	dispute cap and 60-day window	finite	one-step
dispute arm C2, agent ticks	dispute cap and 60-day window	finite	one-step
dispute arm C3, bounded env.	dispute cap and 60-day window	finite	beyond one step
dispute, day per call	dispute cap and 60-day window	finite	beyond one step
dispute, turn-taking day	dispute cap and 60-day window	finite	beyond one step

E SUPPLEMENT TO THE LLM BASELINE

E.1 REFERENCE DECISIONS AND SCORING

For every solved instance with an evaluable controllable call, we construct a fixed *reference decision set* from the synthesized shield. Candidate states are collected in a fixed order and deduplicated by canonical form. For instances solved by finite synthesis, candidates come from the canonical states of the solved finite abstraction. For symbolically solved instances, candidates come from the initial states, states retained from an enumeration of the reachable state space, and a bounded exploration of successors from the initial states. For the enumeration, every candidate state and call is labelled using the synthesized $S^\star$. Final decisions are selected by round-robin over tool names and distinct states so that no single tool or state dominates the reference set. We retain at most 300 candidate states per instance and select at most 200 reference decisions, including up to 100 blocked calls where such calls exist.

A generated program is scored by executing it on every reference state–call pair. A trial matches an instance only when all decisions on the reference set agree with the labels induced by the synthesized shield. The reference sets are constructed without randomness and frozen before model evaluation; neither the states nor their labels are included in a prompt. A mismatch therefore witnesses a concrete error in the generated program, while agreement on the finite reference set is not itself a correctness certificate.

E.2 PROTOCOL AND PROMPTS

Models and settings. We use `gpt-5.6-luna` and `deepseek-v4-pro`, both with high reasoning effort and provider-default settings otherwise. Both models are evaluated on the 32 solved main-census instances, five trials each, against the corresponding synthesized reference shield. The seven controlled extensions with an evaluable synthesized shield are evaluated in the same way and reported separately.

Prompt. Each prompt is rendered from the corresponding relational environment specification using the same template. It gives the runtime representation of states and grounded calls, the relational data model, the standing requirement as a first-order formula together with an English rendering, and the native precondition and complete transition semantics of every controllable tool and uncontrollable environment action. The model is asked to return a deterministic Python function `allow(state, call)` deciding whether the grounded call may remain available in the supplied state.

The prompt explicitly requires the program to preserve the standing requirement under every documented outcome and environment action, to account for whether safety can continue to be enforced after an allowed call, and not to restrict calls unnecessarily. It also requires calls whose native precondition is false to be rejected and the implementation to return a Boolean for every well-typed input. Listing 1 shows a representative Telecom prompt; the complete prompts and stored model outputs are included in the artifact.

Listing 1: Prompt for Telecom, "C1002, all 15 tools", no active line with an overdue bill; 13 API blocks elided.

````
You are implementing the runtime authorization layer for a tool-using agent.

The agent may receive arbitrary user requests and may choose its own plans. Your job is not
    to
solve a particular user task or prescribe an action sequence. Your job is to implement the
standing interface that determines which agent tool calls may be exposed.

## Objective

Write a deterministic Python function:

```python
def allow(state, call) -> bool:
    ...
```

`allow(state, call)` must return `True` exactly when the grounded tool call may be exposed to
    the
agent in the supplied state.

The implementation must satisfy all of the following:

1. Every execution produced through calls that the interface exposes must respect the
    standing
   requirement below, including every documented possible outcome and every documented system
   event.
2. The interface must continue to provide this guarantee as execution proceeds; do not permit
     a
   call merely because the immediate resulting state looks acceptable if making that call
    would
   leave no way to continue operating while preserving the requirement.
3. Do not unnecessarily restrict the agent. If a call can remain available while preserving
    the
   guarantee above, it should remain available.
4. A call whose native API precondition is false must return `False`.
5. The implementation must be total: for every well-typed `state` and `call`, it must return a
   Boolean and must not raise an exception.
6. Do not use the current user's task, an assumed plan, or guesses about what the agent is
    trying
   to accomplish. This is a deployment-wide interface.

If the application itself offers no agent call at all in a state, you do not need to invent
one.

## Runtime representation

A state is represented as a mapping from relation names to sets of tuples.

```python
state["RelationName"]        # {("id1", "id2"), ...}
```

Every relation declared for this deployment is present as a key in `state`, even when its
value is the empty set. Relation values are Python `set` objects.

A grounded call is represented as:

```python
{
    "name": "tool_name",
    "args": {"parameter_name": "value"}
}
```

Concrete relations, tools and parameters for this deployment appear below.

You may define helper functions, but the required public entry point is:

```python
````
````

````
def allow(state, call) -> bool
```

## Application data model

The application stores objects of the following kinds:

```text
customer, line, bill, plan, device, phone
```

Values of those kinds are opaque string identifiers; their spelling carries no
meaning.

The remaining value kinds are drawn from fixed, closed sets:

```text
line_status: "Line/Active", "Line/Suspended", "Line/Pending Activation", "Line/Closed"
bill_status: "Bill/Draft", "Bill/Issued", "Bill/Awaiting Payment", "Bill/Paid", "Bill/Overdue
     ", "Bill/Disputed"
account_status: "Account/Active", "Account/Suspended", "Account/Pending Verification", "
     Account/Closed"
date: "2024-08-20", "2024-10-05", "2024-12-15", "2025-01-05", "2025-01-10", "2025-01-15",
     "2025-01-19", "2025-01-20", "2025-02-01", "2025-02-05", "2025-02-19", "2025-02-25",
     "2025-02-26", "2025-03-01", "2025-03-15", "2026-06-30", "2026-08-15", "2026-12-31",
     "2027-02-28"
```

The following relations are available in `state`:

```text
AccountStatus(customer, account_status)
CustomerPhone(customer, phone)
OwnsLine(customer, line)
OwnsBill(customer, bill)
LineStatus(line, line_status)
LinePlan(line, plan)
LineDevice(line, device)
LinePhone(line, phone)
RoamingEnabled(line)
ContractEnd(line, date)
SuspensionStart(line, date)
LastPlanChange(line, date)
LastSimReplacement(line, date)
BillStatus(bill, bill_status)
BillIssued(bill, date)
BillDue(bill, date)
Plan(plan)
Device(device)
DateBefore(date, date)
Today(date)
```

Wherever a statement below quantifies over one of the opaque identifier kinds, it
ranges over the identifiers of that kind that occur in the current state. Wherever it
quantifies over one of the fixed value kinds, it ranges over that kind's whole set as
listed above.

## Standing requirement

The standing requirement for this deployment is:

```text
for every customer c and every line l, if (OwnsLine(c, l) holds and LineStatus(l, "Line/
     Active")
    holds), then there is no bill b such that OwnsBill(c, b) holds and BillStatus(b,
    "Bill/Overdue") holds
```

Formally:

```text
forall c: customer, l: line .
    ((OwnsLine(c, l) and LineStatus(l, "Line/Active"))
      ->
        not (exists b: bill . (OwnsBill(c, b) and BillStatus(b, "Bill/Overdue")))
    )
```

The English text above is only a rendering of the formula. The formula defines the
requirement.
````
````

````
## Available agent APIs

Each API below may be offered to the agent. A grounded call names the API and binds
each parameter listed in its signature to a value.

### suspend_line

```text
suspend_line(customer, line)
```

The call succeeds only when:

```text
(
    exists as: account_status . AccountStatus(customer, as)
  and
    OwnsLine(customer, line)
  and
    exists ls: line_status . LineStatus(line, ls)
  and
    LineStatus(line, "Line/Active")
)
```

Otherwise the API rejects the call.

**State transition semantics**

The state then changes as follows:

```text
- remove LineStatus(l, ls) wherever it is present and:
      l = line
- add LineStatus(l, ls) for every combination satisfying:
      (l = line and ls = "Line/Suspended")
- remove SuspensionStart(l, d) wherever it is present and:
      l = line
- add SuspensionStart(l, d) for every combination satisfying:
      (l = line and Today(d))
- every state fact not listed above remains unchanged.
```

### resume_line

```text
resume_line(customer, line)
```

The call succeeds only when:

```text
(
    exists as: account_status . AccountStatus(customer, as)
  and
    OwnsLine(customer, line)
  and
    exists ls: line_status . LineStatus(line, ls)
  and
    (LineStatus(line, "Line/Suspended") or LineStatus(line, "Line/Pending Activation"))
)
```

Otherwise the API rejects the call.

**State transition semantics**

The state then changes as follows:

```text
- remove LineStatus(l, ls) wherever it is present and:
      l = line
- add LineStatus(l, ls) for every combination satisfying:
      (l = line and ls = "Line/Active")
- remove SuspensionStart(l, d) wherever it is present and:
      l = line
- every state fact not listed above remains unchanged.
```

[... the remaining 13 APIs follow in the same format: send_payment_request,
````

```
 enable_roaming, disable_roaming, get_customer_by_id, get_customer_by_phone,
 get_bills_for_customer, get_data_usage, get_details_by_id_{line,bill,customer,
 device,plan} and transfer_to_human_agents; the reads carry a guard and
 "The state is unchanged." ...]

## Output

Return only one Python program.

The program may contain helper functions, but it must define:

```python
def allow(state, call) -> bool:
```

Do not call external services, use network access, or depend on information not included in
     this
specification.
```

E.3 Complete results

Results by benchmark. Table 14 breaks out the five-trial results by benchmark.

		gpt-5.6-luna		deepseek-v4-pro	
Benchmark	n	pass^1	pass^5	pass^1	pass^5
Telecom	7	97.1	85.7	100.0	100.0
Airline	12	93.3	75.0	93.3	66.7
Banking	4	85.0	50.0	95.0	75.0
AppWorld	2	90.0	50.0	90.0	50.0
WorkBench	7	80.0	42.9	51.4	28.6
All	32	90.0	65.6	85.6	65.6

Table 14: LLM results by benchmark on the 32 solved main-census instances evaluated in Table 2.

Consistency over trials. Figure 7 gives pass^k against k on the 32 solved main-census instances. `gpt-5.6-luna` matches 21 of these instances in all five trials and `deepseek-v4-pro` matches 21. The two five-of-five sets overlap on 15 instances; `gpt-5.6-luna` alone matches 6 more in all five trials and `deepseek-v4-pro` alone 6. `gpt-5.6-luna` matches every instance at least once, while `deepseek-v4-pro` never matches 3.

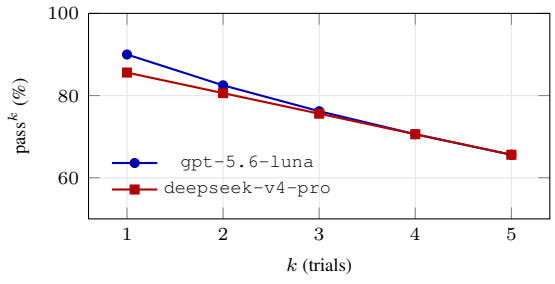


Figure 7: pass^k against k on the 32 solved main-census instances.

Per-instance grids. The grids below show the instances of Table 1. Green denotes a full match on the reference decision set in every trial; orange denotes over-restriction; purple denotes disagreement across trials. An instance that was not solved is crossed out according to its synthesis outcome.



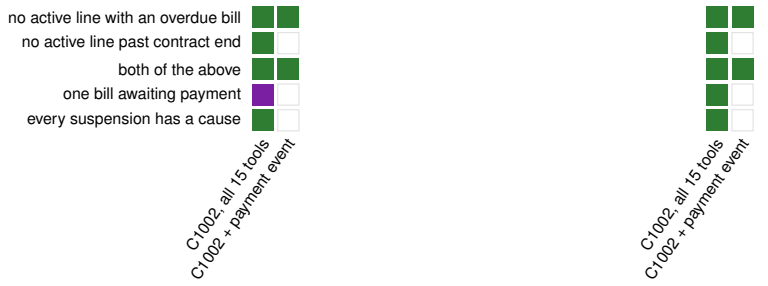


Figure 8: Telecom: `gpt-5.6-luna` (left) and `deepseek-v4-pro` (right).

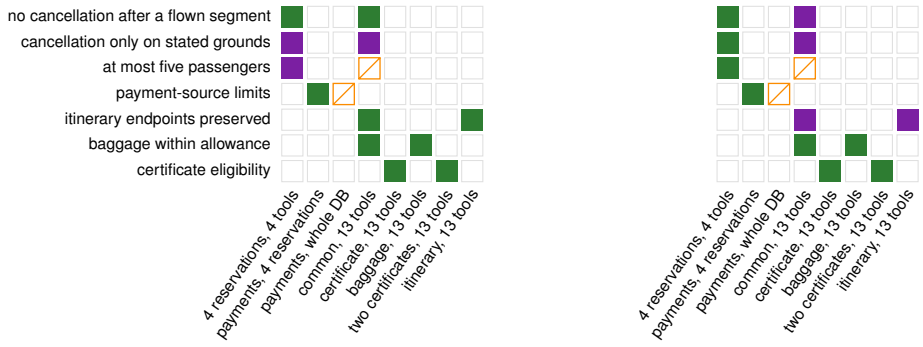


Figure 9: Airline: `gpt-5.6-luna` (left) and `deepseek-v4-pro` (right).

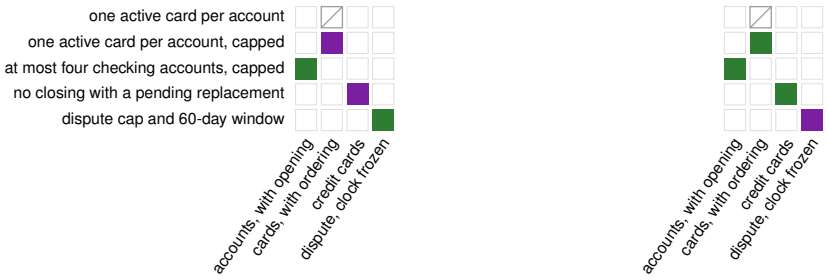


Figure 10: Banking: `gpt-5.6-luna` (left) and `deepseek-v4-pro` (right).

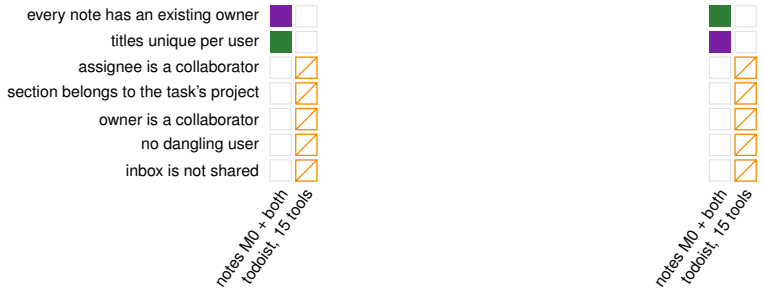


Figure 11: AppWorld: `gpt-5.6-luna` (left) and `deepseek-v4-pro` (right).

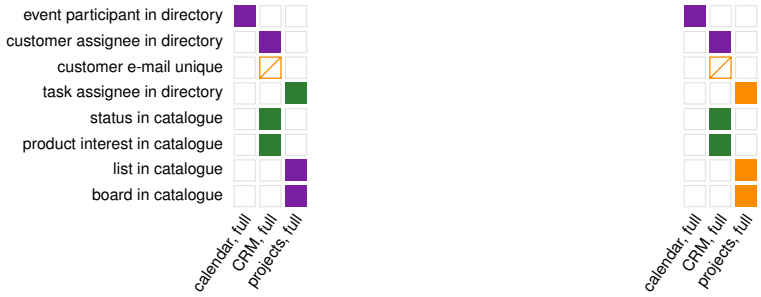


Figure 12: WorkBench: `gpt-5.6-luna` (left) and `deepseek-v4-pro` (right).