**Proceedings of the ASME 2026**
**International Design Engineering Technical Conferences &**
**Computers and Information in Engineering Conference**
**IDETC-CIE 2026**
**August 23–26, 2026, Houston, TX, USA**

**DETC2026-XXXX**

# RELIABLE NATURAL-LANGUAGE TO SYSMLV2 TRANSLATION VIA VALIDATOR-DRIVEN ITERATIVE REFINEMENT

**Chance LaVoie[1,*], Eladio Andujar Lugo[1], Levent Burak Kara[1]**

[1]Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA

## ABSTRACT

**Keywords:** SysMLv2, model-based systems engineering, large language models, validator-in-the-loop, natural language to model translation, syntactic validity

## 1. INTRODUCTION

Model-Based Systems Engineering (MBSE) replaces document-centric workflows with model-centric engineering, where formal artifacts carry requirements, structure, behavior, and verification intent across the lifecycle [1–4]. By making the model the primary technical source, MBSE reduces cross-document reconciliation overhead and improves traceability under change [2, 5].

SysMLv2 strengthens this transition by standardizing a formal language with complementary textual and graphical representations and machine-readable artifacts under OMG [6–8]. In practice, the same model can be authored as text, rendered visually, and exchanged across tools with less ambiguity. Because SysMLv2 is formal and textual, model construction is scriptable.

This scriptability, combined with the recent rise of LLMs in engineering practice, creates a concrete opportunity for automated SysMLv2 modeling workflows. LLM-based pipelines can generate candidate models directly from natural-language inputs.

Software engineering provides encouraging precedent: controlled and field studies report substantial productivity gains from LLM-assisted generation, including 55.8% faster task completion, 26.08% higher weekly completed tasks in enterprise randomized deployments, and measurable increases in pull-request throughput [9–11]. These results suggest that integrating generation into everyday tool use can materially accelerate engineering iteration.

The analogous opportunity in MBSE is to automate low-value model authoring steps while increasing modeling velocity. If natural-language prompts can be converted into *production-validator–accepted* SysMLv2, teams can spend less effort on syntax correction and more effort on architecture and verification reasoning.

Motivated by this potential, recent SysMLv2 generation systems have moved beyond unreliable one-shot prompting toward structured generation workflows [12, 13]. SysTemp uses a template-first multi-agent design centered on a *TemplateGeneratorAgent* and a *ParserAgent*. A Jinja2-based template tool first builds a structured SysMLv2 skeleton, then a writer agent fills in content while parser feedback guides iterative correction [12]. By contrast, the 2025 *Computers in Industry* framework organizes generation as an agentic loop with a RAG-based context engine and a validation engine, with ANTLR-based grammar validation as the syntax gate [13]. In that framework, the context engine retrieves semantically similar natural-language prompt and SysML file pairs and exposes them to the LLM as context [13]. Both approaches rely on context-free grammar (CFG) parsing as the primary basis for syntactic validation and as the primary acceptance metric.

In parallel, SysMBench has established a common benchmark for evaluating natural-language–to–SysML generation quality, especially on semantic alignment criteria [14]. Read together, the current state of the art suggests that generating SysMLv2 from natural language that passes grammar-parser checks is feasible.

Grammar parsability guarantees structural conformance to SysMLv2 context-free syntax: valid token ordering, balanced delimiters, and production-rule-compliant declarations. It does *not* guarantee production validation acceptance, because industrial validators enforce additional model-wide constraints such as name resolution, type consistency, ownership/composition rules, multiplicity constraints, and cross-reference integrity. Therefore, parser-passing outputs provide no assurance that they will be accepted by a production modeling environment for downstream use, including visualization, semantic validation, simulation, and optimization. Without acceptance under a full industrial validator, such models remain unusable in practice. In an auxiliary repository demonstration, we show ten distinct cases that pass SysML ANTLR parsing but fail production validation, empiri-

*Corresponding author: chancel@cmu.edu
Documentation for `asmeconf.cls`: Version 1.45, February 22, 2026.

cally reinforcing this distinction [15]. The critical gap is reliable generation of *production-validator–accepted* SysMLv2 from arbitrary natural-language prompts.

To close this gap, we instantiate a generate–verify–refine workflow in which candidate SysMLv2 models are iteratively evaluated against a production validation backend and revised until acceptance. This paradigm is not novel in itself. It draws from established traditions in formal methods and inductive synthesis, where candidate artifacts are proposed, evaluated by an external oracle, and corrected using deterministic feedback from that oracle [16–19]. Our contribution is to operationalize this verification-driven pattern for natural-language–to–SysMLv2 generation under industry-facing acceptance criteria.

Related work in code generation has already demonstrated the technical value of iterative verifier-in-the-loop refinement. Wang et al. show that deterministic compiler diagnostics can be used as structured correction signals in multi-stage neural code generation, improving acceptance of generated programs under executable checks [20]. Grubišić et al. similarly use compiler feedback to iteratively refine LLM outputs, treating validator responses as an explicit control signal for subsequent generations [21]. Taken together, these works establish the core mechanism we rely on: generate a candidate, evaluate it with a deterministic backend, feed diagnostics back into the next iteration, and repeat until the validator reports no remaining errors.

We apply this principle directly to MBSE. Our framework places a production SysMLv2 syntax validator, SysIDE [22], in the loop as a deterministic oracle, and generation proceeds until zero-error acceptance is achieved under that validator. At convergence, the resulting model is production-ready and operational within the modeling environment—loadable, renderable, analyzable, and suitable for downstream verification workflows. By elevating production-validator acceptance to the convergence criterion, SysMLv2 generation moves from syntactic plausibility to deployable modeling infrastructure, establishing a practical foundation for Copilot-like support in MBSE workflows.

## 2. RELATED WORK — NEEDS A LOT OF WORK – CHANCES LOWEST PRIORITY

Our review focuses on prior efforts in automated SysML v2 generation, grammar-constrained structured synthesis, and validator-guided iterative refinement, with an emphasis on approaches addressing reliable syntactic correctness in low-data modeling languages.

### 2.1. LLM-Based SysML v2 Model Generation

Recent work has explored using large language models to generate SysML v2 models from natural language. Bouamra et al. [12] propose SysTemp, a multi-agent framework that decomposes generation into requirement extraction, template-based skeleton construction, and grammar-level parsing feedback. Two specialist agents structure the flow: a *TemplateGeneratorAgent* uses a Jinja2-backed template tool to construct a syntactically organized SysMLv2 skeleton, and a *ParserAgent* validates the generated text against formal SysMLv2 grammar and returns diagnostics to a writer agent for iterative repair [12]. This work demonstrates that structural scaffolding and agent decomposition improve grammar conformity in sparse-data settings, with syntactic acceptance anchored to context-free grammar (CFG) parsing. However, syntactic correctness is defined at the grammar level and convergence is reported empirically rather than enforced as a termination invariant.

Cibrián et al. [13] introduce a distinct agentic architecture in *Computers in Industry*: generation is orchestrated in an agentic loop with a RAG-based context engine and a validation engine (for grammar-level checking), using ANTLR-generated CFG parsing as the syntactic acceptance mechanism. Their context engine retrieves semantically similar natural-language and SysML exemplar pairs to condition generation [13]. Their approach reports 100% syntactic validity across 20 curated prompts under grammar-level parsing. While this represents a significant advance in structured generation reliability, correctness is enforced through grammar parsing rather than full production validation. Because grammar-level validation ensures context-free structural conformity but does not enforce full static semantics under a production modeling environment, grammar-valid models may still fail production validation in practice. Together, these efforts establish that iterative validation substantially improves syntactic success in SysML v2 generation. However, prior systems define correctness primarily at the grammar level and evaluate on limited scenario sets. For industry use, the unresolved gap is generalizable *production-validator–accepted* syntactic correctness across diverse prompts. An additional gap is retrieval dependence: pipelines that require semantically similar NL-SysML exemplars may be less generalizable when comparable examples are sparse or unavailable. Our work builds on these insights while shifting the correctness oracle from grammar parsing to a production SysMLv2 validator and scaling evaluation to benchmark-level scenarios.

### 2.2. Grammar-Constrained and Template-Based Structured Synthesis

Structured synthesis approaches aim to reduce hallucinations in LLM output by constraining generation via templates or grammar rules. SysTemp [12] explicitly uses a template generator based on Jinja2 to construct syntactically compliant model skeletons prior to completion. Grammar-constrained decoding and post-generation parsing similarly reduce token-level structural invalidity.

While grammar validation ensures adherence to context-free rules, it does not enforce type resolution, cross-reference integrity, constraint satisfaction, or production modeling environment compatibility. Thus, grammar-valid artifacts may remain unusable within industrial MBSE workflows. Concretely, a model such as `port p: UndefinedType;` can be grammar-parsable yet fail production validation because the referenced type is unresolved; conversely, a missing delimiter (e.g., omitted semicolon) fails parsing before production validation semantics are even checked. Our approach differs by treating grammar conformity as necessary but insufficient and requiring full production validation acceptance prior to termination.

### 2.3. Validator-in-the-Loop and Verifier-Guided Generation

Prior work in neural code generation has explored leveraging compiler diagnostics to improve the compilability of model outputs. For example, Wang et al. [20] propose a multi-stage refinement framework that uses compiler feedback to iteratively revise generated programs and increase compilation success rates. Such approaches demonstrate that deterministic compiler signals can serve as effective supervisory feedback in programming-language settings. However, these systems operate in mature programming ecosystems and treat compilation success as an empirical objective rather than as a structural termination invariant.

Beyond compiler-feedback approaches in code generation, iterative generation guided by deterministic verifiers has also been studied under counterexample-guided inductive synthesis and execution-based refinement paradigms. Grubišić et al. [21] demonstrate that LLVM compiler feedback can serve as a supervisory signal for refining LLM-generated intermediate representation. Their work shows that deterministic compilation signals constrain output space and improve structural validity in programming languages.

While these approaches establish the feasibility of compiler-aware refinement in software domains, they differ in both objective and context from model-based systems engineering. Compiler feedback in such systems is typically used to improve optimization quality or increase compilation probability, rather than to enforce strict termination conditions. Moreover, these methods operate in programming languages with extensive training corpora and stable ecosystems.

In contrast, our work applies validator-in-the-loop refinement to SysMLv2, a newly standardized modeling language with sparse representation in LLM training data and strict production-validation requirements. Rather than treating validation results as heuristic feedback, we enforce zero-error acceptance under a production SysMLv2 validator (invoked via `syside check`) as a termination invariant. This elevates syntactic correctness from an empirical metric to a property guaranteed by construction, aligning generation reliability with production MBSE modeling-environment criteria rather than grammar-level approximations.

## 3. METHODOLOGY

### 3.1. Study Objective and Paired Design

This study evaluates one question: for the same prompt and the same model, does validator-in-the-loop refinement increase production validation acceptance relative to single-shot generation? The scope is strictly syntactic.

The experimental unit is one prompt–model pair. For each unit, we evaluate two paired conditions taken from the same run trajectory:

1. **Baseline (single-shot):** production validation outcome for the initial candidate ($k = 0$).
2. **Pipeline (iterative):** production validation outcome at the final available iteration after iterative repair.

Because both outcomes are taken from the same prompt–model run, this design isolates the effect of iterative validator feedback while holding prompt content and model identity fixed.

### 3.2. Validator-in-the-Loop Generation Procedure

Our controller follows a generate–validate–repair workflow for natural-language–to–SysMLv2 generation. At each iteration, the model proposes a complete SysMLv2 candidate, the production validator returns deterministic diagnostics, and the next model call is conditioned on those diagnostics.

Let $P$ denote the natural-language prompt, $M_t$ the generated candidate at iteration $t$, and $V(\cdot)$ the production validator. The update is

$$M_{t+1} = f(P, M_t, V(M_t)),$$

where $f(\cdot)$ is the model revision operator conditioned on validator feedback.

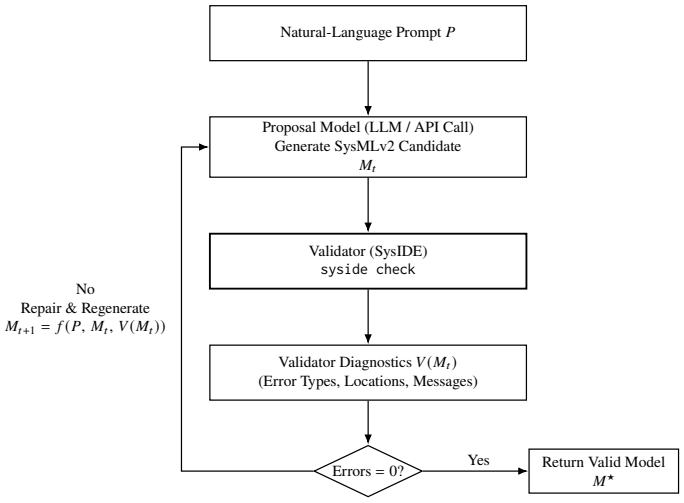


**FIGURE 1: Validator-in-the-loop generate–validate–repair workflow. The prompt is fixed per case, and each revision is driven by deterministic validator diagnostics.**

The production validator oracle is SysIDE validation (`syside check`) [22]. A run is successful and terminates only when zero validation errors are reported. This oracle choice is additionally supported by an auxiliary ten-case demonstration in the repository showing ANTLR parser pass with production-validation failure, i.e., grammar conformity without operational acceptability [15].

### 3.3. Dataset and Outcome Extraction

SysMBench provides paired natural-language prompts and ground-truth SysMLv2 models for benchmark evaluation [14]. In this study, we use the curated natural-language prompt set (IDs 1–151) as generation inputs because it was designed to stress SysMLv2 LLM generation across diverse modeling patterns.

To assess model-agnostic behavior of the same controller, we run four model configurations: OpenAI Codex 5.2 (`gpt-5.2-codex`) [23], Anthropic Sonnet 4.6 (`claude-sonnet-4-6`) [24], DeepSeek Reasoner (`deepseek-reasoner`) [25], and Mistral Large (`mistral-large-latest`) [26]. This yields 604 prompt-level cases (151 prompts × 4 models).

From saved run records, we extract single-shot and pipeline pass/fail outcomes, iterations run, iterations to success, first/final error counts, cumulative error counts, per-iteration runtime, and token usage. For grammar-versus-production analysis, we also

extract single-shot ANTLR parse pass/fail and single-shot SysIDE pass/fail from the same initial candidate artifacts for all 604 prompt–model cases. Error families are grouped from validator diagnostics (for example, parsing and reference errors), while warnings are tracked separately and do not change pass/fail labels.

### 3.4. Evaluation Metrics

**3.4.1. Production Validation Acceptance** The primary evaluation metrics are production validation acceptance rates. For each analysis slice (overall and per-model), we report:

1. single-shot pass rate (initial candidate, $k = 0$),
2. pipeline pass rate (final iteration of the validator-gated loop).

Secondary diagnostics include iterations-to-success, first-iteration and cumulative error counts, and validator error-family frequencies.

**3.4.2. Grammar vs Production Validity** To quantify the operational gap between grammar-level validity and production-validator acceptance, we compare ANTLR and SysIDE outcomes on the same single-shot artifact from each case. This comparison is performed on the initial candidate ($k = 0$ in repair-cycle indexing).

Grammar validity is defined as ANTLR parse success using the ANTLR-based SysML parser integrated in the repository. Production validity is defined as SysIDE acceptance (zero SysIDE errors) under `syside check`. For each case, we record binary pass/fail outcomes for both checks and construct a $2 \times 2$ contingency table (ANTLR pass/fail $\times$ SysIDE pass/fail).

**3.4.3. Convergence Metrics** To characterize how quickly the loop approaches acceptance, we index progress by repair cycles. Let $k = 0$ denote the initial single-shot generation (no validator feedback), and let $k \geq 1$ denote $k$ rounds of generate–validate–repair. Let $A_k$ denote cumulative production-validation acceptance after repair cycle $k$.

We define residual failure mass as

$$R_k = 1 - A_k,$$

where $R_k$ is the fraction of prompt–model cases not yet accepted at repair cycle $k$. We then define an empirical contraction ratio

$$\rho_k = \frac{R_{k+1}}{R_k}.$$

When $0 < \rho_k < 1$, residual failure shrinks from one cycle to the next; when $\rho_k$ is approximately stable across $k$, the observed trajectory suggests contraction-like behavior, with approximately multiplicative reduction in the early repair regime under standard contraction-style analyses in iterative methods [27–30]. In finite empirical campaigns, we estimate contraction behavior from early-to-mid cycles, before residual mass becomes very small; terminal transitions are excluded from rate estimation because finite-sample tail effects can produce unstable ratios near the finite-sample resolution. This characterization is descriptive of observed finite-sample behavior and does not assert formal convergence guarantees.

We also report time-to-threshold acceptance

$$T_\varepsilon = \min\{k : A_k \geq 1 - \varepsilon\},$$

with explicit reporting of $T_{90}$, $T_{95}$, and $T_{99}$. This follows standard iteration-to-threshold (iteration-to-$\varepsilon$) complexity summaries in optimization [31]. These metrics provide an interpretable rate summary complementary to pass-rate metrics.

This framing follows empirical convergence-rate characterizations used in iterative optimization and search, including multiplicative (geometric-style) interpretations of residual reduction [30], while remaining descriptive rather than proving formal convergence guarantees. It is also consistent with iterative verifier-feedback refinement in code generation, where deterministic diagnostics guide successive corrections toward acceptance [20, 21].

### 3.5. Statistical Reliability Analysis

For convergence reliability, we model each prompt–model case as a Bernoulli success if it reaches eventual production-validator acceptance (zero SysIDE errors) within the allowed loop. Let $p$ denote convergence probability for prompts drawn from the SysMBench-style benchmark distribution under the same controller, validator, and backend configuration. Because the validator-gated loop is monotonic—accepted cases terminate immediately—regression transitions (single-shot pass $\rightarrow$ pipeline fail) are structurally excluded. Improvement is therefore quantified descriptively rather than via symmetric paired hypothesis tests.

We report an exact 95% Clopper–Pearson lower confidence bound for $p$ [32]; in the all-success case this is $p_L = (\alpha/2)^{1/n}$ with $\alpha = 0.05$. We also report the complementary one-sided 95% binomial upper confidence bound on failure probability $q = 1 - p$. In the zero-failure case, this bound is obtained by solving $(1 - q)^n = 0.05$, yielding $q_U = 1 - 0.05^{1/n}$. For large $n$, this expression is well approximated by $q_U \approx -\ln(0.05)/n \approx 3/n$.

These reliability bounds are intentionally scoped to SysMBench-style prompt distributions under the evaluated configuration and are reported numerically in Results. They are not treated as universal guarantees over arbitrary natural-language inputs.

### 3.6. Reproducibility

All code, run artifacts, and analysis outputs used in this study are stored in the project repository [33]. The repository includes the scripts required to regenerate campaign statistics, tables, and figures.

## 4. RESULTS

### 4.1. Primary Outcome: Production Validation Acceptance

Across all 604 prompt-level trials (151 prompts for each of four models), single-shot production validation succeeded for 309/604 cases (51.16%). Under validator-in-the-loop refinement, pipeline production validation succeeded for 604/604 cases (100.00%).

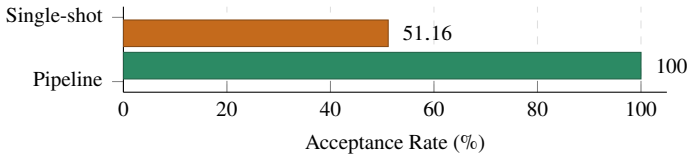


**FIGURE 2: Overall single-shot vs final pipeline production-validation acceptance across all 604 prompt-level cases.**

### 4.2. Per-Model Reliability

All models reached 151/151 eventual production validation under the validator-gated loop, but single-shot pass rates differed substantially. Figure 3 provides the per-model comparison with exact acceptance values annotated on the bars.

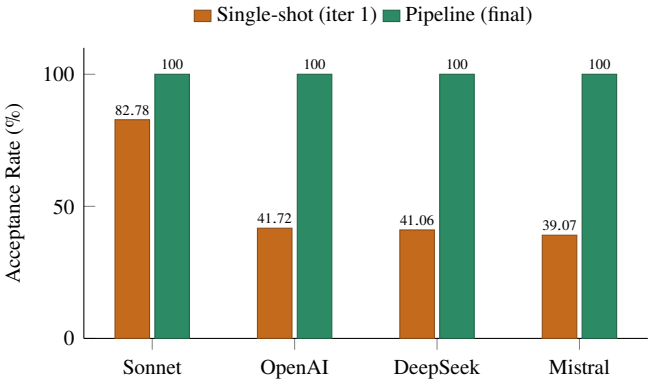


**FIGURE 3: Per-model single-shot vs final pipeline production-validation acceptance.**

Anthropic Sonnet 4.6 had the highest single-shot pass rate (82.78%), while OpenAI (41.72%), DeepSeek Reasoner (41.06%), and Mistral Large (39.07%) showed worse single-shot behavior and larger single-shot-to-pipeline gaps.

### 4.3. Convergence Behavior

Convergence is indexed by repair cycles: $k = 0$ denotes the initial single-shot generation (no validator feedback), and $k \geq 1$ denotes validator-guided repair cycles; acceptance corresponds to zero-error output under the production validator.

Figure 4 first presents the pooled convergence trajectory across all prompt-level cases.

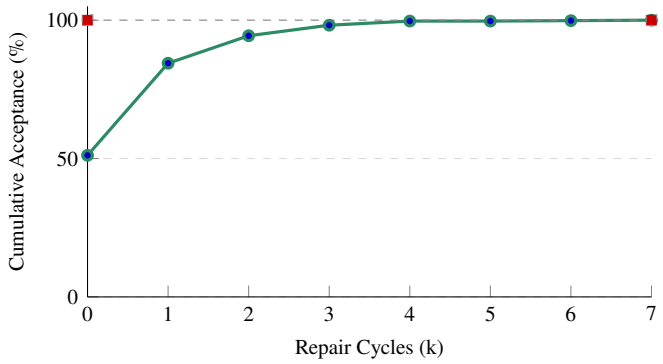


**FIGURE 4: Cumulative production-validation acceptance versus repair cycles ($k$). $k = 0$ denotes initial single-shot generation; $k \geq 1$ denotes validator-guided repair cycles. Acceptance corresponds to zero-error output under the production validator.**

The pooled curve shows a large first-step jump from $k = 0$ to $k = 1$, followed by rapid compression by $k = 2$ and then a short tail. Table 1 provides the exact counts and percentages underlying Figure 4: acceptance increases from 51.16% at $k = 0$ to 84.44% at $k = 1$, reaches 94.37% by $k = 2$, and reaches 99.67% by $k = 4$. Only two cases remain in the grouped $k = 5$–8 tail (0.33%), where cumulative acceptance reaches 100.00%. Consistent with this front-loaded pattern, iterations-to-acceptance are summarized by mean 1.727, median 1, IQR 1–2, and maximum 8.

**TABLE 1: Distribution of repair cycles to first production-validation acceptance (pooled across 604 prompt-level cases).**

| Repair cycles ($k$) | Cases | Share | Cumulative |
|---|---|---|---|
| 0 | 309 | 51.16% | 51.16% |
| 1 | 201 | 33.28% | 84.44% |
| 2 | 60 | 9.93% | 94.37% |
| 3 | 23 | 3.81% | 98.18% |
| 4 | 9 | 1.49% | 99.67% |
| 5–8 | 2 | 0.33% | 100.00% |

**Rate Characterization.** To quantify the observed convergence shape, we report an empirical contraction analysis of residual failure mass. Under contraction-style iteration analysis, let residual failure mass be $R_k = 1 - A_k$. The observed residual sequence is $R_0 = 0.4884$, $R_1 = 0.1556$, $R_2 = 0.0563$, $R_3 = 0.0182$, and $R_4 = 0.0033$. Using early cycles, the empirical contraction ratios are $\rho_0 \approx 0.1556/0.4884 \approx 0.32$, $\rho_1 \approx 0.0563/0.1556 \approx 0.36$, and $\rho_2 \approx 0.0182/0.0563 \approx 0.32$. These values cluster around an average early-cycle contraction factor of approximately 0.33 (computed as the arithmetic mean of $\rho_0$–$\rho_2$). Tail transitions are excluded from contraction-rate estimation because ratios become unstable when residual mass is near the finite-sample resolution. These early-cycle ratios cluster in a narrow band, suggesting an approximately multiplicative ("contraction-like") reduction pattern in the initial repair regime. In practical terms, early cycles remove roughly two-thirds of the remaining failures per cycle (in the observed early regime).

Complementing the contraction estimate, time-to-threshold metrics quantify convergence speed. The observed thresholds are $T_{90} = 2$, $T_{95} = 3$, and $T_{99} = 4$. These values characterize how

rapidly validator-guided refinement approaches the acceptance set in repair-cycle space.

To assess whether this pooled behavior is shared across backends, Figure 5 overlays per-model cumulative acceptance trajectories.

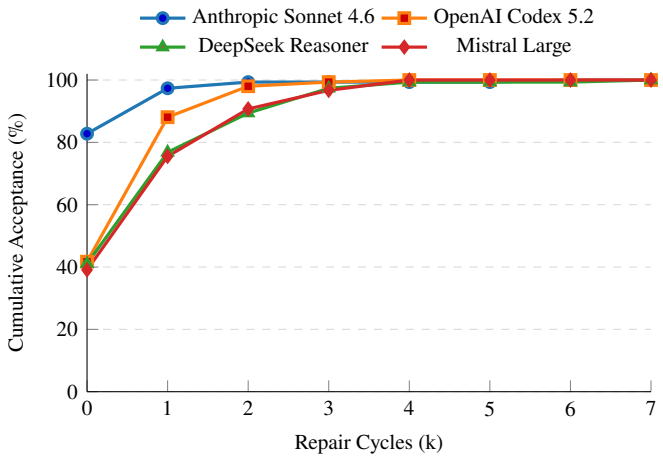


**FIGURE 5: Per-model cumulative production-validation acceptance versus repair cycles ($k$).**

Figure 5 shows that single-shot starting points differ across models, but the trajectories compress rapidly once validator-guided repair begins. Despite different initial acceptance levels, all curves move quickly toward full acceptance within a small number of repair cycles. This pattern supports the model-agnostic control-signal interpretation: deterministic validator diagnostics drive the dominant convergence dynamics across backends.

### 4.4. Statistical Reliability

This subsection reports realized values for the convergence-reliability analysis defined in Section 3.6. Across the full campaign, $n = 604$ prompt–model cases were evaluated, and eventual production-validation acceptance was observed in all cases ($x = 604$, failures $= 0$). Single-shot pass rate was 51.16% (309/604), while pipeline pass rate was 100.00% (604/604).

Under the Bernoulli convergence model, the exact 95% Clopper–Pearson lower confidence bound for convergence probability $p$ in the all-success case is $p_L = (\alpha/2)^{1/n}$ with $\alpha = 0.05$ and $n = 604$, giving $p_L = (0.025)^{1/604} \approx 0.9939$ (99.39%) [32]. Therefore, with 95% confidence, convergence probability is at least 99.39% for SysMBench-style prompts under the evaluated controller, validator, and backend configuration.

For failure probability $q = 1 - p$, the complementary one-sided 95% binomial upper bound is obtained from $(1 - q)^n = 0.05$. Substituting $n = 604$ gives $q_U = 1 - 0.05^{1/604} \approx 0.00495$ (0.495%). The large-$n$ approximation gives $q_U \approx -\ln(0.05)/604 \approx 0.00496 \approx 3/604$.

These reliability bounds are intentionally scoped to SysMBench-style prompt distributions under the evaluated configuration and do not claim universal convergence over arbitrary natural-language inputs. Because the validator-gated loop is monotonic (accepted cases terminate immediately) and repairs are guided by deterministic diagnostics, the observed reliability is consistent with a structural property of validator-in-the-loop control on this benchmark distribution.

### 4.5. Grammar Validity vs Production Validity and Error Burden

Beyond aggregate reliability, practical MBSE usability requires production validation acceptance in a modeling environment, not only grammar parsability. To quantify this operational gap, we compared first-iteration ANTLR parsing outcomes against first-iteration production validation outcomes for all 604 prompt–model cases.

**TABLE 2: First-iteration contingency between grammar parsing (ANTLR) and production validation acceptance (SysIDE), $n = 604$.**

| | SysIDE pass | SysIDE fail | Total |
|---|---|---|---|
| ANTLR pass | 309 | 60 | 369 |
| ANTLR fail | 0 | 235 | 235 |
| Total | 309 | 295 | 604 |

The first-iteration grammar pass rate was 369/604 (61.09%), while the first-iteration production pass rate was 309/604 (51.16%). Grammar-only passes occurred in 60 cases. This corresponds to 60/369 = 16.26% of grammar-valid outputs and 60/604 = 9.93% of all outputs. Grammar-level parsing overestimates production usability by 16.26% among parseable artifacts. No cases were observed with production pass and grammar fail (0/604). Thus, grammar validity is strictly necessary but not sufficient for production-validation acceptance under the evaluated configuration.

First-iteration validator diagnostics totaled 2,209 errors, giving a mean of 2,209/604 = 3.66 errors per generated model, indicating that first-iteration outputs frequently contain multiple structural defects rather than isolated formatting errors. Restricting to failing first-iteration cases (295), the mean burden was 2,209/295 = 7.49 errors per failing model. Error burden was concentrated in `parsing-error` (1,144; 51.79%) and `reference-error` (824; 37.30%), which together account for 89.09% of first-iteration errors. These results indicate that validator-gated repair is addressing multi-error structural artifacts and that grammar-level parsing alone would have admitted non-usable models into downstream workflows.

### 4.6. Open-Source Dataset Release

This campaign releases an open-source dataset of 604 generated SysMLv2 outputs (151 prompts × 4 model backends), together with run-level records, iteration histories, and validator diagnostics in the project repository [33]. This provides a reproducible, inspectable benchmark-scale corpus for follow-on syntactic-reliability studies.

## 5. DISCUSSION

The core empirical result is that production-validator gating closes a large reliability gap left by single-shot generation. In our setting, single-shot acceptance is only 51.16% overall, while the same prompt–model cases reach 100.00% pipeline acceptance under iterative validator feedback. This matters operationally

because MBSE artifacts are only useful when accepted by the modeling environment; parser-passing but non-accepted artifacts cannot be reliably rendered, inspected, and used in downstream engineering workflows.

The per-model pattern is also informative. First-pass behavior differs substantially across models, but all models converge to full acceptance under the same loop structure and validator oracle. This supports the central design claim that deterministic validator feedback functions as a model-agnostic control signal: the loop does not depend on one provider having unusually favorable priors for SysMLv2 syntax. In practical terms, this suggests teams can treat validator-in-the-loop orchestration as infrastructure, then swap model backends based on cost, latency, or governance constraints rather than purely on first-shot syntax performance.

Secondary diagnostics help explain why this loop is effective. Early failures are concentrated in a small set of recurring error families (especially parsing and reference resolution), which are precisely the kinds of issues that deterministic diagnostics localize well. Because these errors are machine-detectable and location-specific, they are amenable to iterative repair without requiring weight updates or task-specific fine-tuning. The observed convergence profile, with most cases resolving in early iterations, further indicates that many failures are shallow syntactic/structural defects rather than irrecoverable generation breakdowns.

From a systems perspective, the observed residual-failure decay mirrors contraction behavior studied in fixed-point iteration [27, 34]. The measured trajectory is empirically consistent with a contraction-like regime in which each validator-guided cycle maps the current failure set to a smaller one. Unlike stochastic self-refinement loops, the deterministic validator oracle yields stable reduction in failure mass across successive cycles. Deterministic diagnostics provide structured error localization and prevent drift in the update direction. This positions production validation as a control operator over generation dynamics rather than merely a syntactic filter. The result is rapid movement toward the acceptance set under repeated repair cycles without observed oscillatory behavior.

This study has clear boundaries. First, all primary claims are syntactic and operational: we evaluate acceptance under a production validator, not semantic adequacy of architecture, behavior, or requirements fidelity. Second, results are reported for one benchmark suite (SysMBench prompt set) and one production validator backend; broader external validity across domains and validation ecosystems remains to be measured. Third, although we include an auxiliary ANTLR-versus-production demonstration to justify the oracle choice, that demonstration is diagnostic rather than a substitute for large-scale cross-tool studies.

Within those bounds, the practical implication is straightforward: if the goal is deployable NL-to-SysMLv2 generation, acceptance under production validation is the correct reliability target, and validator-in-the-loop refinement is an effective mechanism for reaching it.

## 6. CONCLUSION

This paper presents a validator-driven pathway for reliable natural-language–to–SysMLv2 generation. We instantiate a generate–validate–repair loop with a production validator oracle and evaluate it on all 151 SysMBench prompts across four model backends (604 prompt-level cases). The result is a large and consistent improvement in syntactic reliability: 51.16% single-shot acceptance versus 100.00% pipeline acceptance, with complete recovery of single-shot failures in this campaign. The campaign also releases an open-source dataset of all 604 generated outputs with associated validation traces for reproducible follow-on analysis [33].

Our contribution is therefore operational rather than architectural: we show that deterministic production-validator feedback can convert plausible-but-unusable outputs into production-validator–accepted artifacts at benchmark scale, without model retraining. The claim is intentionally scoped to syntactic acceptance under the chosen validator. Semantic correctness, architectural quality, and downstream engineering fitness remain open evaluation axes.

Future work should extend this foundation along three directions: (i) cross-validator replication to test portability across production modeling environments, (ii) semantic and task-level evaluation layered on top of syntactic acceptance, and (iii) optimization of iteration policies for lower runtime and token cost while preserving reliability.

## ACKNOWLEDGMENTS

## REFERENCES


[1] Estefan, Jeff. "Survey of Model-Based Systems Engineering (MBSE) Methodologies." Technical Report No. INCOSE-TD-2007-003-02. INCOSE MBSE Initiative. 2007. URL https://www.incose.org/docs/default-source/ProductsPublications/SE-Resources/mbse/mbse-methodology-survey-rev-b.pdf.

[2] Madni, Azad M. and Sievers, Michael. "Model-Based Systems Engineering: Motivation, Current Status, and Research Opportunities." *Systems Engineering* Vol. 21 No. 3 (2018): pp. 172–190. DOI 10.1002/sys.21438. URL https://doi.org/10.1002/sys.21438.

[3] INCOSE. "Model-Based Systems Engineering (MBSE) Initiative." https://www.incose.org/products-and-publications/se-resources/mbse (2025). Accessed February 2026.

[4] SEBoK Editorial Board. "Model-Based Systems Engineering (MBSE)." https://sebokwiki.org/wiki/Model-Based_Systems_Engineering_(MBSE) (2025). Systems Engineering Body of Knowledge, accessed February 2026.

[5] INCOSE. "Systems Engineering Vision 2035." Technical report no. International Council on Systems Engineering. 2022. URL https://www.incose.org/docs/default-source/aboutse/se-vision-2035.pdf.

[6] Group, Object Management. "OMG Systems Modeling Language (SysML) v2 Specification, Version 1.0 Beta." Technical report no. Object Management Group (OMG). 2024. URL https://www.omg.org/spec/SysML/2.0/. Official release draft, September 2024.

[7] Object Management Group. "About SysML 2.0 (Normative and Machine-Readable Artifacts)." https://www.omg.org/

spec/SysML/2.0/About-SysML (2025). Accessed February 2026.

[8] Object Management Group. "OMG Announces Formal Adoption of SysML v2." https://www.omg.org/news/releases/pr2025/07-21-25.htm (2025). Highlights textual and graphical notation and standard API/services layer; accessed February 2026.

[9] Peng, Sida, Kalliamvakou, Eirini, Cihon, Peter and Demirer, Mert. "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot." *arXiv preprint arXiv:2302.06590* (2023)DOI 10.48550/arXiv.2302.06590. URL https://arxiv.org/abs/2302.06590.

[10] Cui, Kevin Zheyuan, Peng, Sida, Quintana, Alexi and et al. "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers." *Working Paper* (2025)URL https://economics.mit.edu/sites/default/files/inline-files/draft_copilot_experiments.pdf.

[11] Demirer, Mert, Peng, Sida, Quintana, Alexi and et al. "The Productivity Effects of Generative AI: Evidence from a Field Experiment with GitHub Copilot." *Working Paper* (2024)URL https://mit-genai.pubpub.org/pub/v5iixksv.

[12] Bouamra, A. et al. "SysTemp." arXiv:2506.21608 (2025). URL https://arxiv.org/abs/2506.21608.

[13] Cibrián, F. et al. "Agent-Based SysML v2 Synthesis with Retrieval-Augmented Generation and Grammar Validation." *Computers in Industry* (2025)DOI 10.1016/j.compind.2025.104350. URL https://doi.org/10.1016/j.compind.2025.104350.

[14] Jin, Dongming, Jin, Zhi, Li, Linyu, Fang, Zheng, Li, Jia, Chen, Xiaohong and Luo, Yixing. "A System Model Generation Benchmark from Natural Language Requirements." *arXiv preprint arXiv:2508.03215* (2025).

[15] LaVoie, Chance, Andujar Lugo, Eladio and Kara, Levent Burak. "ANTLR-versus-Production-Validation Demonstration for SysMLv2." https://github.com/cmuchancel/SysMBench_Compiler_In_Loop/tree/main/experiments/antlr_vs_syside (2026). Ten-case demonstration showing grammar-parser acceptance without production-validation acceptance. Accessed February 21, 2026.

[16] Clarke, Edmund M., Grumberg, Orna, Jha, Somesh, Lu, Yuan and Veith, Helmut. "Counterexample-Guided Abstraction Refinement." *Computer Aided Verification (CAV)*: pp. 154–169. 2000. DOI 10.1007/10722167_15.

[17] Solar-Lezama, Armando. "Program Synthesis by Sketching." Ph.D. Thesis, University of California, Berkeley. 2008.

[18] Jha, Susmit, Gulwani, Sumit, Seshia, Sanjit A. and Tiwari, Ashish. "Oracle-Guided Component-Based Program Synthesis." *International Conference on Software Engineering (ICSE)*: pp. 215–224. 2010. DOI 10.1145/1806799.1806833.

[19] Alur, Rajeev, Bodik, Rastislav, Juniwal, Garvit, Martin, Milo M. K., Raghothaman, Mukund, Seshia, Sanjit A., Singh, Rishabh, Solar-Lezama, Armando, Torlak, Emina and Udupa, Abhishek. "Syntax-Guided Synthesis." *Formal Methods in Computer-Aided Design (FMCAD)*: pp. 1–8. 2013. DOI 10.1109/FMCAD.2013.6679385.

[20] Wang, Xin, Chen, Wenhu, Chen, Xinyun and Wang, William Yang. "Compilable Neural Code Generation with Compiler Feedback." *Findings of the Association for Computational Linguistics: ACL 2022*: pp. 138–150. 2022. URL https://aclanthology.org/2022.findings-acl.2/.

[21] Grubišić, A. et al. "Compiler Feedback for Large Language Models." arXiv:2403.14714 (2024). URL https://arxiv.org/abs/2403.14714.

[22] Sensmetry. "SysIDE: The Open-Source IDE and Compiler for SysML v2." https://sensmetry.com/syside/ (2024). Accessed November 2025.

[23] OpenAI. "GPT-5.2-Codex Model Documentation." https://developers.openai.com/api/docs/models/gpt-5.2-codex (2026). Model used in this campaign: `gpt-5.2-codex`. Accessed February 20, 2026.

[24] Anthropic. "Introducing Claude Sonnet 4.6." https://www.anthropic.com/news/claude-sonnet-4-6 (2026). API usage specifies `claude-sonnet-4-6`. Accessed February 20, 2026.

[25] DeepSeek. "Reasoning Model (`deepseek-reasoner`)." https://api-docs.deepseek.com/guides/reasoning_model (2026). Accessed February 20, 2026.

[26] Mistral AI. "Function Calling (Model Example: `mistral-large-latest`)." https://docs.mistral.ai/capabilities/function_calling/ (2026). Accessed February 20, 2026.

[27] Banach, Stefan. "Sur les Opérations dans les Ensembles Abstraits et Leur Application aux Équations Intégrales." *Fundamenta Mathematicae* Vol. 3 No. 1 (1922): pp. 133–181. DOI 10.4064/fm-3-1-133-181. URL https://eudml.org/doc/213289.

[28] Varga, Richard S. *Matrix Iterative Analysis*, 2nd ed. Springer, Berlin, Germany (2009). DOI 10.1007/978-3-642-05156-2. URL https://doi.org/10.1007/978-3-642-05156-2.

[29] Nocedal, Jorge and Wright, Stephen J. *Numerical Optimization*, 2nd ed. Springer, New York, NY, USA (2006). DOI 10.1007/978-0-387-40065-5. URL https://doi.org/10.1007/978-0-387-40065-5.

[30] He, Jiawei and Lin, Guangming. "Average Convergence Rate of Evolutionary Algorithms." *IEEE Transactions on Evolutionary Computation* Vol. 20 No. 2 (2016): pp. 316–321. DOI 10.1109/TEVC.2015.2444793. URL https://doi.org/10.1109/TEVC.2015.2444793.

[31] Polyak, Boris T. *Introduction to Optimization*. Optimization Software, Inc., New York, NY, USA (1987).

[32] Clopper, C. J. and Pearson, E. S. "The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial." *Biometrika* Vol. 26 No. 4 (1934): pp. 404–413. DOI 10.1093/biomet/26.4.404. URL https://doi.org/10.1093/biomet/26.4.404.

[33] LaVoie, Chance, Andujar Lugo, Eladio and Kara, Levent Burak. "SysMBench Compiler-in-the-Loop Campaign Repository." https://github.com/cmuchancel/SysMBench_

Compiler_In_Loop (2026). Code, artifacts, and analysis scripts for this study. Accessed February 21, 2026.

[34] He, Jiawei and Lin, Guangming. "Average Convergence Rate of Evolutionary Algorithms." *arXiv preprint arXiv:1504.02266* (2015)DOI 10.48550/arXiv.1504.02266. URL https://arxiv.org/abs/1504.02266.

## APPENDIX A. AUXILIARY DEMONSTRATION: GRAMMAR PARSABILITY VS. PRODUCTION VALIDATION

To support the design choice of using production validation (rather than grammar parsing alone) as the acceptance oracle, we ran an auxiliary demonstration included in the repository under `experiments/antlr_vs_syside/` [15]. This demonstration is not a second primary experiment; it is a construct-validity check showing that parser acceptance and production acceptance are distinct outcomes.

We evaluated 10 intentionally distinct SysMLv2 examples in `examples/mismatch_10_distinct/`. Each file was designed to remain grammar-parseable while violating a model-wide constraint typically enforced by a production validator. The evaluation pipeline was:

1. ANTLR parse check (third-party SysML parser integration in the repository).
2. Production validation check using SysIDE.

Results were unambiguous: all 10/10 examples passed ANTLR parsing, while 0/10 were accepted by production validation (10/10 mismatch cases). Validator diagnostics were dominated by unresolved-reference failures (9/10, `reference-error`), with one invocation-typing failure (1/10, `invocation-expression-instantiated-type`). This pattern directly illustrates that context-free syntax conformance is necessary but not sufficient for operational model acceptance in production modeling environments.

### Concrete Example

The following example is grammar-parseable but fails production validation:

```
package Demo07 {
part def Wheel {
attribute hubDiameter: LengthValue;
part tire { attribute width: LengthValue; }
attribute outerDiameter: LengthValue = (hub-
Diameter + 2 * tire.height);
}
}
```

ANTLR parsing accepts the structure. Production validation rejects it with:

```
error (reference-error): No Feature named
'height' found.
```

**TABLE 3: Ten-case auxiliary demonstration: all examples parse under ANTLR, none pass production validation.**

| ID | Injected Condition | ANTLR | SysIDE |
|---|---|---|---|
| 01 | Missing imported namespace | Pass | Fail (reference) |
| 02 | Missing port type | Pass | Fail (reference) |
| 03 | Missing attribute type | Pass | Fail (reference) |
| 04 | Missing specialization base | Pass | Fail (reference) |
| 05 | Action typed by undefined behavior type | Pass | Fail (reference) |
| 06 | Invocation target is not a behavior | Pass | Fail (invocation) |
| 07 | Missing referenced feature in expression | Pass | Fail (reference) |
| 08 | Missing root-qualified namespace | Pass | Fail (reference) |
| 09 | Redefinition of missing feature | Pass | Fail (reference) |
| 10 | Missing namespace in type use | Pass | Fail (reference) |