ACCEPTED BUT INVALID: WHEN REASONING BENCHMARKS CREDIT WRONG ANSWERS

Anonymous authors
Paper under double-blind review

ABSTRACT

When a reasoning benchmark marks an answer as correct, we usually assume that the submitted answer actually satisfies the task. We show that this assumption can break, even while the evaluator runs exactly as implemented. In MathConstruct, for instance, an archived model answer repeats an index in a required permutation, yet the released checker accepts it and awards credit. To see how far the problem extends, we craft small, explicit violations of the written task requirements and pass them through the released evaluation code. Across seven MathConstruct tasks, the released checkers accept all 127 applicable invalid constructions in our fixed cross-instance test, and the erroneous acceptance of five invalid archived model answers has already entered the task-weighted benchmark scores. DCP-Bench-Open reveals a further and subtler failure: omitted answer fields are left free for the evaluator's solver to complete. In the tested release, the evaluator accepts an empty JSON object for all 164 problems in our controlled completeness test, and accepts 735/735 incomplete outputs overall. Together, these findings show two ways an evaluator can award credit for a task the model never completed, i.e., by checking too little of the submitted answer, or by completing part of the answer itself.

1 INTRODUCTION

A reasoning benchmark asks a model to solve a problem and uses an evaluator to mark the answer. When the evaluator says "correct," we expect the model to have met the requirements of the question. In this paper, we check the other side of that exchange: **can the evaluator recognize an answer that should not pass?**

Consider an answer saved in the MathConstruct archive. The task requires each of 223 indices to appear exactly once. The model returns 224 entries, repeating index 1. The checker marks the answer correct. It checks that every required index is present, but compares the indices as sets, which discard duplicates. The answer also passes the circular-adjacency condition. What the checker misses is the difference between "every index is present" and "every index appears exactly once."

That difference matters because the accepted answer earns credit in the benchmark score. A reader looking only at the score sees a solved problem, not an extra index. The grade reflects what the program checked, even when that falls short of what the question asked.

Constructive benchmarks make this issue especially important (Balunovic et al., 2025; Michailidis et al., 2026). Many tasks admit multiple valid outputs, so evaluation must rely on executable parsers, binders, solvers, and checkers rather than exact-string matching. These components are therefore part of the measurement itself. A small omission in evaluation code can turn an invalid construction into a credited answer.

We study this problem in two constructive-reasoning benchmarks. We start from the written task, make a small change that breaks one explicit requirement, and pass the changed answer through the released evaluation path. In MathConstruct, this exposes seven checker failures across seven tasks, including missed distinctness, ordering, positivity, nonempty-witness, row-scanning, and multiplicity requirements. We then test the same failure mechanisms across additional official instances.

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.

We also ask whether these failures ever reached real benchmark results. In the recovered MathConstruct archive, we verify five accepted model outputs that violate explicit requirements and trace the corresponding credit into five task-weighted scores. DCP-Bench-Open exposes a different mechanism: instead of merely overlooking a submitted violation, the evaluator can leave missing output fields for its own solver to fill in.

Together, the two benchmarks show the same underlying problem at different stages of evaluation: **the condition implemented by the evaluator can be weaker than the task presented to the model.** Our practical recommendation is simple: benchmark evaluators should be tested with answers that are designed to fail specific written requirements, not only with examples that are expected to pass.

2 Related Work

Strengthening executable evaluation. EvalPlus expands code-generation tests with LLM- and mutation-generated inputs (Liu et al., 2023). UTBoost strengthens issue-related tests to detect patches that pass existing tests without resolving the underlying issue (Yu et al., 2025), and SWE-bench audits examine evaluation labels (Wang et al., 2026). VeRA and IFBench develop renewable or verifiable task families (Cheng et al., 2026; Pyatkin et al., 2025). Our study examines existing constructive tasks, asking whether their released evaluation paths enforce explicit requirements in the task statements.

Connecting written intent to executable checks. PAT-Agent, ChekProp, and CONSET connect informal intent to formal models, executable properties, or semantic tests (Zuo et al., 2025; Etemadi et al., 2025; Huang et al., 2026). Work on intent formalization likewise distinguishes machine-checkability from correspondence with the intended specification (Endres et al., 2024; Lahiri, 2024; Misu et al., 2026). We use task-derived predicates and minimized counterexamples, drawing on property-based testing (Claessen & Hughes, 2000), to make this correspondence testable in released benchmark evaluators.

Reliability of verifiers and benchmark scores. VerifyBench and SpecEval study verifier quality and specification–judgment consistency (Yan et al., 2026; Ahmed et al., 2026). SoundnessBench, verifier reward-hacking studies, and Verus-SpecGym investigate counterexamples or accepted rule violations (Zhou et al., 2025; Helff et al., 2026; Agarwal et al., 2026). Our focus is the path from a submitted construction to benchmark credit. MathConstruct and DCP-Bench-Open provide the two evaluation settings (Balunovic et al., 2025; Michailidis et al., 2026). Within these settings, we connect clause-level failures to repairs and, where matching historical records are available, to the scores that incorporated the erroneous verdicts.

3 Testing What the Evaluator Actually Accepts

3.1 The acceptance contract

The basic question is whether the evaluator can accept a response that violates the written task. Let t denote a task, z an instance, and r a submitted response. The evaluator selects an answer with E_t, parses or binds it with P_t, and produces a verdict with V_t; $\perp$ denotes parsing or binding failure. Its accepted-response set is

$$A_t(z) = \{r : E_t(r) \text{ is defined},\ P_t(E_t(r)) \neq \perp,\ V_t(z, P_t(E_t(r))) = 1\}. \tag{1}$$

Let $S_t(z)$ be the responses whose submitted construction is complete and satisfies the necessary written requirements. A sound evaluator should satisfy

$$A_t(z) \subseteq S_t(z). \tag{2}$$

A counterexample is therefore simple: find a response that the released evaluator accepts even though its submitted construction violates a necessary written requirement. The rest of the paper builds such counterexamples and follows them through evaluation.

3.2 Constructing answers that should fail

For each audited requirement, we implement a small independent check derived from the task statement. We then start from an accepted official construction and make one targeted change: duplicate an element, remove a required witness, collapse a strict inequality, change the sign of a value, or otherwise break the chosen requirement.

We keep a candidate only when it still has the expected structure, reaches the official parser, violates the target requirement, and preserves the recorded non-target conditions. This makes the counterexample easy to interpret: the answer is rejected by the written requirement but accepted by the released evaluator.

For MathConstruct, candidates are serialized in the benchmark's normal answer format and reparsed through the official parser before checker evaluation. We therefore test the released response path rather than calling checker functions only on hand-constructed internal objects.

3.3 Checking the identified failure

Once a counterexample is found, we add the missing check and rerun the same answer. The purpose is to verify that the identified condition explains the false acceptance. We also rerun official constructions and nearby valid controls as regression tests. These controls are supporting evidence for the diagnosis; they are not a separate headline result.

Fixed mutations then test whether the same omission recurs across additional official-generator instances. Historical analysis asks a different question: whether saved model outputs encountered an omission and received credit. For that analysis, we reconstruct the historical answer-selection and checking path, match each verified false accept to its aggregate record, and replace only the confirmed erroneous verdict.

Figure 1 summarizes the two failure stages and their connection to scores.

4 Empirical Study

4.1 Evaluation settings

Our MathConstruct audit fixes both a paper-era revision and a later maintained revision. The paper-era repository imports 126 task classes; a September 2025 maintenance update removed five problems and corrected five verification functions (ETH SRI, 2025). All seven checker failures reported here are still present in this post-maintenance revision. We inventory all 121 checkers in the maintained snapshot and manually inspect 10 tasks. The source audit records 15 mutations or controls, followed by a fixed cross-variation test of the seven confirmed failures.

For historical analysis, we recover official logs and reproduce the matching paper-era evaluation path. The saved-conversation path selects the last parseable eligible message in reverse order before applying the historical parser and checker.

For DCP-Bench-Open, we fix release v0.2.2 and derive controlled deletions from accepted released-example output objects. We compare released evaluation with an exact-schema gate and retain a complete valid control for every problem.

4.2 MathConstruct: invalid answers pass because of a missing requirement

Seven audited MathConstruct tasks accept constructions that violate explicit requirements in the problem statement. A BxMO example makes the issue visible immediately. The task requires $1 < m < n$. Replacing the official pair $(2, 8)$ with $(2, 2)$ gives

$$((2, 2), (6, 48), (14, 224)).$$

The answer still has the expected format and still passes the released checker. The checker verifies other properties of the pair, but never checks the strict inequality $m < n$. The implementation is not crashing or behaving nondeterministically; it is simply checking less than the task asks for.

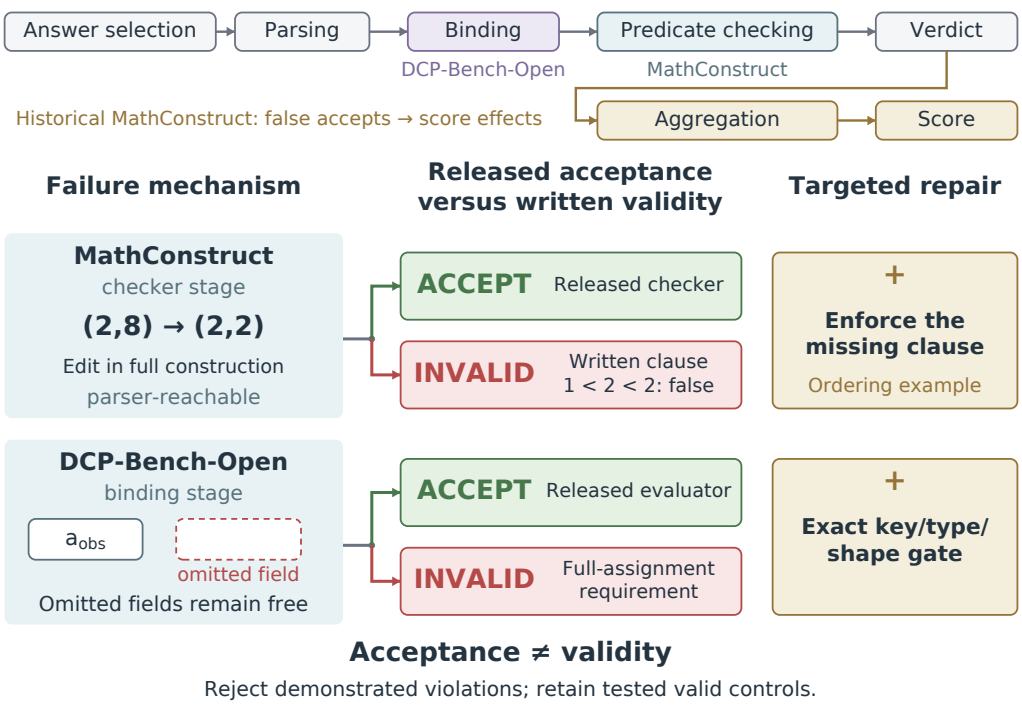


Figure 1: Two ways to accept an answer that does not meet the task. MathConstruct can accept a well-formed construction while failing to enforce a required condition. DCP-Bench-Open can leave omitted fields free for its solver to complete. Targeted checks reject the demonstrated violations and preserve the tested valid controls. The historical MathConstruct audit additionally traces false accepts into archive scores.

Task	Written requirement violated by the retained mutation	Released	Repaired
BMO 2015 N7	Four different positive integers: duplicate a member of a quadruple.	Accept	Reject
BxMO 2011	Strict ordering: replace $(2, 8)$ with $(2, 2)$.	Accept	Reject
EMC 2022	At least one divisor witness: delete the sole divisor.	Accept	Reject
IMO 2000 C4	No forbidden empty run in any row or column: move one pawn to create an invalid row.	Accept	Reject
IMO 2017 N6	Positive rational values: replace $1/2$ with $-1/2$.	Accept	Reject
ToT 2018	Positive fractions: negate one fraction.	Accept	Reject
USAMO 2005	A permutation: append a duplicate index.	Accept	Reject

Table 1: Seven MathConstruct tasks in which a small, parser-reachable mutation violates an explicit written requirement but is accepted by the released checker. Adding the corresponding missing check rejects each counterexample. Full constructions appear in the appendix.

Figure 2 shows this missed ordering check alongside the missing-field problem in DCP-Bench-Open.

The other failures follow the same pattern in different forms. Some checkers inspect the values but miss a relation between them; one accepts an empty witness because the implemented product and universal condition are both vacuously satisfied; one scans the wrong row; two omit positivity; and one converts a sequence to a set, erasing multiplicity.

Table 1 summarizes the seven task-level counterexamples.

All seven witnesses use ordinary finite values at the expected type and nesting depth. Appendix B contains the complete mechanism atlas, as illustrated in Figure 4.

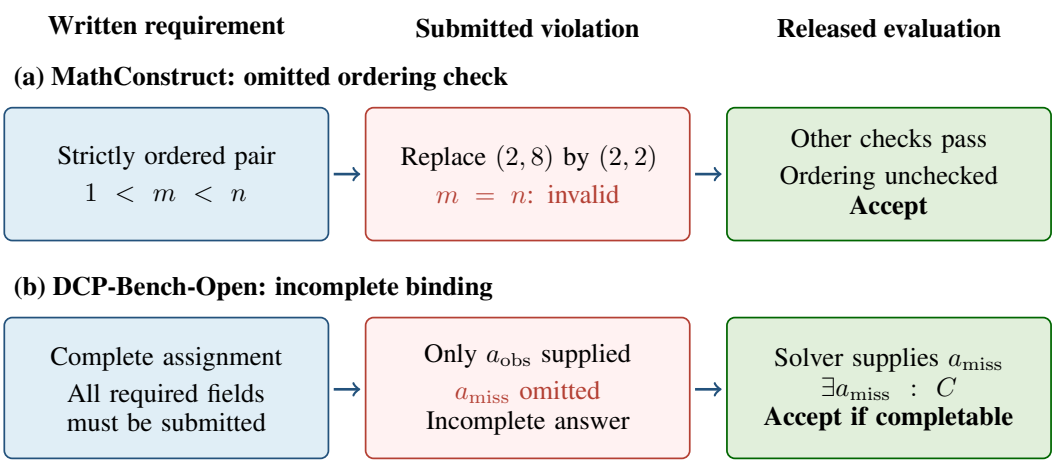


Figure 2: Two ways an evaluator can accept an invalid answer. In MathConstruct, the submitted construction violates a required ordering condition that the checker never tests. In DCP-Bench-Open, the submission omits required fields and the evaluator leaves those values for its solver to choose.

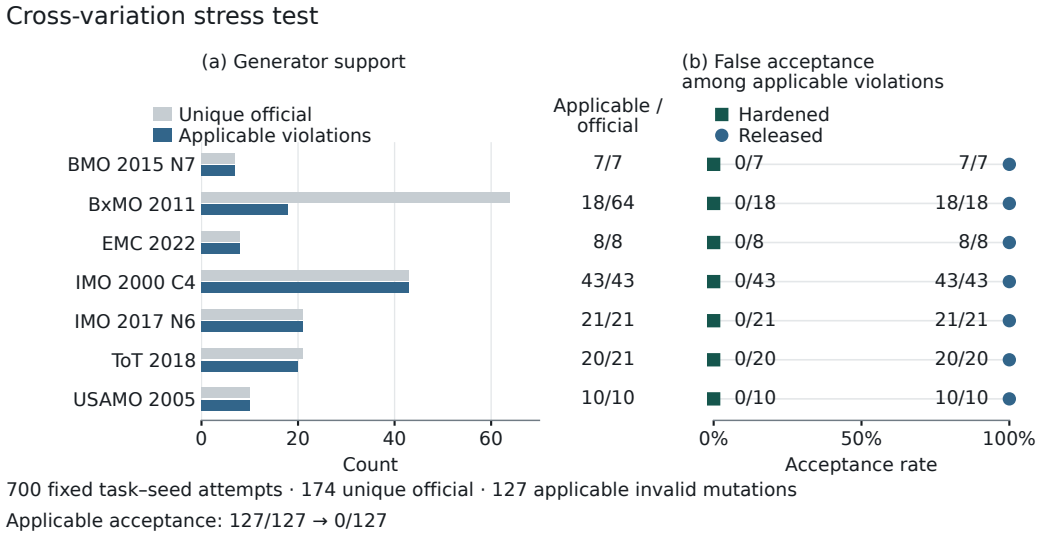


Figure 3: Cross-instance recurrence of the seven MathConstruct failures. The fixed mutation operators produce 127 applicable invalid constructions across 174 unique official instances. Released checkers accept all 127; adding the missing checks reduces acceptance to 0/127.

Adding the missing condition changes the verdict in every case: each counterexample is rejected after the corresponding check is added. The complete regression results, including official constructions and nearby valid controls, are reported in Appendix B.

We next ask whether these failures are limited to the first hand-inspected examples. Using one fixed mutation per task over official-generator seeds 0–99 yields 174 unique official instances and 127 mutations that satisfy the pre-check criteria. The released checkers accept all 127 applicable invalid constructions. After adding the missing conditions, none of the 127 is accepted.

The recurrence differs by task because some mutations are not applicable to every generated instance. Figure 3 reports the per-task support. Regression results on the corresponding official constructions are given in Appendix B.

4.3 HISTORICAL MATHCONSTRUCT OUTPUTS: WRONG ANSWERS RECEIVED REAL BENCHMARK CREDIT

The controlled mutations show that the checkers can fail. The archive shows that they did fail on model outputs that received benchmark credit.

We verify five such answers. One is the USAMO example from the introduction: the task requires a permutation of 223 indices, but the saved answer contains 224 entries because index 1 appears twice. No required index is missing, and the circular-adjacency condition passes. Set conversion hides the duplicate, so the historical checker accepts the answer.

The other four cases occur in an EMC task that asks for distinct lovely numbers together with divisor witnesses. The checker distinguishes complete witness sequences but does not require the numbers represented by those sequences to be distinct. In one archived answer, eleven different witness sequences describe only three distinct lovely numbers. The arithmetic checks pass, but the answer does not provide the eleven different numbers requested by the task.

This second EMC condition is different from the nonempty-witness condition in the source audit. It was discovered in the historical records and is analyzed separately.

We then match each false accept to its original aggregate and recompute the task-weighted score after changing only that verdict. Each correction lowers the affected aggregate by 0.1969–0.1984 percentage points, which can be observed in Table 2. The numerical change is modest, but the path is direct: an answer that violated the task received credit, and that credit entered the reported measurement.

Archive collection	Model	Original score (%)	Corrected score (%)
`cot-parser-main`	Flash-Thinking	27.0503	26.8519
`code-parser-main`	GPT-4o-mini	6.9444	6.7460
`cot-rephrase`	QwQ	12.7953	12.5984
`cot-rephrase`	o1-mini	20.4724	20.2756
`cot-rephrase`	Flash-Thinking	28.5433	28.3465

Table 2: Task-weighted archive scores before and after correcting the five verified false accepts, one in each model/collection pair. Reductions are 0.1969–0.1984 percentage points. The Flash-Thinking `cot-parser-main` baseline matches the 27.05% printed in the original benchmark paper; the other rows are verified archive aggregates, not separately verified printed results.

Appendix H provides the mathematical certificates, exact aggregates, and publication linkage.

4.4 DCP-BENCH-OPEN: THE EVALUATOR COMPLETES WHAT THE ANSWER LEAVES OUT

A false accept does not have to come from a missing mathematical condition. In DCP-Bench-Open, the evaluator can instead do part of the task on behalf of the submitted answer.

The benchmark (Michailidis et al., 2026) asks the generated model to output a complete assignment to declared variables. The evaluator parses the submitted JSON object, fixes the fields that are present, and then solves the ground-truth constraint model. If a required field is missing, no equality is added for that variable. The solver is free to choose its value.

Let a_{obs} denote the values reported by the submission and a_{miss} the required values it omits. For satisfaction problems, the released behavior is therefore

$$V_{\mathrm{DCP}}(a_{\mathrm{obs}}) = 1 \quad \Longleftrightarrow \quad \exists a_{\mathrm{miss}} : \, C(a_{\mathrm{obs}}, a_{\mathrm{miss}}), \tag{3}$$

This is a different question from the one posed to the model. The task asks the model to provide the full assignment; the evaluator checks whether the partial assignment can be completed into one. Optimization problems also retain the released objective comparison.

The simplest test is an empty JSON object. It supplies none of the required variable values. Yet the released evaluator accepts an empty object for every one of the 164 problems in our controlled test.

We also delete one required field at a time, or retain only one required field. Across the three deletion families, the released evaluator accepts all 735 incomplete outputs. A pre-binding schema check that requires the declared keys, types, and shapes rejects all 735.

A separate semantic review also finds one DCP problem whose released model omits a required degree-ordering condition; details are provided in Appendix G.

5 Discussion

5.1 A reproducible grade can still be the wrong grade

Replaying the historical MathConstruct evaluation reproduces the original acceptance. That tells us which computation produced the score, but not whether the computation implemented the task correctly. The repeated-index example makes the distinction concrete: the evaluation is reproducible, the code behaves as written, and the answer is still not a permutation.

This matters because benchmark scores inherit the evaluator's definition of success. If the implementation checks a weaker condition than the prompt, the score measures success on that weaker condition.

5.2 Correct examples are not enough to test an evaluator

Correct examples alone do not reveal these differences. Give a checker a valid permutation, and both set comparison and exact permutation checking will accept it. Give the DCP evaluator a complete assignment, and both complete-answer validation and solver completion can accept it. The difference appears only when an answer contains a repeated index or a missing field.

This is why task-derived counterexamples are useful regression tests. Each one asks a simple question: if this written requirement is broken while the rest of the answer remains well formed, does the evaluator notice? A ComBench example shows another way a wrong answer can pass. The verifier's general checks use ascending team lists, so they miss a rule that gives the wrong result when the same teams are written in reverse order (Appendix I).

5.3 Check the answer the model actually submitted

The DCP result highlights another design principle: evaluation should preserve the boundary between what the model supplied and what the evaluator inferred or completed. Parsing, binding, normalization, and solver calls can silently change the object being judged.

Benchmark releases should therefore retain the submitted artifact and test each major written requirement with at least one answer that violates it. For constructive tasks, rejection tests are as important as acceptance tests.

Limitations

Our MathConstruct semantic review is targeted rather than exhaustive: we manually inspect 10 tasks among 121 inventoried checkers, and the historical analysis covers three recovered archive collections. The controlled mutations and DCP deletions test specific failure mechanisms; they do not estimate how often arbitrary model outputs trigger those mechanisms. Historical recovery is incomplete, and the five corrected archive scores are not sufficient to evaluate ranking changes. The DCP schema check is based on the representations used in the released examples, so other valid encodings may require separate treatment. The independent requirement checks are computational rather than expert-annotated. These limitations motivate broader audits, but they do not affect the concrete counterexamples reported here.

6 Conclusion

A benchmark should give credit for the answer the model actually submitted, and judge against the requirements the task actually states. Our study shows two ways this principle can fail. Math-

Construct's checkers can overlook required properties such as ordering, positivity, distinctness, and exact multiplicity, letting invalid answers through. DCP-Bench-Open, by contrast, can accept an incomplete submission, leaving missing values for its own solver to fill in. These are not merely hypothetical risks, and five archived MathConstruct answers show that such evaluator errors can also reach recorded benchmark scores.

The practical lesson is simple: test the evaluator with answers that should fail. A valid reference answer shows that the grader can accept something correct; a targeted counterexample shows whether it can reject something that violates the task.

ETHICS AND REPRODUCIBILITY

Ethics statement. This study analyzes public benchmark artifacts, locally generated model outputs, and anonymized technical reviews. No sensitive personal data were collected; reviewer identities and private correspondence are not released. The institutional office determined that the review activity was not human-subjects research.

Reproducibility statement. The public supplement separates the MathConstruct, DCP-Bench-Open v0.2.2, and ComBench evidence. A separate historical-evidence supplement contains saved records, source versions, and verification code. `ARTIFACT_README.md` maps all packages. Appendices F, G, H, and I describe the replay, DCP tests, historical reconstruction, and ComBench example, along with the materials available to reproduce them.

REFERENCES

Anmol Agarwal, Natalie Neamtu, Pranjal Aggarwal, Seungone Kim, Jannis Limperg, Cedric Flamant, Kanna Shimizu, Bryan Parno, and Sean Welleck. Verus-SpecGym: An agentic environment for evaluating specification autoformalization, 2026. URL https://arxiv.org/abs/2605.26457.

Ahmed M. Ahmed, Kevin Klyman, Yi Zeng, Sanmi Koyejo, and Percy Liang. SpecEval: Evaluating model adherence to behavior specifications. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=VzLIQ3Lqm9.

Mislav Balunovic, Jasper Dekoninck, Nikola Jovanović, Ivo Petrov, and Martin Vechev. MathConstruct: Challenging LLM reasoning with constructive proofs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 2742–2769. PMLR, 2025. URL https://proceedings.mlr.press/v267/balunovic25a.html.

Zerui Cheng, Jiashuo Liu, Chunjie Wu, Jianzhu Yao, Pramod Viswanath, Ge Zhang, and Wenhao Huang. VeRA: Verified reasoning data augmentation at scale, 2026. URL https://arxiv.org/abs/2602.13217.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 268–279. Association for Computing Machinery, 2000. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024. doi: 10.1145/3660791. URL https://doi.org/10.1145/3660791.

Khashayar Etemadi, Marjan Sirjani, Mahshid Helali Moghadam, Per Strandberg, and Paul Pettersson. LLM-based property-based test generation for guardrailing cyber-physical systems, 2025. URL https://arxiv.org/abs/2505.23549.

ETH SRI. MathConstruct official repository. GitHub repository, 2025. URL https://github.com/eth-sri/mathconstruct. September 2025 maintenance update.

Lukas Helff, Quentin Delfosse, David Steinmann, Ruben Härle, Hikaru Shindo, Patrick Schramowski, Wolfgang Stammer, Kristian Kersting, and Felix Friedrich. LLMs gaming verifiers: RLVR can lead to reward hacking, 2026. URL https://arxiv.org/abs/2604.15149.

Qiqing Huang, Xingyu Wang, Wanda Guo, Guofei Gu, and Hongxin Hu. Semantics over syntax: Uncovering Pre-Authentication 5g baseband vulnerabilities. In *35th USENIX Security Symposium*. USENIX Association, August 2026. URL https://www.usenix.org/conference/usenixsecurity26/presentation/huang-qiqing.

Shuvendu K. Lahiri. Evaluating LLM-driven user-intent formalization for verification-aware languages. In *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design*, pp. 142–147, 2024. doi: 10.34727/2024/isbn.978-3-85448-065-5_19. URL https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_19.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0943. URL https://papers.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

Kostis Michailidis, Dimosthenis Tsouros, and Tias Guns. DCP-Bench-Open, 2025. URL https://doi.org/10.5281/zenodo.17800138.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. DCP-Bench-Open: Evaluating LLMs for constraint modelling of discrete combinatorial problems, 2026. URL https://arxiv.org/abs/2506.06052.

Md Rakib Hossain Misu, Iris Ma, and Cristina V. Lopes. VeriAct: Beyond verifiability—agentic synthesis of correct and complete formal specifications, 2026. URL https://arxiv.org/abs/2604.00280.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hanna Hajishirzi. Generalizing verifiable instruction following. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-1645. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/46499a0622ecf568b72d17b61e45dbd5-Paper-Datasets_and_Benchmarks_Track.pdf.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. In *2026 IEEE/ACM 48th International Conference on Software Engineering*. ACM, 2026. doi: 10.1145/3744916.3764576. URL https://doi.org/10.1145/3744916.3764576.

Yuchen Yan, Jin Jiang, Zhenbang Ren, Yijun Li, Xudong Cai, Yang Liu, Xin Xu, Mengdi Zhang, Jian Shao, Yongliang Shen, Jun Xiao, and Yueting Zhuang. VerifyBench: Benchmarking reference-based reward systems for large language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=JfsjGmuFxz.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Shunkai Zhang, Haoran Zhang, Yun Luo, Qianjia Cheng, Haodi Lei, Yizhuo Li, Runzhe Zhan, Zhilin Wang, Bangjie Xu, Yucheng Su, Xinmiao Han, Xiaoye Qu, Dongrui Liu, Zhouchen Lin, Yu Qiao, Ning Ding, Yafu Li, and Yu Cheng. ComBench: A benchmark for rigorous proof reasoning and constructive realization in olympiad-level combinatorics. *arXiv preprint arXiv:2606.10479*, 2026. URL https://arxiv.org/abs/2606.10479.

Xingjian Zhou, Keyi Shen, Andy Xu, Hongji Xu, Cho-Jui Hsieh, Huan Zhang, and Zhouxing Shi. SoundnessBench: A soundness benchmark for neural network verifiers. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=UuYYldVLH3.

Xinyue Zuo, Yifan Zhang, Hongshu Wang, Yufan Cai, Zhe Hou, Jing Sun, and Jin Song Dong. PAT-Agent: Autoformalization for model checking, 2025. URL https://arxiv.org/abs/2509.23675. Accepted at the 40th IEEE/ACM International Conference on Automated Software Engineering.

APPENDIX

STATEMENT OF LLM USAGE

LLMs assisted with parts of the implementation and analysis, manuscript polishing, literature discovery, and error checking. We reviewed and verified all LLM-assisted material and take responsibility for the final content.

A COMPLETE COUNTEREXAMPLE RECORDS

Each construction below is defined completely as an exact mutation of the named task's official construction at commit `3e51d847547a6586237ae55b83131b0e31b8476d`. This representation avoids duplicating hundreds of unchanged integers. The portable file `repro/cases/counterexamples.json` contains the exact full serialized boxed candidates and provenance. The minimized mutation used for each regression appears in `repro/cases/minimized_cases.json`.

A.1 MC-CURRENT-01: WITHIN-QUADRUPLE DUPLICATION

Task: `bmo-shortlist-2015-n7`. In the first official quadruple, replace its fourth value with its first value, $1428570058823529411 7647$. The mutated quadruple is

$$(1428570058823529411 7647,\ 1428570117647058823529 4,\\ 1428570176470588235294 1,\ 1428570058823529411 7647).$$

All other quadruples are unchanged. The public parser reaches the construction and the released checker accepts it. The statement requires four different positive integers. The independent clause predicate $|\mathrm{set}(q)| = 4$ rejects the first quadruple. Hardening adds that condition after the existing length-four check. The valid control reverses the order of the official quadruple collection.

A.2 MC-CURRENT-02: COLLAPSED STRICT INEQUALITY

Task: `bxmo-2011-1`. Replace the first official pair $(2, 8)$ with $(2, 2)$. The complete candidate is $((2, 2), (6, 48), (14, 224))$. The released checker accepts even though $1 < 2 < 2$ is false. Hardening rejects $m \geq n$. The valid control reverses the official pair collection.

A.3 MC-CURRENT-03: ZERO-LENGTH DIVISOR WITNESS

Task: `emc-2022-2`. Replace the first official witness $[1, 1]$ with $[1]$ and leave the other nine witnesses unchanged. The released checker evaluates the empty product as one and the empty universal condition as true. The statement requires positive k, while the independent clause predicate requires each witness to contain at least $[n, d_1]$. Hardening requires length at least two. The valid control appends an additional divisor 1 to the first official witness, giving $[1, 1, 1]$, which is permitted because the d_i may repeat.

A.4 MC-CURRENT-04: UNSCANNED INVALID ROW

Task: `imo-shortlist-2000-c4`, with $n = 10$ and $k = 5$. Starting from the official board, move the pawn in column 4 from row 0 to row 1: set $(0, 4)$ from 1 to 0 and $(1, 4)$ from 0 to 1. The

complete minimized board is

$$\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}$$

The move preserves 20 pawns and every column but creates nine adjacent zeros in row 0. The current stale-variable row scan accepts. An independent row/column scan rejects. Hardening builds each row string from `x[i]`. The valid control rotates the official board by 180 degrees.

A.5 MC-CURRENT-05: NEGATIVE RATIONAL

Task: `imo-shortlist-2017-n6`.

In the first official triple, replace $1/2$ by $-1/2$, yielding $(1/6, 1/3, -1/2)$, and leave the other nine triples unchanged. The direct sum is 0 and the reciprocal sum is 7, so the algebraic checks accept. Positivity fails. Hardening rejects nonpositive values. The valid control swaps the first two positive values in the first triple.

A.6 MC-CURRENT-06: NEGATIVE FRACTION

Task: `tot-2018-1`. Replace the sixth official fraction with $-18143999/21772806$ and leave the other nine fractions unchanged. The complete list is

$$\begin{aligned}
&(\tfrac{3628802}{3628801}, \tfrac{3628803}{7257602}, \tfrac{3628804}{10886403}, \\
&\tfrac{3628805}{14515204}, \tfrac{3628806}{18144005}, \tfrac{-18143999}{21772806}, \\
&\tfrac{3628808}{25401607}, \tfrac{3628809}{29030408}, \tfrac{3628810}{32659209}, \\
&\tfrac{3628811}{36288010}).
\end{aligned}$$

The released checker accepts the denominator and irreducibility relations, but the sign predicate rejects. Hardening requires every fraction to be positive. The valid control reverses fraction order.

A.7 MC-CURRENT-07: DUPLICATED PERMUTATION INDEX

Task: `usamo-2005-1`. Append a second copy of the first index, 2, to the 255-entry official permutation. The resulting candidate has 256 entries and ends with the original final index followed by 2. Its exact serialized list is retained in `repro/cases/counterexamples.json`. Set equality hides the duplicate, so the released checker accepts. The independent clause predicate requires exactly 255 indices with the required set. Hardening adds the exact-length check. The valid control cyclically rotates the official permutation.

A.8 MINIMALITY AND PARSER REACHABILITY

Six cases are a single replacement, deletion, or insertion from a valid official construction. The board case requires two cell replacements because the released checker enforces an exact pawn count: a single cell flip changes that count. All seven candidates are serialized in final-boxed-answer form and reparsed at the expected type and depth before adjudication. The raw case provenance retains the original four-cell pawn witness. The minimized evidence table and regression suite use the two-cell witness above.

B Checker Logic, Predicates, and Patch Summaries

Table 3 summarizes what each released checker tests, which written requirement is missing, and the targeted check used in our regression test.

Task	Relevant current logic	Separately implemented clause predicate and hardening
`bmo-shortlist-2015-n7`	Checks entry length, sorts each entry, and checks uniqueness of whole entries.	Predicate: four distinct values per entry. Patch: add `len(set(entry)) != 4` rejection.
`bxmo-2011-1`	Bounds m, sorts by n, and compares prime-factor sets.	Predicate: $1 < m < n$. Patch: reject `m >= n`.
`emc-2022-2`	Computes `prod(seq[1:])` and a universal condition over the same suffix.	Predicate: witness length at least two. Patch: reject shorter witnesses.
`imo-shortlist-2000-c4`	Counts pawns, scans a stale `row` in the row loop, then scans columns correctly.	Predicate: full row/column scan. Patch: construct the row string from `x[i]`.
`imo-shortlist-2017-n6`	Checks digit-size condition and integer direct/reciprocal sums.	Predicate: every rational > 0. Patch: reject nonpositive triple members.
`tot-2018-1`	Checks distinct denominators, irreducibility, and difference denominators.	Predicate: every fraction > 0. Patch: reject nonpositive fractions.
`usamo-2005-1`	Compares submitted indices and mapped divisors as sets, then checks cyclic adjacency.	Predicate: exact cardinality and required index set. Patch: require the exact number of indices before set comparison.

Table 3: Released checker logic, independently implemented requirement checks, and targeted hardening for the seven MathConstruct gaps.

The patch targets the audited public commit and changes only the seven checkers while adding one focused regression module. Parser behavior, scoring, generation, and unrelated checker logic remain unchanged. `ARTIFACT_README.md` records the source lock and patch digest.

Across the 700 task–seed attempts, 174 unique official constructions yield 127 applicable mutations. Both released and repaired checkers accept all 174/174 official constructions. Each repaired checker also retains its nearby valid control; the combined seven-patch regression retains all 121 official constructions in the maintained snapshot. The BxMO mutation applies to 18 of 64 instances because the other 46 transformed objects fail a recorded prime-divisor/list condition. The ToT mutation applies to 20 of 21 instances because it is undefined at $n = 5$; the remaining five operators apply across their unique tested instances.

Table 4 reports the regression results on the corresponding official constructions.

Task	Released	Repaired
BMO 2015 N7	7/7	7/7
BxMO 2011	64/64	64/64
EMC 2022	8/8	8/8
IMO 2000 C4	43/43	43/43
IMO 2017 N6	21/21	21/21
ToT 2018	21/21	21/21
USAMO 2005	10/10	10/10

Table 4: Official-construction retention in the cross-instance regression. Each denominator is the number of unique official constructions tested for that task.

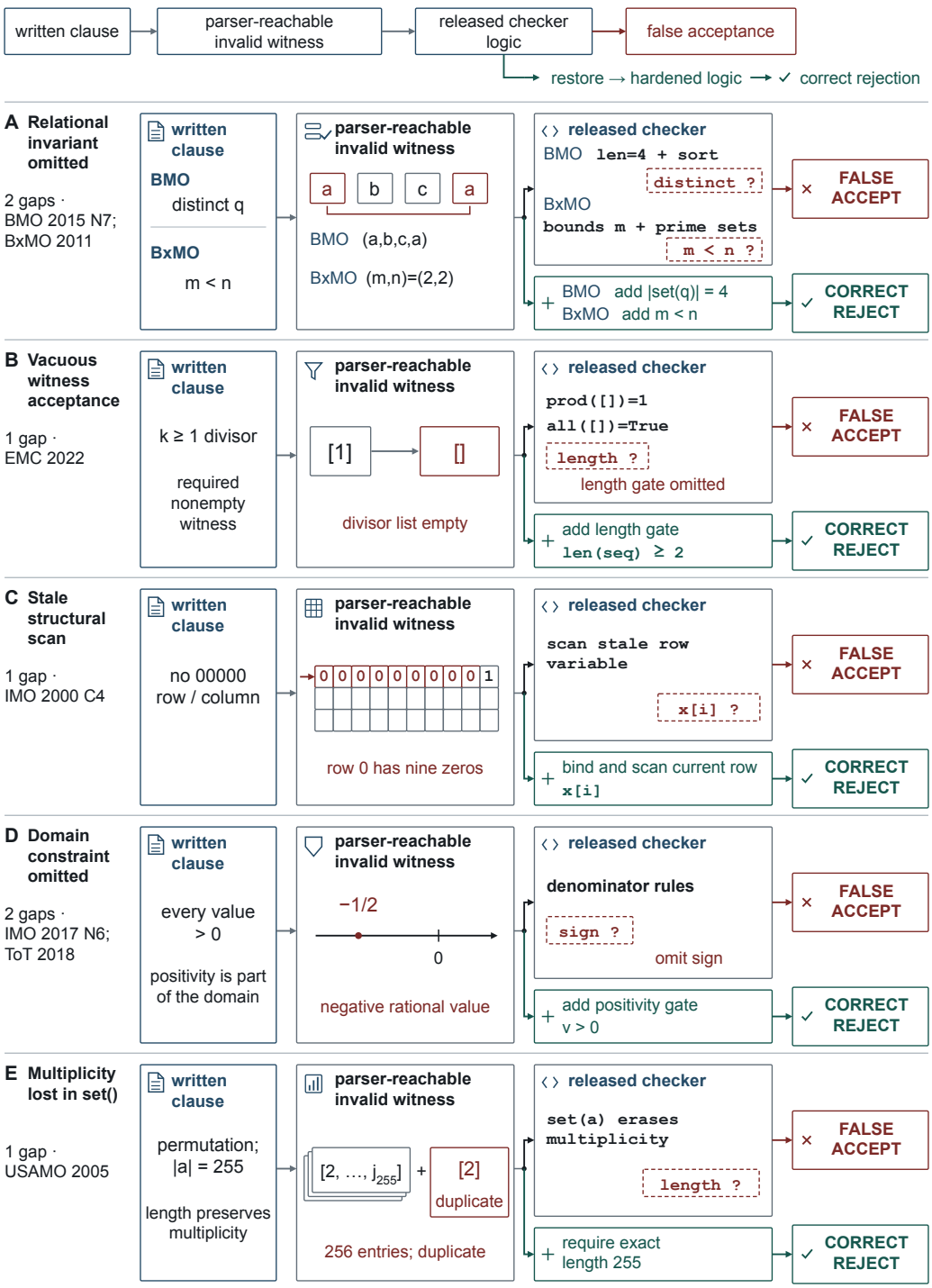


Figure 4: How seven MathConstruct requirements are lost in five implementation patterns. Each row connects a written requirement to an accepted invalid construction and the check that corrects its verdict. The patterns are omitted relational conditions, vacuous witness acceptance, a stale row scan, omitted positivity checks, and multiplicity lost through set conversion.

C Reproducibility Protocol

The public supplement groups the evidence by benchmark. The MathConstruct source-audit component contains the exact seven-case map, written clauses, parser-reachable witness descriptions, released logic, focused-repair descriptions, and all 700 cross-variation records with released, hardened, applicability, and official-retention fields.

The replay component contains the recorded evaluator outcomes and termination fields for all 280 scheduled observations, together with model, prompt, and environment identifiers needed to reproduce the analysis. A standard-library verifier reconstructs the reported counts from the released records without rerunning generation.

The DCP-Bench-Open component records the v0.2.2 release locks, required-key contracts, 899 controlled cases, frozen 30-problem sample, 154 clause-family classifications, diamond-free witnesses and controls, custom 215-row diagnostic, and minimal hardening patches. Its verifier checks the package manifest and reconstructs every paper-facing count without downloading or executing model outputs.

`ARTIFACT_README.md` is the stable entry point for all three benchmarks, repository-relative commands, digest inventories, and the anonymous-release exclusion policy.

D Artifact Map

`ARTIFACT_README.md` describes the MathConstruct, DCP-Bench-Open, and ComBench materials in `supplementary-material.zip`. The `mathconstruct/` namespace contains the preserved source-audit, cross-variation, replay and public aggregate records. The `dcp-bench-open-v0.2.2/` namespace contains the controlled completeness matrix, semantic-clause records, custom diagnostic, version locks, minimal patches, and license notices. The `combench/` directory contains the task records, tested rules, counterexample, and verification code for Appendix I. Each benchmark directory has its own verifier; digest inventories are stored within the MathConstruct and DCP directories and at the supplement root for ComBench. All three exclude private responses, reviewer identities, correspondence, and local paths.

The separate `historical-evidence-supplement.zip` supplies public historical MathConstruct conversations, source versions, aggregate records, independent verification scripts, and reporting records for Appendix H. The original supplement remains unchanged.

E FULL SEVEN-CASE EVIDENCE TABLE

Table 5 provides the full evidence record for the seven MathConstruct counterexamples.

Table 5: Complete evidence table for the seven established semantic false acceptances. Released and hardened verdicts are shown explicitly. "Edit size" counts construction-level leaf changes from the official construction.

Task	Explicit statement requirement	Minimal parser-valid witness	Missing or incorrect checker condition	Released checker	Separately implemented clause predicate	Hardened checker	Edit size
`bmo-shortlist-2015-n7`	Four different positive integers per quadruple.	Replace the first quadruple's fourth value by its first.	Sorts each quadruple and checks uniqueness only between complete quadruples.	ACCEPT	$\|q\| = 4$ and $\|\mathrm{set}(q)\| = 4$ for every q.	REJECT	1 repl.
`bxmo-2011-1`	$1 < m < n$.	Replace $(2, 8)$ by $(2, 2)$.	Bounds m and checks factor sets but permits equal components.	ACCEPT	$1 < m < n$ for every pair.	REJECT	1 repl.
`emc-2022-2`	Positive integer k and at least one d_i.	Replace witness $[1, 1]$ by $[1]$.	Empty product and universal check pass vacuously.	ACCEPT	Every witness has length at least 2.	REJECT	1 del.
`imo-shortlist-2000-c4`	No row or column contains k adjacent empty cells.	Move pawn $(0, 4) \to (1, 4)$, preserving count and every column. Row 0 gains nine zeros.	Row loop scans a stale final row, while columns are scanned correctly.	ACCEPT	No row or column string contains 0^k.	REJECT	2 repl. / 1 move
`imo-shortlist-2017-n6`	Positive rational numbers.	Replace $1/2$ by $-1/2$ in the first triple.	Checks integer direct and reciprocal sums while leaving sign unchecked.	ACCEPT	Every submitted value is > 0.	REJECT	1 repl.
`tot-2018-1`	Positive irreducible fractions.	Replace the sixth fraction by $-18143999/21772806$.	Checks irreducibility and denominator relations while leaving sign unchecked.	ACCEPT	Every fraction is > 0.	REJECT	1 repl.
`usamo-2005-1`	A permutation of 255 divisor indices.	Append a second copy of initial index 2.	Set equality discards duplicate multiplicity and omits the exact-length check.	ACCEPT	Length and set cardinality are 255 with required index set.	REJECT	1 ins.

F FROZEN NATURAL-OUTPUT REPLAY

F.1 FROZEN DESIGN AND TARGET EVENT

The schedule contains Qwen3-4B-4bit at revision `4dcb3d101c2a062e5c1d4bb173588c54ea6c4d25` and DeepSeek-R1-Distill-Llama-8B-4bit at revision `e23907a0aad41aa45845a938ef0fe7dfeaab3c05`. It crosses the two checkpoints with seven fixed MathConstruct tasks and 20 seeds per model–task cell, yielding $2 \times 7 \times 20 = 280$ scheduled observations. Seeds 1101–1120 repeat in fixed order in each cell. Prompts are user-only and benchmark-faithful.

Both checkpoints use temperature 0.6, top-p 0.95, min-p 0, and an 8,192-token output cap. Qwen uses top-k 20 with thinking enabled; DeepSeek uses top-k 0. Each record contains one raw response field. The official evaluator supplies that whole response to the MathConstruct parser. Extraction starts at the last literal `\boxed{` opening marker and scans forward to its brace-balanced closure; an unfinished final box fails extraction without fallback to an earlier complete box. Successful extraction is followed by task-specific parsing and released-checker invocation.

We count a target event when an output reaches the official parser, is accepted by the released checker, and violates the predefined requirement. Exact-response deduplication is used only as a secondary check.

F.2 RESULTS

All 280 scheduled outputs were generated and evaluated. The official parser reaches 35 outputs, and the released checker accepts 16. None of those 16 accepted outputs violates the replay's predefined target requirement, so the target-event count is 0/280. The run also contains 254 token-cap terminations and 26 end-of-sequence terminations. Table 6 reports the complete accounting.

Frozen replay field	Count
Scheduled, generated, and evaluated observations	280
Official parser-reachable responses	35
Official released-checker accepts	16
Preregistered target events	0
Evaluation failures	0
Exact-byte duplicates	0
Token-cap terminations	254
End-of-sequence terminations	26

Table 6: Frozen natural-output replay accounting. Counts use all 280 scheduled outputs. Accepted-output counts are intermediate flow fields, and the primary denominator remains the full schedule.

Figure 5 reports termination by checkpoint.

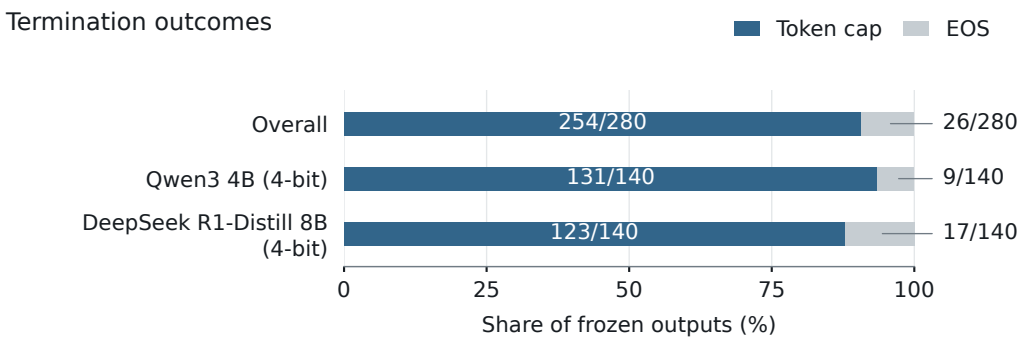


Figure 5: Termination outcomes in the frozen replay. Model-specific bars retain their 140-response denominators; the aggregate contains 254 token-cap and 26 end-of-sequence terminations.

F.3 Exploratory post hoc sensitivity analysis

A post hoc final-segment selector changes three classifications among the 16 whole-response accepts, whereas a last-complete-box selector changes none. Because these selectors were examined after the main replay definition, we report this only as a parsing sensitivity check.

G DCP-Bench-Open Validation Evidence

G.1 Locked release and evaluation contract

The audit uses DCP-Bench-Open v0.2.2 (Michailidis et al., 2026; 2025), commit `6b0094c6f080a34950ec2f52ee07fed8150327bd`. The release JSONL has SHA-256 `cc1914a45fb75fd819646250338fb75e4430b91ab2b17de20a84a3d25fa0b32e`. The released `eval.py` has SHA-256 `85148dee59e1cb7c4507e9a439a4cda0 fcdd6eae7ab397298ab12bbdcba0324b`. The locked release contains 164 discrete combinatorial satisfaction and optimization problems.

The evaluator runs a submitted constraint model, extracts the standard-output substring from the first `{` to the last `}`, and passes the entire substring to `json.loads`. It does not select the last valid JSON object from multiple objects. For a parsed object, it inserts equalities for the fields present and solves the released ground-truth model. Missing fields add no equality. For satisfaction problems, the remaining variables may be completed existentially by the solver. For optimization problems, acceptance also depends on the released objective comparison. The completeness audit isolates this binding behavior across the defined deletion families.

G.2 Controlled output completeness

For each problem, the audit derives a required key set from the machine-readable `decision_variables` record and checks it against the explicit output instruction. All 164 records have exact agreement between these sources. One accepted released-example object supplies the values for a valid control and for three deterministic deletion families. The empty-object family contains 164 cases. The leave-one-required-field-out family contains 338 cases, one per required field. The one-required-field-only family applies to the 59 problems with at least two required fields and contains 233 cases. Family construction is fixed before released or hardened evaluation.

The released evaluator accepts all 164 empty objects, all 338 leave-one-field-out objects, and all 233 one-field-only objects. Thus it accepts 735/735 controlled incomplete outputs, and every one of the 164 problems is affected by at least one omission. A pre-binding gate requiring the exact declared key set and recursively matching the example JSON types and shapes rejects all 735. Both versions accept 164/164 complete released-example controls. The exact-schema gate addresses the tested completeness failure mode, while the separate semantic audit examines clause enforcement.

G.3 Frozen semantic-clause audit

A separate sample was frozen before semantic outcomes. It contains 30 problems stratified by source, problem type, model size, and released-instance availability. Description review identifies 154 clause families. Mapping them to released model expressions classifies 149 as complete, one as absent, one as implemented differently, and three as unresolved. The semantic review identifies one confirmed omission in `csplib_050_diamond_free`. The task description requires vertex degrees to appear in nonincreasing order, but the released model does not encode that condition. We construct one full-schema witness for each released size $N \in \{10, 11, 12, 13, 14, 15\}$. Released evaluation accepts all six witnesses. Adding only the missing ordering constraint rejects all six.

The independent oracle checks Boolean square shape, symmetry, a zero diagonal, the diamond-free condition, positive degrees divisible by three, the degree-sum condition, and the target ordering independently of the released model. The added constraint is

$$\sum_j M_{i,j} \geq \sum_j M_{i+1,j} \qquad (0 \leq i < N-1).$$

Released and one-clause-hardened models accept all six intended-valid controls.

The remaining reviewed clauses either match the released model, are unsatisfied under the other released constraints, or could not be resolved unambiguously from the available task description. The handshaking-clause query is unsatisfiable under the remaining constraints. The three unresolved clauses concern all unique degree sequences in the diamond-free request, an optional perfect-square placement interpretation, and an unstated template-design objective. We therefore report only the confirmed ordering omission.

G.4 Custom all-released-instances diagnostic

The release includes 23 problems with multiple instance records, totaling 215 rows. The ordinary evaluator uses the released example/default configuration. The custom diagnostic performs the all-row extension by forming a one-row dataset for each released instance and applying the same released model output. ACCEPT versus NON-ACCEPT is stable across construction and independent verification: 48 rows accept and 167 do not. All 23 official example/default controls accept. The first released row accepts for 21/23 problems because five example preambles differ from the first dataset record. Fixed outputs accept on every released row for 2/23 problems and fail at least one row for 21/23. Raw REJECT versus ERROR subdivisions vary with environment and timeout, so the reported result uses the normalized split. These results characterize the custom diagnostic. Official benchmark accuracy remains tied to the ordinary evaluation path.

G.5 Artifacts and release matching

No exact public natural-output replay was identifiable for v0.2.2. Available stored generated-model metadata names `v1_verified` and lacks a byte-identical tuple of description, instance, schema, prompt, submitted program, and evaluator.

The public evidence projection is under `dcp-bench-open-v0.2.2/`. It contains the 899 controlled-case outcomes, 164 key contracts, frozen sample identifiers, 154 clause classifications, six witnesses and six controls, independent-oracle records, minimal hardening patches, the 215-row diagnostic, version locks, citations, Apache-2.0 notice, and a standard-library verifier. It excludes the internal construction archive, local paths, correspondence, and the upstream benchmark dataset. Readers retrieve the upstream release from its official repository or Zenodo record and can check its identity against the recorded hashes.

H Historical False Acceptance and Archive-Score Effects

H.1 Collection selection and historical binding

The historical audit covers three recovered MathConstruct collections with saved conversations, matching task instances, and corresponding aggregate records: `cot-parser-main`,

code-parser-main, and cot-rephrase. Within the seven affected task classes, the matched frames contain 1,068 records and 73 released accepts. We check those accepted artifacts against the relevant written requirements and verify five false accepts.

The three frames contain 480, 252, and 336 affected records, with 41, 19, and 13 released accepts, respectively. The remaining 68 accepted artifacts do not violate the audited requirements. Raw-only records outside the corresponding aggregates comprise 372 base, 248 code-parser, and 40 rephrase records.

The official log archive has SHA-256 `0350e229eedc1c7e1fe322eed4ba2a7b30c14e528962e5beedf7b255efad9096`. Historical reproduction uses commit `9f6f5874d35cecb4562bf458a1125a206957c8b0`. Each artifact was located again in that archive and matched to exactly one aggregate row by collection, model, task, and parameters. Original conversations, record-specific rephrased statements, response hashes, and selected-artifact hashes were checked against the discovery copies. All five artifacts are distinct; their counts represent two omissions, not five defects.

The saved-conversation path chooses the last parseable eligible message in reverse order, then applies the historical parser and checker. It selects message 2 in the three-message USAMO record and message 0 in each one-message EMC record, using zero-based indices. This conversation selection differs from the single-response local replay. Saved generated programs were not executed. Aggregate-reference and selection comparisons found no mismatch in the reconstruction.

H.2 Five mathematical certificates

Record identifiers below resolve to complete saved artifacts and per-row arithmetic in the historical-evidence supplement. Mathematical verification uses independently written standard-library logic without importing the discovery predicates or patched checkers.

USAMO, `4a7d44d6db0459ad98d13465`. The code-parser-main/GPT-4o-mini record, raw index 3, has parameters $(a, b, c, d) = (72, 30, 52, 43)$ and $n = 4{,}829{,}760$. Exact divisor enumeration gives 224 divisors including 1, requiring the 223 indices $1, \ldots, 223$. The artifact supplies 224 indices, with index 1 duplicated and no required index missing. It is not a permutation of the required domain. Expanding the supplied circular sequence gives zero coprime adjacencies: circular adjacency passes, and exact multiplicity is the violation. The historical checker accepts; both repaired checker implementations reject by length. This is the source-audit USAMO omission in a recovered code-parser output.

EMC, `3b7869695169bc4c2374ed13`. The cot-parser-main/Flash-Thinking record, raw index 1, requests $n = 11$ distinct lovely numbers. Its eleven unique witness sequences have only three first fields: 1 occurs six times, 2 three times, and 6 twice. Every sequence satisfies the arithmetic conditions below. Distinct sequences therefore fail to provide eleven distinct lovely numbers.

EMC, `4d332cad5a324d0dcc5d1d68`. The cot-rephrase/QwQ record, raw index 0, explicitly requests ten distinct lovely numbers. Its ten unique sequences have four first fields: 1 and 2 occur three times each; 6 and 42 twice each. All row arithmetic passes.

EMC, `684cb359a95fdbb7a9e4921f`. The cot-rephrase/o1-mini record, raw index 1, explicitly requests ten distinct lovely numbers. Ten unique sequences contain five first fields; 1806 appears six times with different witness sequences. All row arithmetic passes.

EMC, `9d3c6de8dff46cd3aac8ed0f`. The cot-rephrase/Flash-Thinking record, raw index 1, explicitly requests eleven distinct lovely numbers. Eleven unique sequences contain five first fields: 1 occurs three times, and 2, 6, 42, and 1806 twice each. All row arithmetic passes.

For each EMC sequence $[\ell, d_1, \ldots, d_k]$, independent integer arithmetic checks positive entries, $k \geq 1$, $\prod_i d_i = \ell$, and $d_i^2 \mid \ell + d_i$ for every i. Every row in all four records passes. The exact base and rephrased statements explicitly require distinct lovely numbers, and the first field denotes that number; no ambiguity about sequence versus number distinctness was found. The four EMC

artifacts are still accepted after the original seven source-audit fixes, because the distinct-number requirement is a separate omission. This omission is separate from the original nonempty-witness gap and was not part of the 127-violation experiment.

H.3 Task-weighted aggregate corrections and paper linkage

The historical analysis computes each task's accuracy across its variations and averages across tasks. We reproduce the complete original aggregate before replacing only the verified erroneous verdict in each affected model/collection pair. Each changed task has four variations, so its accuracy decreases by $1/4$. The main/code pairs average 126 tasks and 475 records; each rephrase pair averages 127 tasks and 480 records. Thus the reductions are $100/(4 \cdot 126) = 0.1984127$ and $100/(4 \cdot 127) = 0.1968504$ percentage points. They are not relative percentage reductions.

Table 7 gives the five reconstructed archive-score corrections.

Collection	Model	Before (%)	After (%)	Original paper surface
cot-parser-main	Flash-Thinking	27.0503	26.8519	Table 2: printed 27.05% matches
code-parser-main	GPT-4o-mini	6.9444	6.7460	Absent from Figure 4
cot-rephrase	QwQ	12.7953	12.5984	Figure 6 point identity unresolved
cot-rephrase	o1-mini	20.4724	20.2756	Figure 6 point identity unresolved
cot-rephrase	Flash-Thinking	28.5433	28.3465	Figure 6 point identity unresolved

Table 7: Verified task-weighted archive aggregates after replacing only the five confirmed false-accept verdicts. Surface references in the final column refer to the original MathConstruct paper, not this manuscript.

The corresponding accepted-record counts are $120 \to 119$ and $28 \to 27$ of 475 for the two main/code pairs, and $53 \to 52$, $88 \to 87$, and $128 \to 127$ of 480 for the rephrase pairs. The sums of task accuracies change from $34 + 1/12$ to $33 + 5/6$, 8.75 to 8.50, 16.25 to 16, 26 to 25.75, and 36.25 to 36. The mean uses task weights rather than the targeted accepted subset.

The archive model keys are `gemini-2.0-flash-thinking-exp` (Flash-Thinking), `Qwen__QwQ-32B-Preview` (QwQ), `gpt-4o-mini`, and `o1-mini`. Only the Flash-Thinking main baseline is directly matched to a printed numerical result. The original paper's Figure 4 shows GPT-4o rather than GPT-4o-mini. Figure 6 maps to the rephrase collection, but its points are unlabeled and plotting source is absent from the authenticated snapshot, leaving exact point identity unresolved. All five archive corrections verify; effects on rankings and published conclusions remain unestablished.

H.4 Repairs and regression checks

The USAMO correction adds an exact-length check before the existing set and adjacency checks. The EMC correction requires the first fields of the submitted witness sequences to be distinct. Official constructions remain accepted under these checks, and the targeted historical false accepts are rejected. The earlier 480-record rerun under the original seven source-audit corrections produces no verdict changes.

H.5 Environment

Checker reproduction used the preserved Python 3.12.14 runtime and hash-matched source trees. The frozen dependency specification was not installable as written because `annotated-types==0.8.0` was unavailable from the configured package index; the reproduced checker results therefore rely on the preserved runtime. Independent mathematical certificates and aggregate recomputation use only the Python standard library. No model-generated program was executed during verification.

I COMBENCH EXTENSION: THE SAME TEAM, WRITTEN A DIFFERENT WAY, CAN FLIP THE VERDICT

I.1 SETUP

We ran a small extension on 12 selected ComBench construction tasks (Zhang et al., 2026). All 12 reference constructions were accepted by the released construction evaluator. We then tested 41 targeted mutations derived from those accepted constructions. Forty were rejected. One was accepted.

I.2 THE ACCEPTED RMO-2019-P15 RULE

RMO-2019-P15 asks for a rule that marks exactly 1 or 3 teams as "cooperating" in any valid partition of 24 students into four teams of six. The accepted candidate starts from a correct parity rule: a team cooperates if it contains an odd number of students in $\{1, 2, 3\}$.

The candidate adds `team[0] < team[-1]`. The evaluator's generic checks use ascending team lists, so this condition never changes their verdicts. The candidate's supplied examples are also ascending. Yet the task defines a team as a list of six distinct students and requires correctness for any valid partition; its example checks preserve member order without sorting.

Reversing each of the same four valid teams makes every endpoint comparison false. Figure 6 shows a partition whose cooperating-team count drops from 1 to 0. Zero violates the required one-or-three condition, although the released evaluator accepts the rule.

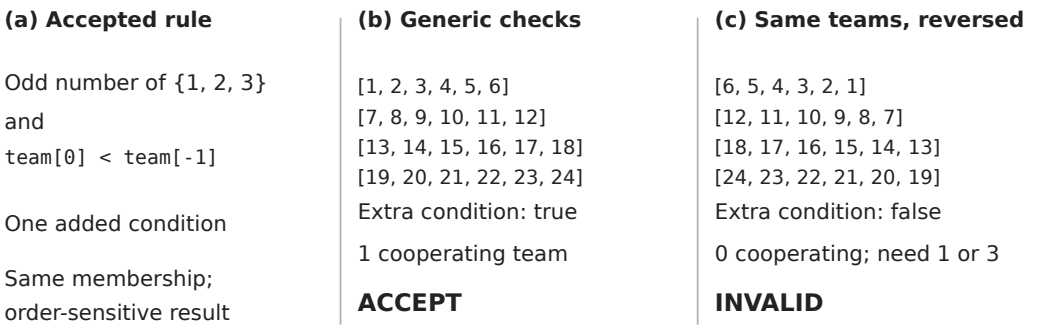


Figure 6: ComBench representation-coverage failure. The released evaluator accepts a rule that adds an order-sensitive condition to an otherwise correct team test. Its generic checks call the rule on ascending team lists, so the extra condition never changes the verdict there. Write the same valid teams in reverse order, and the accepted rule now produces zero cooperating teams, violating the required one-or-three condition.

I.3 VERIFICATION AND REPAIR

Replaying the saved response reproduced acceptance. Two valid controls, the original reference rule and a valid relabeling rule, also pass. A reversal-consistency check rejects the bad rule and retains both controls. This targeted repair compares the rule's output before and after reversing each team list.

I.4 REPRODUCTION EVIDENCE

The supplement's `combench/` directory contains the selected-task list, mutation outcomes, saved responses, evaluator, counterexample, repair, and independently implemented confirmation checks. Its README gives record-reconciliation and isolated-replay commands.