JOT
JOURNAL OF
OBJECT TECHNOLOGY

Automated Benchmark Generation for Object Constraint Language (OCL) with SMT-Based Verification

Ankit Jha, Rosemary Monahan, and Hao Wu
Computer Science Department, Maynooth University, Ireland

ABSTRACT OCL tools are growing but their evaluation still relies on small, manually selected and domain-specific examples. Existing OCL benchmarks typically lack scale, systematic coverage of OCL features and rarely verify whether constraints can hold simultaneously within a benchmark suite. This makes rigorous tool comparison difficult. We present an automated framework for generating solver-verified OCL benchmark suites from UML/Ecore metamodels. The framework combines a library of over one hundred reusable constraint patterns, an adaptive coverage-guided generation engine and an SMT solver-based verification pipeline that labels each constraint as satisfiable (SAT) or unsatisfiable (UNSAT) under bounded semantics. Our framework ensures that only well-typed, semantically meaningful and solver-verified OCL constraints are generated. We evaluate the framework on ten UML/Ecore metamodels covering multiple OCL language features and generating 50–100 machine verified constraints per model which approximates the user-required constraints. We further use the generated benchmark to evaluate existing OCL tools from the literature, identify major research gaps such as complex string functions, nested quantifiers, conflicts in constraint and specific feature testing. These benchmark generation techniques will advance the state of the art through evaluation of existing and future tools.

KEYWORDS Object Constraint Language (OCL), Automated Benchmark Generation, Satisfiability Modulo Theories (SMT), Formal Verification, Constraint Consistency, Tool Evaluation.

1. Introduction

The Object Constraint Language (OCL) is the standard language for specifying invariants, pre-conditions and post-conditions on UML metamodels in Model-Driven Engineering (Petrascu & CHIOREAN 2012). It captures domain rules that cannot be expressed using diagrammatic notation. Over the last two decades, a rich ecosystem of OCL-based tools has emerged, including parsers, editors, interactive validators, test generators and static analyzers based on Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers (Dadeau et al. 2019; Wu & Timoney 2020). Despite this progress, the evaluation of OCL tools remains severely limited. Publicly available benchmarks are typically small, manually created and centered on a narrow set of case studies (e.g., library or banking metamodel). They cover only a limited subset of OCL features and rarely provide a solver-backed satisfiable (SAT)/ unsatisfiable (UNSAT) label or verification of whether all constraints in a benchmark suite can be satisfied simultaneously. Without standardized and verified benchmarks, rigorous comparison of OCL tools remains difficult (Gogolla & Cabot 2016).

To address this gap, we present an automated framework for generating solver-verified OCL benchmark suites from UML/Ecore metamodels. Given a metamodel and a user-defined benchmark profile, the framework produces feature-diverse sets of OCL invariants together with structured

JOT reference format:
Ankit Jha, Rosemary Monahan, and Hao Wu. *Automated Benchmark Generation for Object Constraint Language (OCL) with SMT-Based Verification*. Journal of Object Technology. Vol. 25, No. 3, 2026.
 http://dx.doi.org/10.5381/jot.2026.25.3.a28

metadata and machine-verified SAT/UNSAT labels under bounded semantics. The framework consists of three integrated layers: a Universal Pattern Library of over one hundred reusable OCL templates; a coverage-guided benchmark engine that instantiates constraints according to user-defined features; and an SMT-based verification pipeline that assigns SAT/UNSAT labels and checks joint satisfiability across the suite (Jha et al. 2024). More broadly, the benchmark-generation principles underlying our framework are not limited to OCL. Related communities face similar challenges in constructing fair and comparable evaluations (Oberkampf & Trucano 2008), including relational model analysis (e.g. Alloy-style specifications), program analysis, verification tool chains and configuration languages. Similar ideas apply to SQL tooling (Ba & Rigger 2023), where coverage-driven benchmark workloads can be generated under controlled profiles to support systematic tool comparison. Our contributions can be summarized as follows:

1. Pattern-based generation. A library of 113 reusable OCL constraint templates consisting of nine feature families, together with a coverage-guided generation engine that guides constraint generation toward user-defined feature targets.

2. Enrichment and conflict generation. A metadata attachment *phase* and a mutation-based strategy for systematically generating conflicted constraints.

3. SMT-based verification. A verification pipeline that encodes each generated constraint into SMT formulas and checks it using bounded model finding with an SMT solver, assigning a label to indicate whether a set of OCL constraints is satisfiable or not.

4. Empirical evaluation. An evaluation across ten UML/Ecore metamodels from diverse domains, generating 50–100 verified constraints per metamodel and identifying concrete gaps in existing OCL tools, including weak support for string reasoning, nested quantifiers and conflict detection.

Paper Organisation

The paper is organized as follows. Section 2 introduces the running example used throughout the paper. Then it discusses the limitations of existing OCL benchmark suites and defines the formal benchmark objectives. Finally, it states the research questions addressed by our evaluation. Section 3 describes the pattern library, comprising 113 reusable OCL constraint templates organized into nine feature categories. Section 4 presents the coverage-guided generation engine, which instantiates patterns according to the user-defined OCL feature targets. Section 5 covers benchmark enrichment and conflict generation via a structured mutation strategy. Section 6 presents the SMT-based verification pipeline. Section 7 reports the empirical evaluation and cross-tool comparison, addressing all three research questions. Section 8 discusses community insights and the practical implications of the generated benchmarks. Section 9 discusses related work and Section 10 concludes the paper and outlines directions for future work.

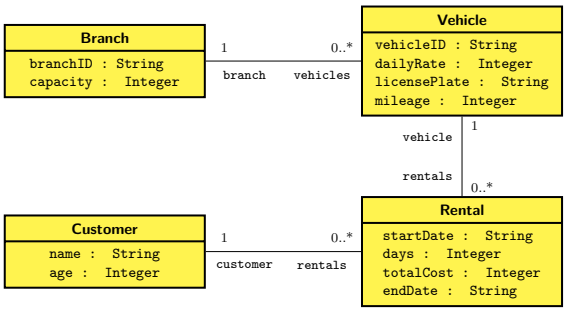

Figure 1 The *CarRental* metamodel consisting of four classes (`Branch`, `Vehicle`, `Customer`, `Rental`) connected by three associations with their role names and multiplicities shown.

2. Background

This section introduces a running example, illustrates the limitations of existing OCL benchmarks and states the research question addressed by this paper.

Running Example

To motivate the discussion, we use the *CarRental* metamodel (Figure 1). Throughout the paper, we use this model to illustrate pattern application, constraint generation and SMT encoding.

2.1. Limitations of Existing OCL Benchmarks

Existing OCL benchmarks reflect five recurring limitations:

1. Small benchmark suites: Publicly available examples typically contain ten's of constraints per metamodel which is insufficient for stress-testing performance or exploring diverse language features (Gogolla & Cabot 2016).

2. Low language-feature coverage: Existing benchmarks cover a narrow range of OCL constructs. They often repeat shallow navigation patterns that access only directly connected classes, rather than multi-step association paths (Jha et al. 2025). Most existing constraints rely on simple numeric comparisons, leaving quantified and higher-order OCL constructs underrepresented (Gogolla et al. 2013).

3. Uncontrolled redundancy: Manually designed suites may contain near-duplicate constraints, providing limited additional information on tool capabilities (Francisco & Castro 2012).

4. Lack of verification evidence: Existing benchmark suites rarely provide solver-confirmed labels distinguishing satisfiable from unsatisfiable constraints. In large models with hundreds of constraints, hidden logical inconsistencies may remain undetected. Without these verified labels, evaluations become difficult to reproduce and more prone to error (Salemi et al. 2016).

5. Suite-level consistency analysis: Many existing OCL tools support consistency checking for the metamodel (Gogolla et al. 2007). However, ensuring that all constraints

in a large suite can be satisfied simultaneously remains challenging because hidden conflicts may arise even when individual constraints are satisfiable.

2.2. Formal Definitions & Benchmark Objectives

Model and constraint language: Let $\mathcal{M}$ be a UML/Ecore metamodel and let $\mathcal{L}_{OCL}(\mathcal{M})$ denote the set of well-typed OCL invariants over M that are expressible within the supported feature set of our framework. The scope of this work is restricted to OCL *invariants*, which express properties that must hold for all instances of a class at all observable states. Pre and post-conditions on operations are not considered.

Benchmark suites and SMT based verification:

$$\mathcal{B} = \{\varphi_1, \ldots, \varphi_n\} \subseteq \mathcal{L}_{\text{OCL}}(\mathcal{M}),$$

An OCL benchmark suite $\mathcal{B}$ for $\mathcal{M}$ is a finite set equipped with metadata (e.g., a metamodel context and its intended feature coverage). To make the $\mathcal{B}$ *machine-verifiable* suite, we verify each generated constraint φ_i by translating $\mathcal{M}$ together with φ_i into an SMT encoding and checking it with an SMT solver. The verification result assigns a label $\ell_i \in \{\mathsf{SAT}, \mathsf{UNSAT}\}$: $\ell_i = \mathsf{SAT}$ if the solver finds at least one satisfying object model within the chosen scope and $\ell_i = \mathsf{UNSAT}$ if no such bounded object model exists. Here, the chosen scope specifies an upper bound on the number of instances per class considered during verification (i.e. $\mathsf{Scope}(C)$ is the maximum number of objects).

Consistency check: Let $\mathcal{M}$ be a fixed UML/Ecore metamodel and let $\mathcal{B} \subseteq \mathcal{L}_{\text{OCL}}(\mathcal{M})$ be a finite set of OCL invariants generated over $\mathcal{M}$. We say that $\mathcal{B}$ is *consistent* if there exists at least one object model (within the chosen bounded scope of $\mathcal{M}$) that satisfies all constraints in $\mathcal{B}$ simultaneously.

Research Questions: Based on the limitations identified above, we define the following research questions:
RQ1. Can the framework generate syntactically & semantically correct, machine-verified and diverse OCL benchmarks?
RQ2. To what extent does the coverage-guided generation engine produce benchmark suites that are diverse across OCL language features and constraint pattern families?
RQ3. Can we use generated benchmarks to clearly identify strengths and limitations of existing OCL analysis tools?

3. Pattern-Based Constraint Generation

The approach we present is a *template-based* constraint generator. We design this generator as an alternative to purely grammar-based fuzzing (Al Salem & Song 2019). Grammar-based fuzzing can generate expressions that are syntactically valid but semantically implausible for real UML/Ecore specifications. Here, we use a library of reusable OCL templates [1]. These templates capture common rules from models such as value limits, null checks, navigation rules and quantifiers over collections. Each template is a pre-validated OCL schema aligned with the OCL 2.4 standard (Object Management Group (OMG) 2014). Compared to unstructured constraint construction, this pattern-based design ensures that every generated constraint is syntactically correct. To instantiate a template, the generator selects classes, attributes, and associations from the input UML/Ecore metamodel and substitutes them into the template placeholders. We keep the generated constraints meaningful by using model-aware generation rules. When we fill a template, we only choose elements that match the correct type, multiplicity and navigation links. E.g., an attribute comparison template is instantiated only with attributes of compatible type, while a navigation template is instantiated only with valid association-end names and multiplicities. We use adaptive pattern selection, so we cover many OCL features without many repeated constraints.

3.1. Pattern Library

We design a pattern library consisting of 113 reusable OCL constraint templates. Each template captures a broad set of common OCL features derived from case studies and established best practices in model-driven engineering (Samimi et al. 2015; Gogolla et al. 2007; Cabot & Gogolla 2012). The templates are grouped into 9 categories, each corresponding to a primary OCL feature. Table 1 summarizes these categories, providing a short description and an example per group. Each template is also assigned a *complexity level* from 1 to 4. The complexity level depends on the number of navigations, the depth of nested quantifiers and the count of logic/arithmetic operators involved; for full details of these metrics, we refer to our previous work (Jha et al. 2025). E.g., a Level 1 pattern involves a single attribute access (e.g., `self.a > 0`), while a Level 4 pattern requires at least two navigation steps and a nested higher-order operation (e.g., `self.a.b->exists(x | x.c->forAll(...))`). We select patterns from different groups listed in Table 1 to ensure a broad coverage of OCL features during generation. The assigned complexity level controls how simple or complex the final constraint may be during the generation process. In the next sections, we explain how each pattern is selected with compatible metamodel elements.

Pattern Representation Each pattern in the library is formally represented by two components:
Textual template (T_p): A parameterized OCL expression with placeholder markers (denoted by `{...}`) that are substituted with concrete values during instantiation. E.g., the `numeric_bounded` pattern is defined by the given template.

[1] The complete pattern library is available in the online repository at https://github.com/AnkitMU/ABC_OCL/blob/main/templates/patterns_revised.json.

Category	Description	Context	Example
basic	Core constraints such as size, uniqueness, null checks, and simple numeric comparisons.	Branch	`self.vehicles->size() > 0`
advanced	Higher-order constraints such as `one`, implication guards, ordering/ranking or acyclicity.	Customer	`self.rentals->one(r | r.endDate > r.startDate)`
collection	Collection operations [`forAll`, `exists`, `select`, `collect`, `any`].	Customer	`self.rentals->forAll(r | r.days > 0)`
navigation	Single and multi-hop navigation chains, verifying safe handling of links and types across the metamodel.	Branch	`self.vehicles->forAll(v | v.rentals->forAll(r | r.days > 0))`
arithmetic	Arithmetic constraints over numeric attributes, including `div`, `mod`, and arithmetic relations.	Rental	`self.totalCost = self.vehicle.dailyRate * self.days`
string	String reasoning with `size()`, `substring()`, `matches`, and comparisons.	Vehicle	`self.licensePlate.size() = 7`
set_operations	Set-level operations such as `union`, `intersection`, `including`, and `excluding`.	Branch	`self.vehicles->intersection(self.vehicles)->size() = self.vehicles->size()`
tuple_let	Tuple literals and `let` bindings used to structure expressions.	Branch	`let c = self.capacity in c > 0`
ocl_library	OCL standard library operations [`oclIsUndefined` and `oclIsInvalid`].	Rental	`self.vehicle.oclIsUndefined() = false`

Table 1 OCL pattern categories with description, context class, and example based on the CarRental metamodel in Figure 1.

```
self.{attribute} >= {min_value} and
self.{attribute} <= {max_value}
```

Typed parameter set. Each template is stored with a *typed parameter set* that explains how its placeholders are filled. For a template t, we write this set as $P(t) = \{p_1, \ldots, p_n\}$. This connects an abstract template (E.g., `self.{collection}->size() >= {k}`) to a concrete metamodel, so the same template can be reused across different models. Each parameter $p_i \in P(t)$ has:

- **Name and expected kind:** what it represents (e.g., attribute, association, literal value, or an OCL subexpression) and its expected type.
- **Candidate source:** valid value sources (e.g., all numeric attributes, all collection-valued associations), so the instantiated constraint stays well-typed.
- **Required or optional:** whether it must be provided, and a default when optional.
- **Dependencies:** rules linking parameters (e.g., the two compared attributes must be different to avoid `self.age > self.age`).

This parameter definition allows the generator to fill templates using only metamodel-compatible choices and avoids trivial or meaningless constraints. It also separates the template structure from *the parameter selection*, making the library easy to extend and portable across models. Type-awareness is enforced at the template level, where each template allows only type-compatible parameters. E.g., string templates use `String` attributes such as `concat`, `matches` and `size`; arithmetic templates use numeric attributes such as `Int` or `Float` and collection templates use multi-valued associations. The framework also automatically distinguishes between `Set` and `Sequence` by reading the ordering property of each association directly from the metamodel. If an association is unordered, it is treated as a `Set`; if it is ordered, it is treated as a `Sequence`.

3.2. Adaptive Pattern Selection

A simple constraint generator could select templates uniformly at random. However, random selection performs poorly for metamodel-based OCL generation because many templates are not suitable for every class and some patterns are selected very often while others are rarely used. We therefore use *adaptive weighted sampling*. In our approach, each template is assigned a weight that depends on the current metamodel and the state of the generation process. Templates with higher weights are more likely to be selected. During generation, these weights are updated using feedback from previous attempts where successful templates become more likely to be selected, while templates that repeatedly fail become less likely. Adaptive weighted sampling addresses three problems of uniform random selection:

(i) Wasted generation attempts. Some templates are not applicable to the current class. E.g., a String template cannot be instantiated if the class has no `String` attributes, and a navigation template cannot be used if no association path exists. Under random selection, such templates may be chosen repeatedly and fail each time. Our approach lowers their weight after failed attempts, reducing wasted work.

(ii) Unbalanced coverage. Random selection often produces many simple constraints, because these templates can be instantiated easily for most classes. More difficult patterns, such as those using `forAll`, `exists`, `let`, `Tuple`, or long navigation chains, appear too rarely. We therefore use a weight function (Section 3.2.1) to maintain the desired balance between pattern diversity and the user-defined profile.

(iii) Repeated validation failures. Some templates repeatedly produce invalid or trivial constraints. Here, validation means that the instantiated constraint must be type-correct and non-trivial. For example, constraints such as `self.rentals->size() >= 0` or `self.rentals->size() <> self.rentals->size()` are rejected because they do not meaningfully restrict the model. We record these failures and reduce the weights of templates that repeatedly fail for a given class. The next subsections define the weight function and describe how feedback is used to update weights.

3.2.1. Weight Computation Let $p \in P$ be a candidate pattern and C a context class in the metamodel.[2] We allocate a score to each pattern. We call this score the *weight* $W(p, C)$ and use it to indicate how likely a pattern is to be selected for a class. We build this weight using five factors: *(1)* what the user wants. *(2)* how widely the pattern can be used across different metamodels. *(3)* how capable the verification tool is at handling the pattern (when verification is enabled). *(4)* how accurately the pattern matches the class C. *(5)* how well the pattern worked for the previous generation attempts for class C. Formally,

$$W(p, C) = w_{\text{base}}(p) \cdot w_{\text{gen}}(p) \cdot w_{\text{solver}}(p) \cdot w_{\text{rel}}(p, C) \cdot w_{\text{hist}}(p, C).$$

User preference $w_{\text{base}}(p)$. Reflects the user-defined priorities specified in the user configuration. Users can configure their profiles to prioritize certain pattern categories. E.g., a user may decide to "select more quantified patterns" or "select more navigation patterns". If the user does not set anything, all patterns are given the same priority.

Structural Compatibility $w_{\text{gen}}(p)$. Some patterns have minimal structural requirements and they are compatible with a wide range of metamodels. These patterns target OCL-specific data types, such as null or undefined checks (`self.attr <> null`) and basic collection properties (`self.items->notEmpty()`). We assign these patterns a positive compatibility weight to ensure that well-typed constraints can still be generated in parts of the metamodel with limited navigation structure. This helps maintain coverage of core OCL data types while reducing generation failures when the metamodel lacks the structural diversity required for more complex patterns.

Verification reliability factor $w_{\text{solver}}(p)$. This factor reflects how consistently a pattern produces a definitive SAT or UNSAT verdict from the SMT solver. Patterns whose encoding maps directly to standard SMT theories (e.g. arithmetic, cardinality) tend to resolve reliably, whereas patterns involving deep quantifier nesting or long navigation chains are more prone to solver timeouts. We therefore assign a slightly higher sampling weight to patterns with higher verification reliability. This factor only adjusts how often a pattern is *attempted*; it does not exclude any pattern from generation and coverage across all enabled pattern families is preserved.

Fits the class $w_{\text{rel}}(p, C)$ A pattern must match the context of the class. E.g., a numeric rule needs a numeric attribute, and a navigation rule requires the correct associations. If the class does not have what the pattern needs, we skip those patterns.

Run-time adaptation $w_{\text{hist}}(p, C)$ During each generation run, the engine maintains counters for every pattern–class pair (p, C), tracking successes and failures. Success occurs when the pattern can be fully applied to class C, getting a well-typed and semantically meaningful OCL constraint (excluding trivial cases such as `self.age > self.age`). A failure occurs when parameter binding is impossible or results in an ill-typed constraint due to structural mismatch. After several attempts for the same (p, C) pair, the empirical success rate is used to dynamically scale the sampling weight. Patterns that frequently succeed for class C receive a positive weight adjustment, while those that repeatedly fail are downgraded. This adaptive strategy reduces wasted attempts by prioritizing structurally compatible patterns. $w_{\text{hist}}(p, C)$ is updated after each generation attempt, allowing continuous adaptation of the sampling distribution within a generation run.

Weighted sampling. After computing the weights for the candidate pattern set P_C, the generator selects the patterns using proportional weighted sampling. Patterns with higher weights are more likely to be chosen, while lower-weight patterns remain selectable ensuring diversity in the generated constraints. The engine performs a lightweight applicability check for the target class C after a pattern is sampled. All required parameters must have at least one valid binding for C or a usable default value. If this condition is not satisfied, then the sampled pattern is discarded and the selection process continues. The sampling distribution can change during a generation process because the history-based weight component $w_{\text{hist}}(p, C)$ is updated during generation. This results in an adaptive selection process that balances coverage and redundancy as generation progresses.

3.3. Context-Aware Parameter Generation

After selecting a pattern p and context class C, the engine must fill the set of parameters of the pattern $\text{ParamSet}(p)$ with specific values taken from the metamodel. This step affects benchmark quality, as the same template may yield either a restrictive constraint or a trivial one depending on the chosen bindings. In the CarRental example (Figure 1), if the context class is `Branch` then some patterns require a numeric attribute (such as `capacity` or `mileage`) while other patterns require a collection-valued link (such as `vehicles` or `rentals`). If we pick the wrong kind of feature then the constraint can be ill-typed, meaningless or unrealistic (e.g., comparing a date-like attribute to a money-like attribute). It can also become incorrect constraints, such as a self-comparison like `self.age > self.age`. To avoid such problems, our benchmark generation engine uses checks and filters while choosing the parameter values.

Type-compatible feature selection. For parameters referring to metamodel features (attributes or associations), the engine retains only type-compatible candidates. E.g., if a pattern requires an integer attribute then only integer-typed attributes of class C are considered. This prevents type mismatches, such as comparing strings with integers or applying numeric operators to Boolean values. For com-

[2] Notation: $\mathcal{M}$ is UML/Ecore metamodel, P is the pattern library, p is one pattern, C is a context class and $W(p, C)$ is the selection weight of pattern p for class C.

parison patterns (e.g. `self.a > self.b`), both operands must be type-compatible. In the case of enumeration comparisons, they must belong to the same enumeration type.

Configurable Semantic Grouping Static type compatibility alone is insufficient for generating meaningful comparisons and it does not attempt dynamic typing. To prevent semantically mismatched comparisons (e.g. comparing a `dailyRate` value to a `name`). To ease this, the engine employs an extensible name-based heuristic grouping strategy defined by a configurable dictionary of keyword to concept mappings (e.g. `Monetary` → {`price, cost, fee`}, `Temporal` → {`date, at, when`}). The grouping logic is domain independent. The engine prefers to bind operands within the same semantic group to improve semantic alignment of generated comparisons, but defaults to standard type-based matching if identifiers do not match known keywords. This design preserves general applicability across arbitrary models while allowing users to register domain-specific vocabularies when desired. The heuristics therefore act as a lightweight safeguard against impossible constraints (e.g. `startDate > totalAmount`) without imposing restrictions on the generator.

Tautology and contradiction avoidance. The engine filters parameter bindings that produce constraints that are always true or always false. Although such cases may be useful for stress testing, our goal is to generate semantically meaningful benchmarks rather than trivial edge cases. A common example is self-comparison, where the same attribute appears on both sides:

`self.age > self.age` (always false)
`self.age = self.age` (always true)

To avoid this, the engine enforces distinct attribute selection whenever a pattern requires multiple comparison operands. It also prevents internal contradictions within a single constraint. For bounded numeric patterns, lower and upper bounds are generated dynamically and the engine ensures that the lower bound is strictly less than the upper bound. This avoids infeasible constraints such as `self.value >= high and self.value <= low`. All checks are performed before constraint construction, reducing wasted attempts and improving quality.

Pattern-specific validation. After the engine fills all parameters, the engine performs pattern-dependent validation checks. For bounded numeric patterns, bounds are sampled from small fixed ranges (e.g. `min_value`/`max_value` from 1–100), ensuring the minimum is strictly less than the maximum. For collection size ranges, limits are drawn from small intervals (typically 1–3), with a check that the lower bound does not exceed the upper bound. For sub-string patterns, the start index is sampled from a small range (e.g.1–5) and the end index is required to be greater than the start. If any check fails, parameter generation raises an error and we redo it. These safeguards ensure that generated constraints remain syntactically valid, type-correct and non-degenerate.

4. Coverage-Guided Generation Engine

The Coverage-Guided Generation Engine builds a full benchmark suite, not just one constraint. It follows the user configuration and tries to generate the requested number of constraints with diverse feature coverage mentioned in Table 1. The engine targets the OCL features given below.

- **Language constructs:** collection operators (`size`, `forAll`, `exists`, etc.), arithmetic and logical operators, string operations, and library/type functions.
- **Structural features:** classes, attributes, associations and navigation depth.
- **Complexity features:** quantifier nesting, navigation chain length and operator combinations.

As presented in Algorithm 1, the generation process is organized into three different phases. **Profile-Guided Generation** decides how many constraints to generate from each pattern group. **Coverage driven back filling** works on missing OCL feature coverage (like operators, navigation depth and quantifier depth) requested by the user. **Redundancy Pruning** removes very similar constraints.

Phase 1: Profile-Guided Generation. The user profile *prof* specifies a total constraint budget N_{total} together with a target distribution over predefined *generation families* (e.g. cardinality, quantified, navigation). These families differ from the library categories in Table 1. E.g., a profile requesting `constraints: 100, sat_ratio: 0.8, unsat_ratio: 0.2, cardinality: 20%, uniqueness: 15%, navigation: 25%, quantified: 20%, arithmetic: 15%, string: 2%, enum: 3%, min: 3, max: 20` (per-class budget of minimum 3 and maximum 20 constraints). Although the table groups patterns by syntactic form and feature type, generation families provide a high-level grouping used to control diversity during benchmark generation. From the profile, the engine computes a target count Q_f for each family f. If $N_{\text{total}} = 100$ and the profile assigns 25% to cardinality and 20% to quantified constraints, then $Q_{\text{card}} = 25$ and $Q_{\text{quant}} = 20$. Patterns from different library categories can contribute to the same family. E.g., `collection` or `set_operations` patterns may count toward cardinality, while iterator-based patterns contribute to quantified constraints. For each family f, the engine repeatedly generates candidate constraints until either (i) $\text{countFamily}(B, f) = Q_f$ (ii) the global budget N_{total} is reached (iii) a category-specific attempt limit K_f is exceeded. Each generation attempt proceeds as follows:
1. Context selection. A context class C is chosen from the metamodel, respecting the limits per-class.
2. Pattern sampling. A pattern $p \in f$ is sampled using the weighted distribution defined in Section 3.2.1.
3. Parameter binding and construction. Compatible metamodel elements are bound to the parameters of p and

a concrete OCL constraint c is generated.

4. Acceptance checks. The candidate c must satisfy lightweight checks, including type-correctness, non-triviality (e.g., excluding tautologies) and similarity filtering against recently accepted constraints.

Only accepted constraints are added to the benchmark suite $\mathcal{B}$ and counted toward Q_f. Rejected candidates do not contribute to the target and generation continues until one of the stopping conditions above is met. At the end of Phase 1, the benchmark usually matches the category proportions correctly. However, some coverage goals can still be weak (e.g., not enough `forAll`, or constraints may lack sufficient navigation depth).

Algorithm 1 Coverage-Guided Benchmark Generation

Input: Metamodel $\mathcal{M}$, Pattern Library P, Profile *prof*, Budget N_{total}
Output: Benchmark suite B

$B \leftarrow \emptyset$
Compute family targets Q_f from *prof*

Phase 1: Profile-Guided Generation
for each family f with target Q_f **do**
  $a_f \leftarrow 0$
  while count$(B, f) < Q_f$ **and** $|B| < N_{\text{total}}$ **and** $a_f < K_f$ **do**
    $a_f \leftarrow a_f + 1$
    Context selection: choose context class C
    Pattern sampling: sample $p \in f$ (weighted)
    Parameter binding: $params \leftarrow$ bindParams(p, C)
    Constraint construction: $c \leftarrow$ generateOCL$(p, C, params)$
    Acceptance checks: compatibility, non-triviality, similarity
    if c is accepted **then**
      $B \leftarrow B \cup \{c\}$

Phase 2: Coverage-Driven Backfilling
$noProgress \leftarrow 0$
while $|B| < N_{\text{total}}$ **and** $noProgress < K_{\text{stall}}$ **do**
  $gaps \leftarrow$ coverageDeficits()
  if $gaps = \emptyset$ **then**
    break
  choose pattern p for $gaps[0]$
  if $p =$ null **then**
    break
  choose context C
  if $C =$ null **then**
    break
  $params \leftarrow$ bindParams(p, C)
  $c \leftarrow$ generateOCL$(p, C, params)$
  Acceptance checks: same as Phase 1
  if c is accepted **then**
    $B \leftarrow B \cup \{c\}$
    $noProgress \leftarrow 0$
  else
    $noProgress \leftarrow noProgress + 1$

Phase 3: Redundancy Pruning
if redundancy pruning is enabled **then**
  Remove near-duplicates on normalized tokens
return B

Phase 2: Coverage-Driven Backfilling. After Phase 1, the engine evaluates the coverage criteria derived from *prof* and identifies the remaining gaps. Our coverage is tracked along three dimensions:

- **Operator gaps:** insufficient occurrences of selected operators (e.g., `forAll`, `exists`, `implies`, `select`, `collect`).
- **Navigation gaps:** insufficient constraints containing longer navigation chains.
- **Quantifier gaps:** insufficient constraints with deeper quantifier nesting.

While coverage gaps remain and the global budget N_{total} has not been reached, the engine continues to attempt to fill these gaps. One missing target is selected and a pattern that can help with that target is chosen to generate new constraints using the same steps as Phase 1. If the candidate is accepted, it is added to the benchmark suite $\mathcal{B}$. The process terminates when either all coverage objectives are satisfied, the constraint budget is exhausted or no suitable pattern can be found for the selected gap.

Phase 3: Final Redundancy Pruning. A redundancy pruning step may be applied, depending on the user configuration *prof* after the generation and back filling of the coverage. The goal of this phase is to reduce near-duplicate constraints within the final benchmark suite $\mathcal{B}$. The engine compares each constraint in $\mathcal{B}$ against the others using a token-based similarity measure. Specifically, we compute the Jaccard similarity over normalized constraint tokens (Bag et al. 2019). If the similarity is higher than a chosen threshold, the engine rejects the constraint. This helps prevent near-duplicate constraints from filling the benchmark.

5. Benchmark Enrichment

After the Coverage-Guided Generation Engine (Section 4) produces an initial set of constraints, the framework applies a benchmark enrichment process. This process has two objectives:

- **Metadata attachment:** Each constraint is augmented with structured, machine-readable metadata (e.g. pattern identifier, pattern category, used OCL operators, navigation depth and assigned complexity level). This metadata supports evaluation of OCL tools, enables feature-level performance analysis, reproducibility of experiments and allows researchers to filter or construct feature targeted benchmark.
- **Conflicts Generation:** we generate *conflicting constraints* by systematically mutating selected constraints. We apply targeted operator-level transformations while preserving the overall structural pattern. E.g., the satisfiable constraint `self.rentals->forAll(r | r.days > 0)` is mutated by conjoining it with a direct existential counterexample `self.rentals->exists(r | r.days <= 0)`, producing a structurally unsatisfiable variant.

After this process, we get an enriched constraint suite consisting of the original candidates and their mutated conflicted constraints each annotated with metadata. This suite is then passed to the SMT-based verification layer (Section 6) for final SAT/UNSAT labeling and consistency checks.

5.1. Metadata and Difficulty Labelling

For each candidate constraint produced by the generation engine (Section 4), the enrichment procedure computes and attaches structured metadata in four groups.

Pattern information:

– pattern identifier (linking back to the pattern library).
– pattern category (e.g., `string`, `collection`, `navigation`).

Syntactic features:

– operators used (e.g., `=`, `<>`, `>=`, `and`, `or`, `implies`),
– navigation depth (maximum dot-navigation chain length in the invariant body).
– quantifier depth (approximate maximum nesting depth of collection such as `forAll` and `exists`).
– collection operators used (e.g., `size()`, `select()`, `collect()`, `isUnique()`).

Context information:

– context class
– referenced classes and associations
– referenced attribute data types (e.g., integers, strings, booleans).

Difficulty indicators:

– an assigned difficulty band (e.g., `easy`, `medium`, `hard`) derived from structural complexity such as navigation depth, quantifier depth and operator usage.

Difficulty is treated as a reproducible heuristic. Constraints with deeper quantifier nesting and longer navigation chains are typically more challenging for verification compared to simple Arithmetic operation (Jha et al. 2025). Encoding difficulty in metadata supports controlled experiments by enabling targeted sub-suites (e.g., "deep navigation" or "quantifier-heavy" constraints).

5.2. Generating Conflicting Constraints

To evaluate verification tools on both satisfiable and unsatisfiable constraints, we apply a structured mutation strategy on generated candidate constraints. This section describes the 4 mutation operators used by our framework. Each operator targets a specific source of logical inconsistency in OCL invariants while preserving the original structural context. Rather than arbitrarily negating operators, each mutation introduces a controlled logical conflict into an existing candidate constraint, producing a UNSAT constraint that is then confirmed by the SMT solver.
The four mutation operators are defined as follows.
MO1: Range inversion. The lower bound is strictly set above the upper bound, leaving the valid range of values empty. E.g., the original constraint:

```
context Branch inv:
  self.capacity >= 5 and self.capacity <= 10
```

is mutated by inverting the bounds to:

```
context Branch inv:
  self.capacity >= 10 and self.capacity <= 5
```

No integer satisfies both conditions simultaneously, making the invariant unsatisfiable.
MO2: Cardinality contradiction. The mutation adds a predicate that directly contradicts the original size assertion. E.g., the original constraint:

```
context Branch inv:
  self.vehicles->notEmpty()
```

is mutated by adding a contradicting predicate:

```
context Branch inv:
  self.vehicles->isEmpty() and
  self.vehicles->notEmpty()
```

Requires the same collection to be simultaneously empty and non-empty, which is unsatisfiable.
MO3: Quantifier counterexample. The mutation adds an existential clause whose body is the negation of the universal body, creating a structural contradiction. E.g., the original constraint:

```
context Branch inv:
  self.rentals->forAll(r | r.days > 0)
```

is mutated by adding a contradicting existential clause:

```
context Branch inv:
  self.rentals->forAll(r | r.days > 0) and
  self.rentals->exists(r | r.days <= 0)
```

To ensure that the two clauses directly contradict each other, we conjoin the original `forAll` with an `exists` counterexample, rather than simply negating the quantifier. A plain negation such as `exists(r | r.days <= 0)` may be satisfiable on its own; the contradiction arises when it is combined with the original universal constraint.
MO4: Boolean contradiction. The mutation asserts both true and false on the same attribute. E.g., the original constraint:

```
context Vehicle inv:
  self.isAvailable = true
```

is mutated by adding a contradicting assertion:

```
context Vehicle inv:
  self.isAvailable = true and
  self.isAvailable = false
```

No boolean value can satisfy both assertions simultaneously.
Although each of the four mutation operators is designed to produce an unsatisfiable constraint, the final SAT/UNSAT label is always confirmed by the SMT-based verification stage (Section 6) to ensure machine-verified results.

6. Verification Pipeline

In our verification pipeline, we perform bounded satisfiability checks. Constraints are labeled SAT or UNSAT with respect to this bounded universe. Our verification pipeline ensures that the generated constraints are both syntactically and semantically meaningful with consistency among the constraints in the benchmark suite.

6.1. Pattern Normalization

The generation library (Section 3.1) contains 113 *universal* templates to maximize diversity during synthesis and cover as many OCL features as possible. For verification, many

of these templates are semantically equivalent and differ only syntactically. We therefore normalize them into 50 core patterns, each with a dedicated SMT encoding (Jha et al. 2024). Since SMT solvers do not understand OCL syntax directly, the verification layer must determine which core encoding applies to each generated constraint. This is not a general-purpose OCL-to-SMT translator, but a pattern-based translation pipeline. Pattern identification uses two layers. First, during generation (Section 4), each constraint is tagged with its source pattern identifier, which is passed to the verification layer and used to select the SMT encoding. Second, a regex-based detector with more than 500 compiled regular expressions re-checks the normalized OCL text independently of the metadata. If the two disagree, the constraint is flagged and not sent to the SMT encoder until the mismatch is resolved. This design improves reliability when several generation templates map to the same core pattern.

6.2. SMT Encoding

Our verification pipeline encodes the OCL constraints into SMT formulas under bounded model semantics. Each normalized constraint produced by the pattern-mapping layer (Section 6.1) is mapped to core patterns (rewritten when applicable) before encoding. Verification is performed over a finite scope where the metamodel is encoded with a bounded number of objects per class. The solver then determines whether a satisfying model exists within this bounded universe.

Model Feature Encoding: Under bounded semantics, each class C is encoded as a finite set of potential instances indexed by $i \in \{0, \ldots, n_C - 1\}$. A Boolean variable $present_C(i)$ indicates whether the i^{th} instance participates in the candidate model. Single-valued attributes (e.g. integers, boolean or string) are represented as indexed arrays $attr[i]$, storing one value per instance. Multi-valued associations are encoded as Boolean relation matrices $R_{C.r}[i][j]$, where each entry denotes whether the j^{th} target instance is linked to the i^{th} source instance. Single-valued associations are modeled as integer index variables $r[i] \in [0, n_T)$, together with a presence bit $r_{present}[i]$ for optional (0..1) references. Referential integrity is enforced if a source instance and its reference are present, the referenced target must also be present. OCL invariants are interpreted as universally quantified over present instances. The OCL keyword `self` corresponds to the current instance index i of the context class.

$$\forall i.\ present_C(i) \Rightarrow body(i),$$

where `body`(i) is the SMT translation of the invariant body with `self` bound to index i. Thus, `self.attr` becomes $attr[i]$, and `self.assoc->size()` becomes a bounded sum over the corresponding row of the relation matrix. Single-valued references encode null/undefined using their presence bits. For reference r, `self.r = null` is translated as $\neg r_{present}[i]$, while `self.r <> null` becomes $r_{present}[i]$.

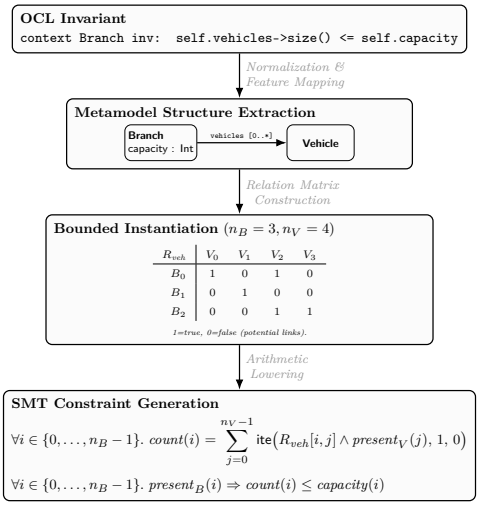

Figure 2 Encoding OCL semantics in SMT: Example of a size constraint encoded via bit-vector summation over a bounded instance relation.

The required references are always defined for the present instances.

Encoding Example: From Figure 2, we illustrate the encoding of a collection cardinality constraint. Given a fixed scope with n_B potential `Branch` instances and n_V potential `Vehicle` instances, the encoding introduces symbolic SMT variables that represent a candidate model. The association `vehicles` is represented as a Boolean matrix $R_{vehicles}[i][j]$, where each entry is a decision variable. The solver assigns each entry true or false. The 0/1 values in Figure 2 are only an illustrative satisfying assignment. For a branch instance B_i, the OCL expression `self.vehicles->size()` is encoded as:

$$count(i) = \sum_{j=0}^{n_V-1} \text{ite}\big(R_{vehicles}[i,j] \wedge present_V(j),\ 1,\ 0\big).$$

Here, ite is the SMT if-then-else operator which is used to convert a Boolean condition into an integer contribution for summation. Thus, $count(i)$ counts how many *present* vehicles are related to B_i. The invariant is then enforced by guarding the inequality with the branch presence bit.

$$\forall i.\ present_B(i) \Rightarrow count(i) \leq capacity(i).$$

The solver checks whether there exists an assignment to these variables that satisfies the constraint within the chosen scope.

Encoding Process: Across all supported patterns, the encoding follows three conceptual steps.
(i) Structure extraction, where the normalized OCL constraint is analyzed to identify the context and features.
(ii) Bounded representation, where the presence variables and association matrices are initialized for the scope.
(iii) Constraint generation, where the logical content is encoded into SMT formulas using arithmetic and Boolean primitives.

6.3. Encoding scope and limitations.

The current SMT encoding covers the OCL 2.4 constructs listed in Table 1, including navigation, collection operations, quantifiers, arithmetic, string operations, Boolean logic, and `let`/tuple expressions. However, three OCL features are not yet covered by the present encoding and are therefore outside the scope of the current benchmark generation framework.

Type checks and casting: Operations such as `oclIsKindOf()`, `oclIsTypeOf()`, and `oclAsType()` require an explicit encoding of inheritance and subtype relations. Supporting these operations would require adding type descriptor variables for each object and encoding the full inheritance hierarchy as additional SMT constraints. These are feasible via additional custom functions, but are outside the scope of the current implementation and are planned for the upcoming version.

Three-valued logic: OCL defines three-valued semantics where navigation over null may produce `invalid`, which propagates through expressions. Our encoding supports basic null checks using presence bits (e.g., `self.r <> null`), but does not model full `invalid` propagation across nested expressions.

Global and transitive operations: Operations such as `allInstances()` and `closure()` are not supported in the current version. Under our bounded encoding, `allInstances()` is possible by enumerating all object instances within the configured bound and `closure()` can similarly be achieved within bounded depth. These are planned extensions for the upcoming version.

6.4. Constraint Verification

Given SMT encoding, the solver returns one of three outcomes: SAT, UNSAT or UNKNOWN (typically due to timeouts). Unknown outcomes are not retained but are reported as warnings rather than being automatically discarded. We perform a consistency check on all SAT-labeled constraints. If any one constraint is contradicting with another constraint, the system reports MODEL IS INCONSISTENT. All verifications are done under a bounded semantics. A SAT result confirms the existence of at least one model within the chosen scope. An UNSAT result indicates that no model exists at that specific scope. If we increase the scope, result may change to SAT.

7. Evaluation

This section reports our experimental evaluation, which addresses the three research questions defined in Section 2. The evaluation addresses a long-standing gap in the Model-Driven Engineering (MDE) community. Although decades of research have produced numerous OCL verification tools, it is difficult to compare them because different papers use different examples, and many tools are no longer maintained (Gogolla et al. 2013). Without standardized and solver-verified benchmarks, there is no clear and fair picture of what existing OCL tools can actually do. Our goal is to generate benchmark suites that are diverse, metamodel-aware and solver-verified with labeled SAT/UNSAT results, giving the community a common test set to measure tool support, feature coverage and correctness.

7.1. Experimental Setup

Implementation and environment: The framework is implemented in Python(3.13.9) with Z3(4.14.1) as the SMT solver. All experiments were performed on an Apple M1-based machine (16 GB RAM). We used a **15-second timeout per** solver call. We chose this timeout to control the total experiment time because we checked many constraints at a time. Longer timeouts are possible, but they can make the full run much slower. It is available on GitHub https://github.com/AnkitMU/ABC_OCL

Model, Benchmark and Tools under evaluation: We evaluated ten metamodels that are diverse across various application domains. Table 2 summarizes their structural characteristics and verification results. For each model, we configure a benchmark profile requesting 50-100 constraints with a target split of 80% SAT and 20% UNSAT. The profile specifies category proportions (35% cardinality/uniqueness, 20% quantified, 25% navigation-heavy, 20% basic attribute invariants & string) and requests coverage over all classes with minimum & maximum value as 3 and 15 constraints per class respectively. Our framework does not always achieve the exact requested distribution because some generated constraints are rejected during validation. We therefore report the ratio between the requested and successfully generated constraints as the generation efficiency. We generated 10 benchmark suites with average of 61 SAT labeled constraints and 17 UNSAT. For cross-tool comparison, we focus on 7 OCL-based tools listed in Table 3. The tools mentioned are no longer maintained or cannot be downloaded and run easily. However, actively maintained tools such as USE Validator (Gogolla et al. 2007) and Eclipse OCL (Willink 2020) are publicly available. Since not all tools could be executed under uniform conditions, Table 3 gives a feature comparison based on the original research article from the literature rather than the direct execution of the tool . We also evaluate our SMT-based framework and systematically compare its capabilities against other existing OCL tools based on our research question. The tools mentioned in Table 3 cover the important aspects of how OCL is evaluated, such as (1) interpreter-based verification (2) diagnosis and optimization (3) translation to other solvers.

Benchmark usage and tool mapping: The generated benchmarks are intended to be reusable across different OCL analysis tools. Each benchmark suite contains three artifacts: (i) the metamodel in Ecore XMI format, (ii) the generated OCL 2.4 invariants, and (iii) a JSON metadata file. The metadata file records, for each constraint, its context class, pattern family, SAT/UNSAT label, difficulty level and complexity measures. To execute a benchmark

with a specific tool, the metamodel and constraints must first be converted into the tool's expected input format. E.g., Eclipse OCL can directly load the `.xmi` metamodel together with the generated `.ocl` file. In contrast, USE requires a single `.use` specification in which the metamodel and invariants are merged. The user should configure the target tool's search scope to match the bounded parameters used during benchmark generation to ensure a fair evaluation.

7.2. Verification, Scalability and Coverage

This subsection addresses **RQ1** and **RQ2** defined in Section 2. We evaluate our framework on three dimensions such as verification, scalability and coverage of the generated benchmarks.

Verification for each constraint is done through a three-stage pipeline consisting of syntactic parsing, semantic type-checking and SAT/UNSAT labeling using the SMT solver(Z3). As shown in Table 2, our framework achieves an average generation efficiency of 89% .

$$\text{Efficiency} = \frac{N_{\text{verified}}}{N_{\text{attempted}}}$$

where N_{verified} denotes the number of constraints that receive a definitive SAT or UNSAT label from the solver and $N_{\text{attempted}}$ denotes the total number of generation attempts across all domains. Rejected constraints during generation come primarily from semantic inconsistencies during parameter generation or solver timeouts on deeply quantified constraints. Constraints that produce unknown results due to timeout are discarded. Our framework ensures that all retained labels are solver verified for reproducible evaluation.

Scalability is assessed by analyzing the verification time relative to benchmark size and model complexity. Although the generator can generate hundreds of constraints, smaller models eventually run out of unique logical combinations. This happens due to the exhaustion of meaningful context–pattern combinations rather than computational limits. The execution time of the solver contributes directly to the overall generation time and increases with the number of evaluated constraints. Across all ten metamodels, the framework achieved an average verification time of 7.9 seconds per benchmark suite. For a benchmark of 100 constraints and a bounded search scope of five instances per class,our framework generates within predefined timeouts on a standard hardware. This demonstrates the practicality of the approach for large-scale benchmark generation.

Coverage is evaluated across pattern categories, operator usage and structural dimensions such as navigation and quantifier depth. The coverage-guided engine respects target quotas within approximately ±3% points and ensures inclusion of less frequently used operators, including `implies` and `xor`. We reduce redundancy by 20% of near-duplicate candidates are removed using token-based similarity metrics as specified in the benchmark profile. It efficiently handles the generation of conflicted constraints through the mutation strategy. Overall, the resulting benchmarks are solver-verified, non-redundant and designed to uncover the functional limitations in existing OCL verification tools.

7.3. Cross-Tool Comparison

This subsection addresses **RQ3** defined in Section 2. A primary objective of our framework is to provide a systematic re-evaluation of OCL tools through a feature-driven, machine verified benchmark suite. We compare seven existing OCL-based tools in 2 dimensions: language feature support and analysis capabilities. These criteria are defined in Table 3.

Verification and analysis capabilities: In Table 3, five of the evaluated tools support basic consistency checking for individual constraints, but support for overall analysis of suite remains limited. Although overall analysis is partially addressed by tools like `EMFtoCSP` and `UMLtoCSP` through bounded model search but achieving comprehensive verification remains a significant computational challenge. Conflict localization is supported by `MaxUSE`, `OCL-Lite` and our framework. OCL-Lite provides conflict explanation through minimal inconsistent subsets. Other tools provide little or no diagnostic feedback. The Counter-example is a model instance that demonstrates why a certain property fails or proves that a set of constraints is inconsistent. It is supported by 5 tools, including our framework. OCL-Lite provides support through sample instance generation. With respect to scalability, several tools handle moderate constraint sets, but `CD2Alloy` exhibits performance degradation on larger benchmarks.

Language coverage and support: The qualitative comparison in Table 3 reveals that there is a big difference in what each tool can do in terms of the supported OCL features between the tools. Basic navigation and quantified expressions are widely supported by tools, but advanced language features remain inconsistently handled. Advanced features such as string operations are fully supported by `Eclipse OCL` and our framework. `EMFtoCSP` provides support through lightweight string reasoning, while USE supports basic string operation. `CD2Alloy` and `OCL-Lite` give more restricted coverage. It lacks support for several core features such as collection and type operations. More advanced features, like transitive closure and extended type operations, are only partly supported by a few tools such as USE, Eclipse OCL and MaxUSE. Notably, both `EMFtoCSP` and `UMLtoCSP` support `allInstances()`, which our framework does not currently encode due to the second-order quantification it requires in SMT solver.

Benchmarking Capabilities and Infrastructure: A key aspect of our approach lies in the systematic generation and formal verification of the benchmark. Earlier, the evaluation of OCL verification tools depended on manually created specifications or small test sets which are not designed for large-scale or systematic evaluation (Gogolla & Cabot 2016). Since existing tools primarily target constraint verification rather than benchmark generation, the absence

of a standardized evaluation method makes comparative analysis challenging. Our framework addresses this gap by providing a feature driven framework that produces machine confirmed SAT/UNSAT labels with detailed metadata.

8. Discussion and Community Insights

The evaluation across 10 experimental metamodels and 7 OCL tool comparison provides clear picture about our framework and it also reflects the current state of OCL verification tools. Despite decades of OCL tool development, existing benchmarks are not systematically tagged by language feature, making it difficult to compare how different tools handle specific OCL constructs such as quantified expressions, string operations or collection iterators (Gogolla et al. 2013; Gogolla & Cabot 2016; Noten et al. 2017). This shows the need for standard benchmarks with solver verified results. Research in this area is far from mature, as many tools cannot handle the complex features of OCL.

Reliability and Reproducibility: A primary motivation for this work is the difficulty in reproducing and systematically comparing the experimental results for OCL tools. Most tools are designed for constraint validation or analysis rather than benchmark construction. Therefore, they differs in the supported features, input formats and execution environments. Table 3, compares the capabilities of the tool rather than ranking tools for the benchmarking. Reproducibility challenges arise from several practical factors such as unavailable or unmaintained tools, missing or incomplete benchmark models, incompatible modeling formats (UML, Ecore) and the absence of solver-verified labels. This comes as a challenge for controlled cross-tool evaluation. Our framework reduces this issue by introducing a standardized benchmark suite and supports a transparent cross-tool evaluation.

Generalisability and Scaling: We tested the framework on ten different metamodel ranging from 8 to 28 classes. This shows that our framework is not designed for only one type of metamodel. However, real industrial metamodels may still be more complex. They might include custom data types and special domain rules. Currently, we encoded strings as numbers such as comparison, sub-string and length are supported but advanced string operations are only approximated. Full support for string data types with nested quantifiers often makes the solver timeout (Jha et al. 2024).

Relevance to the OCL Community: The gaps identified in Table 3 highlight clear directions for tool improvement. Many tools have weak string support and limited suite-level consistency checking. Our benchmarks are metamodel-aware, feature driven and solver-labeled. They can be used as stress tests and as regression tests when tools are updated or there is a new tool in OCL community. This helps ensure that what a tool claims matches what it can actually do.

9. Related Work

OCL verification and analysis tools: A few collections of constraints are available through tools such as USE (Gogolla et al. 2013), but these are mainly for academic demonstration rather than for systematic benchmarking. Many tools have been developed for OCL constraint analysis. USE (UML-based Specification Environment) (Gogolla et al. 2007) is widely used to validate UML/OCL models through interactive instance creation and constraint evaluation. MaxUSE (Wu 2017) extends USE with optimization-based model finding and QMaxUSE (Wu 2022) adds quantitative verification. These tools are strong for interactive exploration, but they often need manual effort to build complex model instances. Eclipse OCL (Noten et al. 2017) provides an essential infrastructure for parsing, type-checking and evaluating OCL constraints. However, it lacks the satisfiability-based verification capabilities necessary for our benchmark verification process.

Translation-based verification and SMT/CSP model finding: Several tools verify the OCL by translating it to other solver backend. UMLtoCSP (Cabot et al. 2007) encodes UML class diagrams and OCL constraints as constraint satisfaction problems, and EMFtoCSP (González et al. 2012) applies similar ideas to EMF models. CD2Alloy (Maoz et al. 2011) translates class diagrams to Alloy for finding a bounded model. Although these approaches support automated reasoning, they typically rely on manually constructed test sets and do not provide systematic coverage of OCL features. Recent work has also explored SMT solvers for OCL verification. (Jha et al. 2024) proposes an OCL to SMT-LIB encoding with a focus on collection operations and (Clavel et al. 2009) translates the OCL into many-sorted first-order logic for automated reasoning. Our work builds on these foundations, but our main focus is systematic benchmark generation with solver-confirmed labels, rather than translation techniques alone.

Benchmarks and empirical tool evaluation: Automated benchmark generation is common in other constraint-solving domains, but it is rare for OCL. SAT Competition (Järvisalo et al. 2012) and SMT-COMP (Reynolds et al. 2015) maintain large benchmark libraries built from encoding, random generator, and industrial sources. CNFgen (Lauria et al. 2017) shows how structured SAT instances can be automatically generated. These efforts demonstrate the value of large and diverse benchmarks, but they do not capture OCL's object-oriented and metamodel-dependent semantics. In the OCL community, examples exist (USE distributions and the Royal and Loyal model (Abukhalaf et al. 2023)), but they are small, manually designed and do not include SAT/UNSAT labels. Therefore, a systematic comparison of OCL tools is difficult. Surveys such as (Gogolla & Cabot 2016) compare tools qualitatively using feature checklists and studies such as (Wu 2022) compare tools in hand-selected models. Our work complements these efforts by enabling benchmark-driven evaluation that is reproducible and scalable across

Table 2 Evaluation and verification metrics for solver-verified benchmark suites across ten metamodels, including their specific metamodel properties and verification time in seconds.

Domain	Class	Attribute	Association	Constraints Requested	Efficiency	Time	SAT	UNSAT
CarRental	8	32	20	100	85	3.9	68	17
Library	15	35	48	50	86	5.8	33	10
University	18	46	42	70	90	8.1	49	14
Banking	17	37	22	100	93	7.4	73	20
E-Commerce	19	45	41	90	83	9.3	58	17
Hospital	22	53	41	100	90	11.1	70	20
Flight Booking	12	30	26	80	90	5.2	57	15
Social Network	14	31	26	75	90	5.6	52	15
Manufacturing	20	50	39	110	91	10.8	79	21
IoT Sensor	28	68	59	100	90	11.9	72	18
			Average	**87.5**	**89**	**7.9**	**61**	**17**

Table 3 Feature comparison of USE (Gogolla et al. 2007), Eclipse (Willink 2020), EMFtoCSP (Büttner & Cabot 2015), UMLtoCSPn (Cabot et al. 2007), CD2Alloy (Maoz et al. 2011), MaxUSE (Wu 2017), and OCL-Lite (Queralt et al. 2012). A tool is marked (✓) if it supports the feature as defined in the OCL 2.4 specification. (○) indicates support with significant limitations (e.g. support for only a subset of operations or requires manual search bounds). (✗) indicates missing feature(s) or an error.

	OCL Language Coverage						Analysis Capabilities			
Tool	**Quant**	**Coll**	**Navig**	**String**	**Typing**	**ClsMap**	**Consistency**	**Conflict**	**Counter**	**Scalability**
USE Model Validator	✓	✓	✓	○	✓	○	✓	✗	○	○
Eclipse OCL (Pivot)	✓	✓	✓	✓	✓	○	○	✗	○	○
EMFtoCSP	✓	✓	✓	✓	○	✓	✓	✗	✓	○
UMLtoCSP	✓	✓	✓	✗	○	✓	✓	✗	✓	○
CD2Alloy	✗	✗	✗	✗	✗	✗	○	✗	○	✗
MaxUSE	✓	✓	✓	✗	✓	○	✓	✓	✓	○
OCL-lite	○	○	✓	✗	○	✗	✓	✓	✓	✓
Our Framework	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓

Evaluation Criteria for Language Coverage:
Quant: Support for first-order quantifiers (`forAll`, `exists`, `one`). **Coll**: Built-in collection operations (`size`, `includes`, `select`, `collect`, etc.).
Navig: Property call resolution across single and multi-hop associations. **Typing**: Reflection and casting (`oclIsTypeOf`, `oclAsType`).
ClsMap: Support for `allInstances()` on classifiers.
Evaluation Criteria for Analysis Capabilities:
Consistency: Ability to find a valid instance (SAT). **Conflict**: Detection of unsatisfiable cores or contradictory invariants.
Counter: Generation of a concrete object diagram for SAT/UNSAT results. **Scalability**: Efficiency when increasing model population/scope.

diverse models. It evaluates features supported with correctness and performance. Related work on test generation for model transformations (Sen et al. 2009) shares the goal of automated test creation, but focuses on transformation rather than OCL constraint verification.

10. Conclusion and Future Work

We presented an automated framework for generating solver-verified OCL benchmarks. The framework combines pattern-based generation, conflicted constraints via mutation and SMT-based verification to produce benchmark suites that are diverse, non-redundant and computationally feasible. Across ten diverse UML/Ecore metamodels, the framework achieved an average generation efficiency of 89%, producing 50–100 verified constraints per metamodel. The resulting benchmarks expose concrete gaps in existing OCL tools, including weak support for string reasoning, nested quantifiers and conflict detection. These results suggest that the OCL community would benefit from more systematic and reproducible evaluation methods. Formally verified benchmark suites provide a clearer, evidence-based view of which language features are robustly supported and which remain challenging.

Future work will target three directions. First, we plan to extend the OCL 2.4 coverage to support type dispatch (`oclIsKindOf`, `oclAsType`), three-valued semantics with `invalid` propagation and `allInstances()` through bounded instance enumeration. Second, we will apply the framework to industrial-scale models and conduct a quantitative cross-tool study using Eclipse OCL and USE Validator. Third, we will investigate learning-based generation methods to further improve constraint diversity and domain relevance.

GenAI Disclosure & Usage This project acknowledges the use of Generative AI in its development and debugging. GenAI tools were utilized for code optimization, documentation assistance and refining the natural language patterns within the OCL generation engine.

References

Abukhalaf, S., Hamdaqa, M., & Khomh, F. (2023). On Codex prompt engineering for OCL generation: an empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)* (pp. 148–157).

Al Salem, H., & Song, J. (2019). A review on grammar-based fuzzing techniques. *International Journal of Computer Science & Security (IJCSS)*, *13*(3), 114–123.

Ba, J., & Rigger, M. (2023). Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2060–2071).

Bag, S., Kumar, S. K., & Tiwari, M. K. (2019). An efficient recommendation generation using relevant Jaccard similarity. *Information Sciences*, *483*, 53–64.

Büttner, F., & Cabot, J. (2015). Lightweight string reasoning in model finding. *Software & Systems Modeling*, *14*(1), 413–427.

Cabot, J., Clarisó, R., & Riera, D. (2007). UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (pp. 547–548).

Cabot, J., & Gogolla, M. (2012). Object constraint language (OCL): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems* (pp. 58–90). Springer.

Clavel, M., Egea, M., & de Dios, M. A. G. (2009). Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, *24*.

Dadeau, F., Fourneret, E., & Bouchelaghem, A. (2019). Temporal property patterns for model-based testing from UML/OCL. *Software & Systems Modeling*, *18*(2), 865–888.

Francisco, M. A., & Castro, L. M. (2012). Automatic generation of test models and properties from UML models with OCL constraints. In *Proceedings of the 12th Workshop on OCL and Textual Modelling* (p. 49–54). New York, NY, USA: Association for Computing Machinery.

Gogolla, M., Büttner, F., & Cabot, J. (2013). Initiating a benchmark for UML and OCL analysis tools. In *International Conference on Tests and Proofs* (pp. 115–132).

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, *69*(1-3), 27–34.

Gogolla, M., & Cabot, J. (2016). Continuing a benchmark for UML and OCL design and analysis tools. In *Federation of International Conferences on Software Technologies: Applications and Foundations* (pp. 289–302).

González, C. A., Büttner, F., Clarisó, R., & Cabot, J. (2012). EMFtoCSP: A tool for the lightweight verification of EMF models. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)* (pp. 44–50).

Järvisalo, M., Matsliah, A., Nordström, J., & Živnỳ, S. (2012). Relating proof complexity measures and practical hardness of SAT. In *International Conference on Principles and Practice of Constraint Programming* (pp. 316–331).

Jha, A., Monahan, R., & Wu, H. (2024). Verifying UML Models Annotated with OCL Strings. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (pp. 1106–1110).

Jha, A., Monahan, R., & Wu, H. (2025). A New Set of Metrics for Measuring the Complexity of OCL Expressions. In *Proceedings of the 24th international workshop on ocl and textual modeling (ocl 2025) at staf 2025.* Koblenz, Germany.

Lauria, M., Elffers, J., Nordström, J., & Vinyals, M. (2017). CNFgen: A generator of crafted benchmarks. In *International Conference on Theory and Applications of Satisfiability Testing* (pp. 464–473).

Maoz, S., Ringert, J. O., & Rumpe, B. (2011). CD2Alloy: Class diagrams analysis using Alloy revisited. In *International conference on model driven engineering languages and systems* (pp. 592–607).

Noten, J., Mengerink, J. G., & Serebrenik, A. (2017). A data set of OCL expressions on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (pp. 531–534).

Oberkampf, W. L., & Trucano, T. G. (2008). Verification and validation benchmarks. *Nuclear Engineering and Design*, *238*(3), 716–743.

Object Management Group (OMG). (2014, February). *Object Constraint Language (OCL), Version 2.4*. OMG Formal Specification. OMG. Retrieved from http://www.omg.org/spec/OCL/2.4/

Petrascu, V., & CHIOREAN, D. (2012). MDE-driven OCL Specification Patterns.

Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012). OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, *73*, 1–22.

Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., & Barrett, C. (2015). Counterexample-guided quantifier instantiation for synthesis in SMT. In *International Conference on Computer Aided Verification* (pp. 198–216).

Salemi, S., Selamat, A., & Penhaker, M. (2016). A model transformation framework to increase OCL usability. *Journal of King Saud University-Computer and Information Sciences*, *28*(1), 13–26.

Samimi, H., Warth, A., Eslamimehr, M., & Borning, A. (2015). Constraints as a design pattern. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (pp. 28–43).

Sen, S., Baudry, B., & Mottu, J.-M. (2009). Automatic model generation strategies for model transformation testing. In *International Conference on Theory and Practice of Model Transformations* (pp. 148–164).

Willink, E. (2020). Reflections on OCL 2.0. *J. Object Technol.*, *19*(3), 3–1.

Wu, H. (2017). MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *International Conference on Integrated Formal Methods* (pp. 348–356).

Wu, H. (2022). QMaxUSE: A query-based verification tool for UML class diagrams with OCL invariants. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 310–317).

Wu, H., & Timoney, J. (2020). Verifying OCL Operational Contracts via SMT-based Synthesising. In *MODELSWARD* (pp. 249–259).

About the authors

Ankit Jha is a PhD student at Maynooth University, Ireland. His research addresses the reliability of software systems through formal verification and building scalable evaluation frameworks.

Rosemary Monahan is a Professor in the Department of Computer Science and an affiliate of the Hamilton Institute at Maynooth University. Her research areas include formal specification languages, systems modelling, and software verification.

Hao Wu is an Assistant Professor in the Department of Computer Science at Maynooth University, Irealnd. His research focuses on creating automated software and tools for solving challenging problems in software engineering. Contact him at *haowu@cs.nuim.ie* and visit *https://classicwuhao.github.io*.