L2R: Verification-Guided Repair of LLM-Generated Smart-Home Automation Rules

Zhelin Zheng[1], Chenglong Fu[2], Xiaojiang Du[1(✉)], and Yuede Ji[3]

[1] Stevens Institute of Technology, Hoboken, NJ, USA
{zzheng27,xdu16}@stevens.edu
[2] University of North Carolina at Charlotte, Charlotte, NC, USA
chenglong.fu@charlotte.edu
[3] University of Texas at Arlington, Arlington, TX, USA
yuede.ji@uta.edu

Abstract. Trigger-action automation lets smart-home users define device behavior through event-condition-action rules, but LLM-generated automation logic may still violate safety constraints when multiple devices interact. TAPAssure combines LLM-based NuSMV model generation with formal verification, but its baseline repair loop passes raw counterexamples back to the LLM. Such traces are precise enough for model checking, but they do not directly tell the LLM which transition block to edit, so repair often rewrites correct logic or requires many iterations before passing verification. This paper presents Log-to-Repair (L2R), a layered counterexample-to-repair step that extracts the failed property, keeps the property-relevant trace slice, maps changed variables to candidate transition blocks, and asks the LLM to patch those blocks while preserving unrelated logic. A stagnation-aware layer widens the local context only when repeated repair attempts make no progress. A normalization-and-fallback layer detects formatting-only repeats and handles indirect causal chains. On a 40-sample benchmark, the final system reduces the average number of verification-and-repair iterations (Process 2 of the TAPAssure workflow) from 3.09 to 2.00 and average runtime from 13.68 s to 9.67 s, while repair success remains statistically comparable (98.0% vs. 99.0% for the baseline, $p = 0.41$); the runtime gain comes mainly from compressing long-tail repair loops. On conflict-focused cases, it matches the baseline success rate of 99.4% while cutting average iterations from 3.32 to 2.02.

Keywords: Smart Home · Internet of Things · Large Language Models · Formal Verification

1 Introduction

Trigger-action programming (TAP) lets smart-home users describe device behavior through event-condition-action rules: turn a light on when motion is detected, start a fan when temperature crosses a threshold, or keep a device off in away mode. User studies show that inexperienced users can learn to express a range

of desired smart-home behaviors through this style of programming [22]. However, rules that appear safe individually can interact through shared devices, environmental conditions, or user modes to violate occupancy, access-control, or climate policies. An empirical study of SmartThings applications documents conflicts involving shared devices, physical interactions, and global variables, and develops IoTCheck to detect them through model checking [21]. Soteria and IoTSan demonstrate the need to analyze interacting IoT applications rather than checking each application in isolation [7,17]. As the number of devices and rules grows, these interactions become harder to reason about manually.

LLMs make automation generation easier still by translating natural-language requirements into executable automation logic. Related work on LLM-assisted tasking and physical invariant extraction illustrates the value of using language models to interpret requirements and system context [1,27]. This does not remove the need for safety validation: generated rules may omit important context, choose the wrong device state, or introduce unsafe multi-device behavior. Formal approaches already support both detection and repair. AutoTap synthesizes and repairs trigger-action programs against LTL properties [30], while TAPFixer repairs rule-interaction vulnerabilities through model checking and negated-property reasoning [29]. These systems establish that automated repair of unsafe automation is an existing research direction. The question addressed here is how to make repair more efficient within an LLM-based generation-and-verification workflow.

TAPAssure [4] is the closest baseline because it combines LLM-based NuSMV model generation with formal verification and iterative repair. In its workflow, Process 1 generates a syntactically valid NuSMV model from user requirements, available devices, and safety properties, and Process 2 verifies that model with NuSMV. If a property fails, NuSMV returns a counterexample trace and the LLM regenerates the model; verification repeats until all properties pass or a retry cap is reached. The design is appealing because the safety check is not left to the LLM's internal reasoning; instead, model checking provides an external, explicit safety oracle. However, the design leaves one important interface problem: a raw counterexample identifies a failing execution, yet it does not directly tell the LLM which device rule, transition block, or condition should be edited. The verifier is doing its job, but its output is not naturally aligned with what the LLM is being asked to do next.

In practice, this verifier-to-repair gap manifests as wasted iterations. The LLM may rewrite unrelated logic, leave the real fault unchanged, or oscillate between near-identical regenerations. The cost can be substantial: the original TAPAssure evaluation includes a syntax-correction trial that required 53 iterations, and the retry cap was set at 60 to prevent unbounded cost [4]. This points to a concrete bottleneck: the limiting factor is not the verifier itself, but the way verifier output is transformed into a repair signal.

This paper presents Log-to-Repair (L2R)[4], a repair step that converts NuSMV counterexamples into localized repair guidance before LLM regeneration. L2R extracts the failed property, keeps the property-relevant trace slice, maps changed variables to candidate `next(...)` transition blocks, and asks the LLM for a small repair rather than a full rewrite. The method consists of three layers. The first layer is the structured L2R prompting core just described. The second layer monitors repair progress and widens the local context only when repair stalls. The third layer normalizes malformed intermediate models, detects repeated outputs that differ only in formatting, and supplies fallback context for indirect causal chains. The work keeps the original two-process verification-centered workflow, the use of NuSMV as the formal verifier, and the iterative regenerate-and-reverify loop. The contribution is specifically a stronger verifier-to-repair interface in Process 2, not a redesign of the surrounding framework or a general-purpose synthesis system.

We make the following contributions:

- We identify raw counterexample prompting as a source of unnecessary repair iterations in TAPAssure-style verification-guided repair, and frame it as a verifier-to-repair interface bottleneck.
- We design L2R, a counterexample-to-repair step that extracts the failed property, trace slice, changed variables, and target transition logic before LLM regeneration, and biases regeneration toward minimal patches.
- We add stagnation-aware context widening and normalization-and-fallback handling, so the repair loop can expand its context when localized repair stops making progress and can detect repeated outputs that differ only in formatting.
- We evaluate the method through a four-condition ablation on 40 smart-home samples and show that it reduces Process 2 iterations and runtime while maintaining comparable repair success.

The remainder of this paper is organized as follows. Section 2 formulates the problem and Section 3 presents the proposed method. Section 4 describes the conceptual testbed, system implementation, benchmark construction, ablation design, and evaluation metrics. Section 5 reports the experimental results. Section 6 discusses findings and limitations, Section 7 positions the work relative to prior systems, and Section 8 concludes.

2 Problem Statement

TAPAssure addresses smart-home safety through a two-process workflow [4]. In Process 1, an LLM generates a syntactically valid NuSMV model from user requirements, device context, and safety properties. In Process 2, NuSMV [10]

[4] The name reflects that the NuSMV counterexample trace is treated as a repair log to be distilled into an edit target; the input is a counterexample trace produced by the model checker, not a runtime execution log.

verifies that model against the same properties; if all hold, the model is accepted, and if any property fails, NuSMV returns a counterexample trace, the LLM regenerates the model, and verification is repeated until convergence or a retry limit. This generate-check-revise structure is related to counterexample-guided inductive synthesis, exemplified by SKETCH's SAT-based synthesis–verification loop [20]. L2R uses LLM regeneration and does not inherit that procedure's completeness guarantees.

The design is sound in principle, but its repair stage has a practical weakness. Prior work on explaining counterexamples shows that exposing a failing execution and identifying its essential error information are distinct tasks [14]. A NuSMV counterexample is a low-level formal artifact: it encodes state transitions, variable assignments, and property failures in a form designed for model-checking analysis, not for targeted model editing. When this raw trace is forwarded to the LLM, the model must infer on its own which automation rule or `next(...)` block is responsible for the violation. Without that localization, the repair tends to be too broad: a violation rooted in a single actuator can trigger a large rewrite that overwrites logic unrelated to the failure. Iterations are wasted, and new errors may be introduced into parts of the model that were already correct. The practical severity of the issue is visible in TAPAssure's own evaluation, where a Qwen2.5 syntax-correction trial required 53 repair iterations before converging, and the retry cap was set at 60 to prevent unbounded cost [4]. Safety-violation repair varies similarly across scenario-property pairs.

Formally, let M be the current NuSMV model, φ a violated safety property, and σ the counterexample trace produced by NuSMV. The baseline repair step asks the LLM to produce a new model M' conditioned directly on σ. The mapping $\sigma \mapsto M'$ is underconstrained: σ is high-dimensional and weakly structured for the purpose of repair. It does not identify which subset of M is causally responsible for the violation of φ, and it does not constrain the LLM to preserve the parts of M that already pass verification. In practice, this creates two repair risks. First, the LLM may edit too much of the model and disturb logic that already passed verification. Second, it may keep editing the same local block even when the violation depends on a wider chain of device states.

We study how σ can be converted into repair guidance that localizes the likely faulty transition, preserves unrelated logic, and expands context only when repeated attempts fail. We focus only on Process 2: the initial model-generation process is unchanged, and NuSMV is not replaced as the verifier. The scope is intentionally narrow. We assume that the safety properties are already expressed in a form NuSMV can check, and that the generated model has passed syntax validation before entering Process 2. We also assume that the verifier is trusted: if NuSMV reports a violation, the repair loop treats the counterexample as the source of feedback; if NuSMV reports that all properties pass, the model is accepted. The challenge addressed here is not verification soundness, but the usefulness of the feedback passed from the verifier to the LLM.

The repair loop must also preserve user intent as much as possible. A repair that satisfies all safety properties by deleting major parts of the automation logic

is not desirable. For this reason, a useful repair prompt should tell the LLM what to change and what to leave unchanged. This preservation requirement is one reason we focus on transition-block localization instead of asking the LLM to regenerate the whole model after every failed verification attempt.

The research question can therefore be stated compactly: how can a raw counterexample trace be transformed into a structured repair signal that localizes the fault, constrains the rewrite scope, and reduces the number of iterations needed for Process 2 to converge, without giving up the soundness guarantees provided by the underlying model checker?

3 Proposed Method

L2R sits between NuSMV and the LLM repair prompt. Instead of forwarding the full counterexample trace, it extracts the evidence most relevant to the failed property, links that evidence back to candidate transition blocks in the NuSMV model, and asks the LLM to revise those blocks while preserving the rest. The method is layered: a structured L2R prompting core (Sections 3.1–3.3), a stagnation-aware broadening layer for repeated no-progress cases (Section 3.4), and a normalization-and-fallback layer that handles malformed intermediate models, indirect causal chains, and formatting-only repeats (Section 3.5). Section 3.6 describes the implementation constraints that hold across all three layers. NuSMV continues to decide whether a model is safe; L2R only reformats failed verification output for the LLM, which keeps the method compatible with the original TAPAssure loop while making each regeneration attempt more focused. Figure 1 summarizes the overall workflow.

3.1 Counterexample-to-Repair Translation

Given a violated property φ and counterexample σ, L2R first extracts the variables appearing in φ and finds the first property-relevant failure state in σ. It then keeps a short trace slice around that state. The purpose of the slice is to keep the evidence that explains the failure while removing trace lines that are not useful for repair. This is especially important for NuSMV traces, where many assignments simply describe the surrounding state and do not indicate which transition should be changed.

L2R then compares the failure state with the previous state to identify changed variables. These variables are mapped to candidate `next(...)` assignments in M. The mapping is not treated as a proof of causality. It is used to form a small candidate repair region for the LLM. The repair prompt contains the violated formula, the inferred target actuator, the reduced trace slice centered on the violation state, root-cause hints derived from the changed variables, and explicit minimal-patch instructions. The preservation instructions tell the LLM not to modify unrelated declarations, safety properties, or device transitions. This structure gives the LLM an explicit repair target instead of leaving it to infer the target from the full raw trace. The goal is not to discard information;

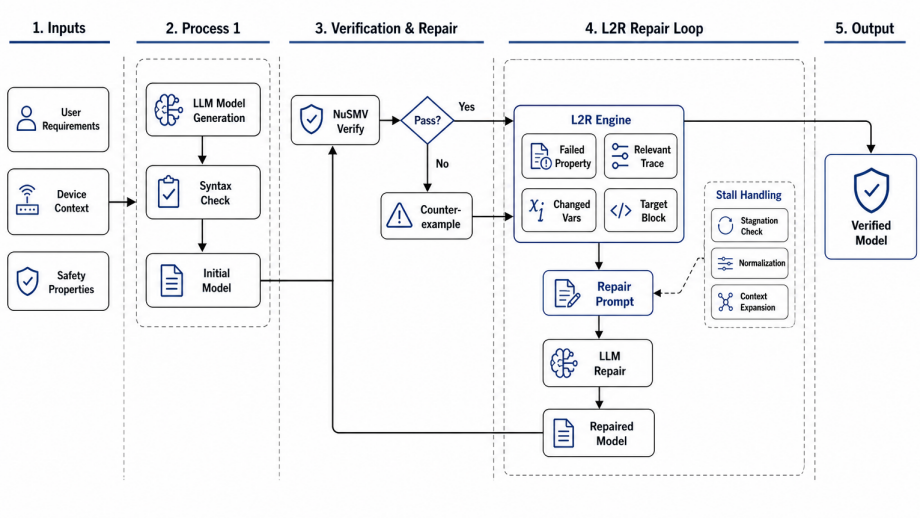


Fig. 1. Overview of the proposed Log-to-Repair workflow. Process 1 generates and syntax-checks an initial NuSMV model. Process 2 verifies the model with NuSMV. If a property fails, L2R converts the counterexample into a repair prompt by extracting the failed property, relevant trace, changed variables, and target transition logic. Stagnation broadening and normalization-and-fallback are engaged only when repeated repair attempts make no progress. The repaired model is re-verified until all safety properties pass or the retry limit is reached.

it is to reduce trace noise and expose the most repair-useful evidence first. Algorithm 1 summarizes the extraction procedure together with the stagnation detection used by the broadening layer (Section 3.4).

3.2 Minimal-Patch-Oriented Regeneration

L2R does not compute a formally minimal patch. Instead, it uses prompt constraints to bias regeneration toward small edits. This distinction matters because the method remains LLM-based: NuSMV verifies the repaired model, but it does not prove that the textual edit is minimal. The goal is to reduce unnecessary rewrites, not to solve a formal program-repair optimization problem. Localization-guided LLM repair has also been studied using MaxSAT-derived program sketches [18]. L2R instead supplies a heuristic candidate region as prompt context; these instructions neither formally restrict the editable region nor guarantee a minimum-size repair.

For example, if the failed property concerns `AC_state`, the prompt marks the corresponding `next(AC_state)` block as the primary edit target and instructs the LLM to leave lock, fan, declaration, and unrelated property lines unchanged. In a raw-counterexample baseline, the LLM may regenerate the whole `MODULE main` block, which can fix one violation while introducing another. The L2R

Algorithm 1 L2R counterexample-to-repair extraction with stagnation detection

Require: model M, violated property φ, counterexample trace σ, repair history H
Ensure: structured repair prompt P

$V_\varphi \leftarrow \textsc{Vars}(\varphi)$ ▷ variables appearing in the failed property
$s^* \leftarrow$ first state in σ where φ-relevant assignments violate φ
$\hat{\sigma} \leftarrow$ trace slice of σ centered on s^*
$\Delta \leftarrow \{v \mid v$ changed between s^{*-1} and $s^*\}$ ▷ changed variables
$B \leftarrow \{\texttt{next}(v)$ blocks in $M \mid v \in \Delta \cup V_\varphi\}$ ▷ candidate edit targets
$sig \leftarrow (\varphi, \text{main variables of } \hat{\sigma})$ ▷ violation signature
$stag \leftarrow sig$ repeats in ≥ 2 consecutive entries of H
$\quad \vee\ \textsc{Fingerprint}(M) \in H\ \vee\ H$ oscillates between two signatures
if *stag* **then**
    widen tracked-variable set with related mode/switch/auth. variables
end if
$P \leftarrow \langle \varphi,$ target actuator, $\hat{\sigma},$ hints from $\Delta \rangle$
append minimal-patch and preservation instructions to P
return P

prompt instead narrows the edit scope and makes each repair attempt easier to inspect.

3.3 Illustrative Repair Flow

Consider a safety property requiring a door to remain locked unless an authorized or present-user condition holds. A raw counterexample may show a sequence of states in which a switch becomes active, the home mode changes, and the lock eventually opens while the authorization variable is false. The trace is useful because it proves the violation, but it is not immediately clear which line of the model should be edited. A direct repair prompt may cause the LLM to rewrite the lock logic, the mode logic, or even unrelated device transitions.

L2R handles this case by first extracting the variables in the failed property, such as `lock_state` and `authorized`. It then keeps the trace window around the first state where the lock becomes open under an unauthorized context. If the change from the previous state is tied to `lock_state`, L2R marks the corresponding `next(lock_state)` block as the first repair target, and the prompt asks the LLM to add or restore the missing authorization condition while preserving unrelated rules. If the same property fails again and the trace suggests that a mode or switch variable indirectly controls the unlock action, L2R widens the context to include those related variables.

3.4 Stagnation-Aware Context Widening

A narrow repair target can fail when the property depends on indirect multi-device logic. For example, a door-unlock property may depend on a switch, a mode variable, and a presence or authorization condition rather than a single lock

transition. In such cases, a strictly localized prompt may keep asking the LLM to edit the same block even though the missing condition appears elsewhere.

L2R therefore monitors recent repair history for three stagnation signals: (i) a violation signature, defined as the failed property paired with the main variables in the current trace slice, repeats in two or more consecutive iterations; (ii) a regenerated model is identical to a previously seen model under normalized comparison; or (iii) the violation alternates between two nearby failure patterns in a simple oscillating form. When any of these signals is triggered, L2R temporarily widens the tracked-variable set used in the repair prompt, allowing the LLM to see a broader local context around the failed property.

The widening is conditional. Easy cases keep the narrow prompt and remain efficient; only genuinely stuck cases receive additional context. Conceptually, broadening is a controlled relaxation of the localization heuristic, applied only after that heuristic has failed for the case at hand.

3.5 Normalization and Fallback Context

The third layer of the method addresses three failure modes that localization and broadening alone do not fully resolve: malformed intermediate models, indirect causal chains, and formatting-only repeats.

Deterministic normalization for malformed intermediate models. A repair attempt sometimes returns NuSMV code that is structurally near-correct but contains formatting irregularities or partial fragments that confuse downstream extraction. L2R applies deterministic normalization to known malformed patterns. For example, normalization ensures that the regenerated text begins from the expected `MODULE main` structure and strips non-code preamble that the LLM occasionally produces. This makes downstream extraction and repeated-output detection more reliable without altering the repair logic itself.

Fallback extraction for indirect causal chains. When the primary target variable does not explain a repeated failure, L2R expands the candidate repair context to include nearby mode, switch, or authorization variables that may sit in the causal chain leading to the violation. This fallback context is meant for cases where the chain passes through an intermediate state variable rather than terminating at the actuator named in the property. The fallback does not encode sample-specific fixes; every regenerated model is accepted only after NuSMV verification.

Stabilized fingerprint comparison. To detect repeated outputs reliably, regenerated models are normalized before comparison by removing comments and collapsing superficial whitespace, and the normalized text is used as a model fingerprint. A fingerprint here is therefore a normalized textual signature of the regenerated NuSMV model that is invariant to formatting-only changes. This is useful because LLMs often produce small formatting differences even when the underlying model logic is unchanged, and plain text equality would therefore underreport repeated-output stagnation.

3.6 Structural Guardrails for Regeneration

The repair prompt is constructed to keep the regenerated model close to the previous verified structure. In addition to naming the target transition block, the prompt explicitly marks safety-property specifications, variable declarations, and unrelated device transitions as preserved context. After regeneration, the output is accepted for the next verification step only if the extracted code is syntactically valid NuSMV and begins from the expected `MODULE main` structure. This prevents a repair attempt from replacing the model with explanatory text or with a partial code fragment.

Without such constraints, an LLM may appear to make progress by rewriting large portions of the model or by changing the property being checked, so the measured improvement would not reflect repair guidance alone.

4 Conceptual Testbed and Experimental Setup

4.1 Conceptual Smart-Home Testbed

To make the benchmark easier to interpret as a coherent system rather than a disconnected collection of rules, we organize the 40 samples around a simplified reference smart-home layout. Instead of presenting the benchmark as an abstract list of automation cases, we anchor representative sample categories to recognizable household spaces, including the kitchen, living room, bedroom, bathroom/service-side corner, entry area, and basement region. Figure 2 places representative devices within this layout to provide a concrete household context for appliance routines, media and activity behavior, quiet-time constraints, climate-control interactions, entry-side security logic, and hazard-response cases. The figure should be understood as a schematic functional zoning diagram rather than a strict architectural floor plan; some devices are placed in nearby rooms for compact presentation rather than to imply strict real-world deployment constraints.

The testbed covers six representative spaces and device contexts. The entry area covers door-access and security-oriented behavior such as smart locks, cameras, alarm signaling, and contact-sensor authorization logic. The living room covers activity-oriented behavior such as TV-centered media use and motion-triggered interaction. The bedroom covers quiet-time behavior and part of the climate-control setting through A/C, heater, window, and speaker. The kitchen covers appliance-oriented routines such as refrigerator and coffee-machine behavior. A bathroom or service-side corner is used as a compact placement for auxiliary environmental devices such as light and fan. The basement region covers hazard-response and utility-oriented behavior through leak sensors, water valves, and warning lights. This framing makes the 40 samples appear as one coherent benchmark rather than as unrelated rules, provides a natural way to explain policy categories such as access control, privacy, climate coordination, appliance cycles, and hazard response, and helps motivate hard cases in a realistic household context. The method itself is not tied to any particular floor plan;

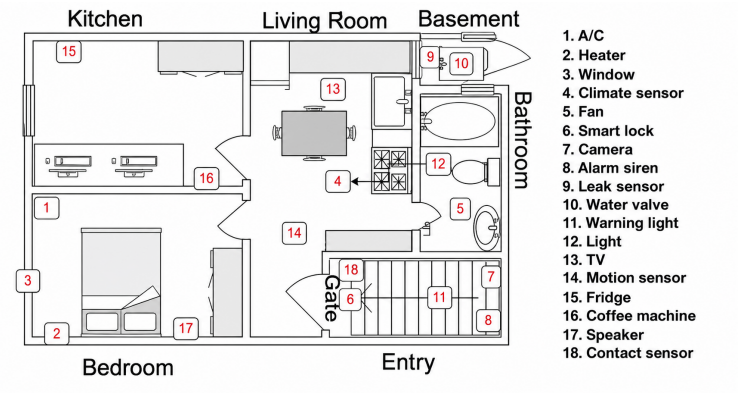


Fig. 2. Simplified reference smart-home layout used to ground the 40-sample benchmark in representative household spaces and device contexts.

the layout is used only to describe benchmark coverage and is not consumed by the repair pipeline.

4.2 System Implementation

The system is implemented as a TAPAssure-compatible layered workflow. In the core pipeline, Process 1 generates a NuSMV model from user requirements and performs syntax validation before passing the model to Process 2. When NuSMV identifies a violated property, the system invokes L2R to transform the raw counterexample into a structured repair prompt, then regenerates the model under the structural guardrails described in Section 3.6. The stagnation-aware broadening layer (Section 3.4) and the normalization-and-fallback layer (Section 3.5) operate on the same loop without replacing it.

For all experimental conditions reported in this paper, the LLM backend is fixed to `gpt-5.4-2026-03-05` so that the comparison across conditions is not confounded by model version. The same backend is used throughout the 5 runs of each condition for both Process 1 model generation and Process 2 repair-oriented regeneration. For reproducibility, all conditions use an LLM sampling temperature of 0.3 and the retry cap of 60 iterations inherited from TAPAssure.

4.3 Benchmark Construction

The benchmark contains 40 smart-home samples. The samples are manually constructed for this study to reflect common automation interactions and likely safety-relevant conflicts in household IoT settings, but their underlying safety-policy basis is adapted from IoTGuard [6]. Representative policy patterns are derived from IoTGuard-style safety categories and rewritten into TAPAssure-compatible automation scenarios with formal safety properties suitable for NuSMV-based verification and repair.

Of the 40 samples, 8 are safe cases (`gt_conflict` = 0) in which no property conflict exists and the repair stage is expected to converge quickly, while the remaining 32 are conflict-focused cases (`gt_conflict` = 1) that require active resolution of unsafe device interactions. The conflict-focused cases include both localized single-device violations and indirect multi-device chains. Each system condition is evaluated across all 40 samples for 5 independent runs, yielding 200 trials per condition; the conflict-focused subset contributes 160 trials per condition.

The benchmark is constructed to cover several policy categories, including access control, privacy, climate coordination, appliance cycles, and hazard response. This diversity matters because the repair stage should be evaluated not only on simple single-device violations, but also on cases involving indirect causal chains, coordinated behavior across multiple devices, and context-sensitive safety conditions.

4.4 Ablation Study Design

To disentangle the contribution of each principal algorithmic addition, we use a four-condition ablation. The conditions are:

- **Baseline:** the TAPAssure-style workflow that forwards raw NuSMV counterexamples to the LLM as repair feedback.
- **Baseline + L2R:** replaces raw counterexample prompting with structured Log-to-Repair guidance and minimal-patch-oriented regeneration.
- **Baseline + L2R + Broadening:** additionally enables conditional repair-context broadening when repeated no-progress behavior is detected.
- **Final System:** additionally enables normalization and fallback context, namely deterministic normalization of malformed intermediate models, fallback extraction for indirect causal chains, and stabilized fingerprint comparison.

This design separates the contribution of L2R prompting from the contributions of stagnation-aware broadening and of normalization-and-fallback handling on harder cases. As noted, all four conditions are evaluated with the same LLM backend so that observed differences are attributable to the repair-interface changes rather than to model version.

4.5 Metrics

The evaluation focuses on Process 2 behavior, since that is the part directly targeted by the proposed method. We report:

- **P1 success rate** and **P2 success rate**, computed over all 200 trials per condition.
- **Average P2 iterations**, computed over successful P2 trials.
- **Median P2 iterations**, computed within each run and then averaged across the 5 runs.

– **Mean Max P2 Iterations**, defined as the maximum P2 iteration count observed within each run, averaged across the 5 runs. This captures long-tail repair behavior that average iteration counts can hide.
– **Average runtime**, computed over successful P2 trials.
– **Median runtime**, computed within each run and then averaged across the 5 runs.

These metrics separate correctness from repair cost. Success rate measures whether the loop eventually produces a model accepted by NuSMV. Average and median iteration counts measure the normal repair burden, while mean max iterations capture rare long-tail behavior within each run. Runtime measures the practical cost of repeated regeneration and verification. Because the proposed method changes only Process 2, these metrics directly reflect the effect of the counterexample-to-repair interface.

5 Results and Analysis

This section reports the main experimental results and analyzes what they reveal about the layered repair design.

5.1 Motivation for Broadening and Hard-Case Support

Before reading the ablation tables, it is useful to describe what stagnation looks like in practice. In the baseline workflow, the repair prompt focuses tightly on the violated formula and a small tracked-variable set. This is enough for many cases, but some failures depend on variables outside that narrow local view. In those cases, the LLM can keep proposing small local edits that do not break the loop. Typical symptoms include the same violation pattern appearing again, a new model that is almost unchanged from the previous one, or repair bouncing between two nearby failure patterns. Two representative hard cases were especially informative when designing the layered method:

– `sample_07` involves coordinated climate-control behavior in which A/C and fan logic must remain consistent under temperature conditions; purely local repair tends to focus on one actuator while missing the coupled multi-device logic.
– `sample_31` involves an indirect access-control chain in which switch behavior, mode transitions, and unlock actions require an authorization or verified-presence context that may not be visible in a narrowly localized trace.

These cases illustrate why a purely local repair view can waste iterations, and why conditional widening and fallback context are useful only after such stagnation symptoms have appeared.

Table 1. Four-condition ablation on all 40 samples. P1/P2 success rates are computed over all 200 trials per condition. Average P2 iterations and average runtime are computed over successful P2 trials. Median values are first computed within each run and then averaged across the 5 runs. Mean Max P2 Iterations is the per-run maximum P2 iteration count averaged across the 5 runs.

Metric	Baseline	+ L2R	+ L2R + Broadening	Final System
P1 Success (%)	100.0	100.0	100.0	100.0
P2 Success (%)	99.0	97.5	95.0	98.0
Avg. P2 Iters	3.09	2.23	1.91	2.00
Median P2 Iters	2.00	1.80	2.00	1.40
Mean Max P2 Iters	27.6	9.8	4.8	7.8
Avg. Runtime (s)	13.68	14.56	11.55	9.67
Median Runtime (s)	6.48	8.68	6.81	7.69

5.2 Full 40-Sample Results

Table 1 reports the four-condition ablation on the full 40-sample benchmark. The results show a layered improvement pattern. The major reduction in repair iterations already appears when structured L2R guidance replaces raw counterexample prompting: average P2 iterations drop from 3.09 to 2.23, and mean max P2 iterations fall sharply from 27.6 to 9.8. This indicates that structured repair prompting largely removes the very long ineffective repair loops that dominate the baseline cost. However, P2 success decreases slightly to 97.5%, and average runtime rises to 14.56 s, suggesting that tighter localization can become too narrow for some harder cases.

Adding stagnation broadening further shortens repair length: average P2 iterations decrease to 1.91 and mean max P2 iterations fall to 4.8, showing that conditional widening is effective at preventing prolonged repair stagnation. However, this broadening-only condition does not improve overall robustness on its own; P2 success falls to 95.0%. Broader repair context can reduce repair myopia and shorten repair paths, but broadening alone can also weaken repair precision on some cases.

Adding normalization and fallback context on top of stagnation broadening yields the final system. In this condition, P2 success rises to 98.0%, while average runtime drops to 9.67 s. The final system therefore reduces average P2 iterations by 35.3% (3.09 to 2.00), average runtime by 29.3% (13.68 s to 9.67 s), and mean max P2 iterations by 71.7% (27.6 to 7.8) relative to the baseline, while keeping P2 success within one percentage point of the baseline.

An important caveat is that the average gains are tail-driven. Median runtime rises from 6.48 s to 7.69 s (+19%), and the +L2R condition raises average runtime to 14.56 s, because L2R adds per-iteration preprocessing: trace slicing, changed-variable extraction, and prompt construction occur before every regeneration call. Typical easy cases therefore pay a small per-iteration overhead, while the overall averages improve because structured guidance compresses the long-tail

Table 2. Four-condition ablation on conflict cases (`gt_conflict` = 1). P1/P2 success rates are computed over all 160 trials per condition. Average P2 iterations and average runtime are computed over successful P2 trials. Median values are first computed within each run and then averaged across the 5 runs. Mean Max P2 Iterations is the per-run maximum P2 iteration count averaged across the 5 runs.

Metric	Baseline	+ L2R	+ L2R + Broadening	Final System
P1 Success (%)	100.0	100.0	100.0	100.0
P2 Success (%)	99.4	97.5	94.4	99.4
Avg. P2 Iters	3.32	2.24	1.99	2.02
Median P2 Iters	2.00	1.90	2.00	1.50
Mean Max P2 Iters	27.0	8.4	4.6	7.8
Avg. Runtime (s)	13.83	15.63	12.73	9.64
Median Runtime (s)	6.90	8.89	7.01	7.80

repair loops that dominate baseline cost (mean max P2 iterations fall from 27.6 to 7.8). The headline runtime reduction should be read as a long-tail improvement rather than a uniform per-sample speedup.

The 1-percentage-point difference in P2 success warrants clarification. Across 200 trials (40×5 runs), the baseline recorded 2 failures in total, while the final system recorded 4. All additional failures occurred in the harder late-benchmark range and involved indirect multi-device causal chains that the L2R localization heuristic did not fully resolve within the retry limit. A two-proportion z-test (baseline: 198/200 successes; final system: 196/200 successes) gives $z = 0.82$ and $p = 0.41$, so the difference is not statistically significant at $\alpha = 0.05$. The main benefit is therefore improved repair efficiency with comparable success, not a higher overall success rate.

5.3 Conflict-Focused Results

Table 2 reports the same four-condition ablation on the conflict-focused subset, where Process 2 repair quality is stressed most directly. As in the full benchmark, L2R provides the main first-step efficiency gain: average P2 iterations decrease from 3.32 to 2.24 and mean max P2 iterations drop from 27.0 to 8.4. However, success decreases from 99.4% to 97.5%, indicating that tightly localized repair is not always sufficient for hard conflict cases.

Stagnation broadening further reduces repair length, bringing average P2 iterations down to 1.99 and mean max P2 iterations down to 4.6. However, conflict-case robustness drops further to 94.4%, showing that broader repair context alone is not enough to reliably resolve the hardest conflict-heavy samples.

When normalization and fallback context are added on top of stagnation broadening, the final system restores P2 success to 99.4%, matching the baseline on conflict cases, while keeping average P2 iterations low at 2.02 and reducing average runtime to 9.64 s. This is the strongest evidence for the paper's main

claim: layered counterexample-to-repair guidance reduces repair effort without reducing success on the cases where repair is most important. The final system therefore reduces average P2 iterations by 39.2% (3.32 to 2.02) and average runtime by 30.3% (13.83 s to 9.64 s) relative to the baseline on the conflict subset. As on the full benchmark, the median runtime increases slightly (6.90 s to 7.80 s) due to per-iteration preprocessing overhead, so the runtime gain on the conflict subset is likewise concentrated in the long tail.

5.4 Interpretation of the Results

The results support four conclusions. First, the verifier-to-repair interface is a real bottleneck: replacing raw counterexample prompting with structured L2R guidance already produces a large reduction in repair cost. Second, stagnation-aware broadening shortens repair length when the loop is trapped in repeated no-progress behavior. Third, broadening alone does not maintain repair success: its standalone use lowers success on both the full benchmark and the conflict subset, indicating a trade-off between broader context and repair precision. Fourth, normalization and fallback context stabilize the final system on harder cases. The final gains should therefore be understood as a layered result rather than as a single change: L2R provides the main structured repair improvement, broadening reduces repair myopia, and normalization-and-fallback handling recovers robustness on difficult late-benchmark cases.

5.5 Hard Cases and Remaining Challenges

The remaining hard cases also help explain why the added layers are needed. In `sample_07`, the climate-control property involves coordinated behavior across multiple devices and an underspecified middle temperature range. Such a case is difficult because purely local repair may focus on only one actuator while missing the coupled logic needed for consistent multi-device behavior.

In `sample_31`, the access-control property involves an indirect chain from a switch through a mode change to a door unlock. This kind of case requires the repair step to see authorization or presence-related context that may not be visible in a narrowly localized trace. As a result, a model can appear to repair one part of the logic while still failing to break the broader unsafe chain.

These cases suggest that some Process 2 failures arise not only from weak repair logic, but also from the interaction between abstract properties and incomplete local context. They also explain why broadening is applied only after repeated no-progress behavior is observed, rather than as the default prompt style, and why normalization and fallback context are applied on top of broadening rather than instead of it.

6 Discussion

The results suggest that verifier feedback should be treated as repair input, not only as diagnostic output. Raw counterexamples are enough to expose violations,

but they are poorly structured for model editing. L2R improves the repair loop by converting counterexamples into property-focused hints tied to specific transition blocks. The contribution is therefore not a redesign of TAPAssure, but a layered strengthening of its Process 2 repair interface in which each layer is observable in the ablation: L2R prompting reduces unnecessary rewrites, broadening shortens stalled paths, and normalization-and-fallback handling recovers robustness on harder cases.

The work has several limitations. The benchmark is manually constructed rather than drawn from a standardized public dataset, so broader evaluation is needed before making claims about general deployment settings. Some remaining failures may come from underspecified safety properties rather than from the repair procedure itself. The localization, broadening, and fallback mechanisms remain heuristic; NuSMV verifies the repaired model, but the method does not guarantee that the identified transition block is causally responsible for the violation or that the generated patch is formally minimal. The evaluation uses a single strong LLM backend over 5 runs; because the method is prompt-heavy, the gains may not transfer directly to weaker or locally hosted models, which matter for privacy-sensitive smart-home deployments. The efficiency gains are concentrated in the long tail: median runtime increases slightly, so typical easy cases pay a small preprocessing overhead. The current evaluation also focuses on repair iterations and runtime; direct metrics for patch size, token cost, and semantic preservation of user intent would help quantify how much unnecessary editing is avoided. Future work should evaluate larger and public benchmarks, validate across multiple LLM backends and model checkers, separate property underspecification from repair failure, make the stagnation trigger more adaptive, and report per-sample runtime distributions with paired significance tests.

7 Relation to Prior Work

Smart-home verification and repair. TAPChecker and TAPAssure combine LLM-based trigger-action rule generation with model checking [4,5]. TAPAssure is the direct experimental baseline: L2R retains its two-process workflow and NuSMV safety check, and changes the feedback supplied during Process 2. This lineage sits within a broader body of work on automation safety. Soteria verifies safety, security, and functional properties of individual and interacting IoT applications [7]; IoTSan uses model checking to expose interaction-level safety violations and attributes problematic behavior to applications [17]. Both motivate reasoning about the combined system, rather than treating each rule as independent. iRuler combines inferred rule dependencies with SMT solving and model checking to detect inter-rule vulnerabilities [23]. IoTCom extracts behavioral models and compositionally analyzes threats across cyber and physical channels [2]. These approaches broaden interaction analysis; L2R addresses the subsequent use of a detected violation to guide model repair.

Formal repair of smart-home rules also predates L2R. AutoTap translates desired properties into LTL and both synthesizes new rules and repairs existing

ones [30]. TAPFixer models physical and latency features and infers patches through negated-property reasoning with model abstraction and refinement [29]. L2R shares their goal of correcting unsafe automation, but addresses a different repair mechanism: it converts a failed NuSMV check into localized guidance for LLM regeneration. Its evaluation measures the cost of that interface within the TAPAssure workflow; it does not establish superiority over these symbolic repair systems.

Counterexample interpretation and LLM-based repair. Groce and Visser study how failing and successful executions can expose the essential information behind an error [14]. L2R shares the motivation of making counterexamples useful for diagnosis, but its property-focused trace window and changed-variable mapping are heuristics, not a proof of causal fault localization. More directly, Orvalho et al. combine MaxSAT-based fault localization, program sketches, and counterexample-guided LLM repair for introductory programming assignments [18]. That work already demonstrates the value of locating buggy statements before asking an LLM to repair them. L2R focuses on mapping smart-home model violations to candidate NuSMV transition blocks, with context widening after stagnation and normalization-and-fallback handling in the same loop.

ChatRepair interleaves patch generation with test feedback and uses earlier repair attempts to inform subsequent prompts [24]. InferFix combines static analysis with LLM-based repair, augmenting prompts with bug-type information and retrieved fixes [15]. Self-Debugging uses execution results and natural-language code explanations to refine generated programs [8]. RepairAgent lets an LLM autonomously interleave code inspection, repair-ingredient search, and patch validation through tools [3]. L2R follows a prescribed NuSMV repair workflow, adapting the counterexample context rather than delegating tool selection to an autonomous agent. These approaches establish the broader value of analysis-informed feedback and iterative repair. L2R's contribution is the specific representation and adaptation of model-checker feedback for TAPAssure-style smart-home models, evaluated through its layered ablation. It does not claim that verification feedback, localization, or repair history is new in itself.

Runtime protection and adjacent systems. IoTGuard enforces policies during execution [6], HAWatcher detects semantics-inconsistent behavior [13], and event-verification systems check trigger authenticity using video or edge-based evidence [11,19]. These mechanisms complement L2R's design-time repair of a formal model. Studies of delayed actions and timeout behavior [9,12], hidden device attributes [25], and unauthorized voice commands or voice liveness [26,28] provide additional security context, but address different failure mechanisms. LLM-assisted tasking and invariant extraction [1,27], as well as hybrid analysis of smart contracts [16], likewise concern adjacent stages or domains. The present work targets the repair loop after a smart-home model violates an already specified safety property.

8 Conclusion

This paper presented Log-to-Repair (L2R), a counterexample-to-repair step for TAPAssure-style LLM-guided IoT automation. L2R keeps the original verification-centered workflow but changes how failed verification results are passed to the LLM: raw NuSMV counterexamples are distilled into concise, transition-block-targeted repair prompts, with context widening and normalization-and-fallback handling engaged only when localized repair stalls.

Evaluated on a 40-sample smart-home benchmark, the final system reduces average Process 2 iterations from 3.09 to 2.00 and average runtime from 13.68 s to 9.67 s, with repair success statistically comparable to the baseline (98.0% vs. 99.0%, $p = 0.41$). On conflict-focused cases, it matches the baseline success rate of 99.4% while reducing average iterations from 3.32 to 2.02, with the gains concentrated in eliminating long-tail repair loops. Taken together, these results indicate that the practical effectiveness of verification-guided LLM repair depends not only on having a verifier, but also on how well the verifier's output is transformed into a repair signal the model can use.

Acknowledgment. This work was supported in part by the United States National Science Foundation (NSF) under grants CNS-2204785, CNS-2205868, 2409212, 2508118, and 2516003.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abshari, D., Shi, P., Fu, C., Sridhar, M., Du, X.: INVARLLM: LLM-assisted Physical Invariant Extraction for Cyber-Physical Systems Anomaly Detection (2024), arXiv preprint arXiv:2411.10918
2. Alhanahnah, M., Stevens, C., Chen, B., Yan, Q., Bagheri, H.: IoTCom: Dissecting interaction threats in IoT systems. IEEE Transactions on Software Engineering **49**(4), 1523–1539 (2023). https://doi.org/10.1109/TSE.2022.3179294
3. Bouzenia, I., Devanbu, P., Pradel, M.: RepairAgent: An autonomous, LLM-based agent for program repair. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 2188–2200. IEEE (2025). https://doi.org/10.1109/ICSE55347.2025.00157
4. Bui, H., Fu, C., Zuo, F.: TAPAssure: Safe Automation in Smart Homes with LLMs and Formal Verification (2025), accepted to EAI SmartSP
5. Bui, H., Lienerth, H., Fu, C., Sridhar, M.: Poster: TAPChecker: Model Checking in Trigger-Action Rules Generation Using Large Language Models. In: Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS) (2024)
6. Celik, Z.B., Tan, G., McDaniel, P.: IOTGUARD: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In: Proc. NDSS Symposium (2019)
7. Celik, Z.B., McDaniel, P., Tan, G.: Soteria: Automated IoT safety and security analysis. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 147–158. USENIX Association (2018), https://www.usenix.org/conference/atc18/presentation/celik

8. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. In: The Twelfth International Conference on Learning Representations (2024), https://openreview.net/forum?id=KuPixIqPiq
9. Chi, H., Fu, C., Zeng, Q., Du, X.: Delay Wreaks Havoc on Your Smart Home. In: Proc. IEEE Symposium on Security and Privacy (S&P) (2022)
10. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource tool for symbolic model checking. In: Computer Aided Verification (CAV 2002). Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
11. Fu, C., Du, X., Zeng, Q., Zhao, Z., Zuo, F., Di, J.: Seeing is Believing: Extracting Semantic Information from Video for Verifying IoT Events. In: Proc. 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) (2024)
12. Fu, C., Zeng, Q., Chi, H., Du, X., Valluru, S.L.: IoT Phantom-Delay Attacks: Demystifying and Exploiting IoT Timeout Behaviors. In: Proc. 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2022)
13. Fu, C., Zeng, Q., Du, X.: HAWatcher: Semantics-Aware Anomaly Detection for Appified Smart Homes. In: Proc. 30th USENIX Security Symposium. pp. 4223–4240 (2021)
14. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Model Checking Software (SPIN 2003). Lecture Notes in Computer Science, vol. 2648, pp. 121–136. Springer (2003). https://doi.org/10.1007/3-540-44829-2_8
15. Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., Svyatkovskiy, A.: InferFix: End-to-end program repair with LLMs. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1646–1656. ACM (2023). https://doi.org/10.1145/3611643.3613892
16. Li, X., Li, Z., Li, W., Zhang, Y., Wang, X.: No More Hidden Pitfalls? Exposing Smart Contract Bad Practices with LLM-Powered Hybrid Analysis. ACM Transactions on Software Engineering and Methodology **34**(6) (2025). https://doi.org/10.1145/3795692
17. Nguyen, D.T., Song, C., Qian, Z., Krishnamurthy, S.V., Colbert, E.J.M., McDaniel, P.: IotSan: Fortifying the safety of IoT systems. In: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies. pp. 191–203. ACM (2018). https://doi.org/10.1145/3281411.3281440
18. Orvalho, P., Janota, M., Manquinho, V.M.: Counterexample guided program repair using zero-shot learning and MaxSAT-based fault localization. Proceedings of the AAAI Conference on Artificial Intelligence **39**(1), 649–657 (2025). https://doi.org/10.1609/aaai.v39i1.32046
19. Reichenbach, T., Fu, C., Du, X., Di, J., Ji, Y.: TrustEvent: Cross-Platform IoT Trigger Event Verification Using Edge Computing. In: Proc. IEEE International Conference on Communications (ICC). pp. 4084–4089 (2024)
20. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907
21. Trimananda, R., Aqajari, S.A.H., Chuang, J., Demsky, B., Xu, G.H., Lu, S.: Understanding and automatically detecting conflicting interactions between smart

home IoT applications. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1215–1227. ACM (2020). https://doi.org/10.1145/3368089.3409682
22. Ur, B., McManus, E., Ho, M.P.Y., Littman, M.L.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 803–812. ACM (2014). https://doi.org/10.1145/2556288.2557420
23. Wang, Q., Datta, P., Yang, W., Liu, S., Bates, A., Gunter, C.A.: Charting the attack surface of trigger-action IoT platforms. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1439–1453. ACM (2019). https://doi.org/10.1145/3319535.3345662
24. Xia, C.S., Zhang, L.: Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 819–831. ACM (2024). https://doi.org/10.1145/3650212.3680323
25. Xu, X., Fu, C., Du, X., Luo, B.: Discovering and Exploiting IoT Device Hidden Attributes: A New Vulnerability in Smart Homes. In: Proc. ACM Conference on Computer and Communications Security (CCS) (2025)
26. Xu, X., Fu, C., Du, X., Ratazzi, E.P.: VoiceGuard: An Effective and Practical Approach for Detecting and Blocking Unauthorized Voice Commands to Smart Speakers. In: Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2023)
27. Yan, Z., Shao, S., Lin, H., Wu, T., Chen, X.: TaskSense: A Translation-like Approach for Tasking Heterogeneous Sensor Systems with LLMs. In: Proc. 23rd ACM Conference on Embedded Networked Sensor Systems (SenSys) (2025)
28. Yang, Q., Cui, K., Zheng, Y.: VoShield: Voice Liveness Detection with Sound Field Dynamics. In: Proc. IEEE International Conference on Computer Communications (INFOCOM) (2023)
29. Yu, Y., Xu, Y., Huang, K., Liu, J.: TAPFixer: Automatic detection and repair of home automation vulnerabilities based on negated-property reasoning. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 4945–4962. USENIX Association (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/yu-yinbo
30. Zhang, L., He, W., Martinez, J., Brackenbury, N., Lu, S., Ur, B.: AutoTap: Synthesizing and repairing trigger-action programs using LTL properties. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 281–291. IEEE (2019). https://doi.org/10.1109/ICSE.2019.00043