A New Set of Metrics for Measuring the Complexity of OCL Expressions

Ankit Jha[1,*,†], Rosemary Monahan[2] and Hao Wu[2]

[1]Department of Computer Science , Maynooth University, Maynooth, Kildare, Ireland

Abstract
The Object Constraint Language (OCL) is extensively used in model-driven engineering to specify constraints on UML class diagrams. Evaluating the complexity of OCL expressions remains a challenge, particularly in the context of verification. This paper introduces a complete new set of metrics for measuring OCL expressions. This gives us the advantage of measuring OCL in a customizable way. First, we enhance the Structural Complexity. Second, we propose two new types of metric: Computational and Dependency Complexity. Finally, we calculate the overall complexity using our Total Complexity metric. Our experimental results demonstrate how one can measure the complexity of an OCL benchmark in terms of its verification time which is used to define the weights for OCL constructs.

Keywords
OCL Complexity Metrics, SMT Solver Verification, Weighted Complexity Metrics, Total Complexity Measurement

1. Introduction

The Object Constraint Language (OCL) is a fundamental component of the Unified Modeling Language (UML), enabling the precise specification of constraints through invariants, pre-conditions and post-conditions on models. As system complexity increases,the complexity of OCL expressions also grows. This makes these expressions harder to understand, maintain and analyze. The increasing complexity of OCL expressions impacts not only the reliability and maintainability of model-driven systems but also the ability to systematically measure and calculate the complexity of existing OCL benchmarks[1]. An accurate complexity measurement is essential to ensure that OCL benchmarks reflect a range of difficulty levels aligned with user-defined requirements. This enables researchers and the OCL community to effectively evaluate OCL tools, including static analysis and verification techniques, under varying constraint complexities. A comprehensive approach to complexity measurement is essential to improve OCL-based systems. It also supports the generation of customizable OCL benchmarks tailored to specific user requirements.
Currently, research on OCL complexity metrics is limited. Many existing metrics such as cyclomatic complexity[2] and Halstead metrics [3] are designed for imperative programming languages and these are not be suitable for declarative languages such as OCL. Therefore, these metrics cannot be directly applied to measure the complexity of OCL expressions. Jordi Cabot, primarily defines OCL expression complexity based on the number of accessed objects, which may not fully capture computational or logical complexity [4]. It lacks empirical validation and does not establish a direct link between complexity values or understandability. This approach does not cover all OCL constructs and it mainly relies on worst/best-case assumptions which are not practically applicable on large sets of OCL expressions. Luis Reynoso focuses only on syntactical based metrics for OCL expressions, potentially overlooking other complexity factors such as logical depth and nested expressions [5]. This approach provides a theoretical validation but lacks empirical studies to justify its assumptions. Moreover, it lacks a comprehensive method for measuring OCL complexity.

OCL 2025: Workshop on OCL, June 11–14, 2025,University of Koblenz, Koblenz, Germany
Ankit.jha.2021@mumail.ie (A. Jha); Rosemary.Monahan@mu.ie (R. Monahan); haowu@cs.nuim.ie (H. Wu)
0009-0008-9159-9739 (A. Jha); 0000-0003-3886-4675 (R. Monahan); 0000-0001-5010-4746 (H. Wu)

This paper introduces a comprehensive method for measuring the complexity of OCL expressions by defining three new metrics: Structural Complexity, Computational Complexity and Dependency Complexity, each of them targeting a key dimension of OCL. We also introduce a Total Complexity metric that combines the structural, computational and dependency dimensions into a single score. This provides a systematic way of estimating the complexity of an OCL expression. A distinguishing feature of our approach is the incorporation of user-defined weights. This enables users to tailor complexity measurements to their specific needs. For example, they can prioritize the Computational Complexity when performance is the key, or focus on the Dependency Complexity in scenarios where maintainability is important. In the context of verification, we analyze the time required to verify OCL expressions using three different SMT solvers. This gives us insight into the computational overhead associated with different keywords and OCL constructs [6]. By measuring the verification time of various OCL constructs, our aim is to assess their impact on the overall complexity. This provides a concrete foundation for assigning weights to different OCL elements. As an example, we assign weights to different OCL constructs based on theirverification time. This helps users better understand how specific OCL constructs contribute to complexity.

2. Background

2.1. Complexity Metrics in Software Engineering

Two of the most commonly used metrics in industry are Cyclomatic Complexity (CC) and Halstead Metrics (HM). Cyclomatic Complexity measures the number of linearly independent paths through a program's control flow graph, making it particularly effective for imperative and procedural languages that rely on control structures such as loops and conditionals [2, 7]. However, OCL is a declarative language that lacks explicit control flow constructs, rendering Cyclomatic Complexity inapplicable to OCL expressions. On other hand, Halstead Metrics are based on the count of operators and operands in a program that focus on computational complexity and effort estimation for executable code [3, 8]. These metrics are designed for imperative programming languages where code execution is driven by sequences of operations and variable manipulations. Since OCL expressions specify constraints and do not define control flow, the fundamental concepts of operators and operands in Halstead's sense do not align well with OCL semantics[9, 4]. Structural metrics align with the declarative nature of the language and provide a meaningful foundation for analyzing OCL constraint complexity[10, 11].

2.2. Complexity in OCL Expressions

OCL expression complexity calculation is a complicated process that depends on many factors. These include nesting depth, operator usage, variable dependencies and computational overhead[1]. Traditional approaches for measuring thecomplexity of OCL expressions have primarily focused on structural aspects, such as expression length and the number of operations [5]. The interplay between syntax and semantics in OCL expressions adds additional complexity. While syntax influences the readability and maintainability, semantics impacts the correctness and performance [12]. A comprehensive complexity analysis was introduced in prior work, where the authors proposed the Overall Weighted Complexity (OWC) metric to assess the complexity of OCL expressions in a structured manner [1]. This metric measures complexity based on factors such as the nesting of collection operations, number of navigation, Boolean operations, attribute accesses, and other OCL constructs. The OWC metric gives higher weight to collection operations, assuming they significantly contribute to complexity without any empirical studies to justify this. However, other aspects such as recursion, nested logical expressions or constraint dependencies may also play a crucial role, but are not given a similar attention. Existing works propose OCL-specific complexity measures, but they often lack an integrated approach that considers multiple complexity dimensions, including computational complexity and verification time [1, 13, 14]. OCL expressions are often used in contexts such as large-scale systems with complex models and multiple classes. These factors can also introduce additional complexity [15]. Verification time, dependency relationships, and execution cost are important factors that need to be considered when analyzing the complexity of OCL expressions [16, 6, 17].

2.3. Verification of OCL Expressions Using SMT Solvers

Formal verification of OCL expressions is a critical step in ensuring the correctness and consistency of models, as it helps to detect and prevent inconsistencies that can arise from constraints [6, 18, 19]. Satisfiability Modulo Theory (SMT) solvers are widely used to verify OCL constraints by checking satisfiability conditions under different logical theories, such as linear arithmetic, bit vectors and arrays [20, 21, 22]. The computational cost of verification can vary significantly depending on the complexity of OCL expressions with certain operators and constructs requiring substantially more time for resolution [23]. For example, the nested quantifiers can significantly increase verification time while arithmetic operators may only slightly increase verification time [6]. Analyzing the verification time per OCL keyword and operator provides valuable insight into their complexity and helps define meaningful weight assignments for complexity assessment. In general, the verification of OCL expressions using SMT solvers is an active area of research with many opportunities to improve the efficiency and effectiveness of verification techniques [24, 25, 26].

3. Defining OCL metrics

To systematically evaluate the complexity of OCL expressions, we introduce a comprehensive set of complexity metrics that consider different aspects of OCL constraints. These metrics are categorized into three main groups: Structural Complexity, Computational Complexity and Dependency Complexity. Each category provides insights into different factors that contribute to the overall complexity of OCL expressions.

3.1. Structural Complexity

We design new metrics to improve effectiveness in complexity measurement and to avoid redundancy which affects the total complexity. In addition to new metrics within the dimension of Structural Complexity, we use existing metrics including Weighted Number of Operations(WNO) and Weighted Number of Messages(WNM) from [5]. Below are the list of new metrics which we considered based on Structural Complexity:

- **Number of Navigated Relationships Cardinality-Aware Navigations** (NNR-C Metric): This metric is designed to measure the complexity of OCL expressions by quantifying the number of relationships navigated while considering the impact of its cardinality (1..1 vs 1..$*$) on computational effort. Unlike basic navigation metrics that count relationships equally [5], NNR-C assigns higher weights to multi-valued (1..$*$) navigation because these relationships introduce iteration over collections, increasing verification and execution complexity. If an OCL expression accesses a single-valued (1..1) relationship, it is given a lower complexity weight, whereas multi-valued navigation (1..$*$) gives higher complexity due to the collection operations ($forAll, exists, select$). By incorporating cardinality-aware weighting, NNR-C provides a more precise estimation of navigation complexity in OCL expressions, making it particularly valuable for model verification, benchmarking and maintainability analysis.

- **Number of Class Navigation plus** (NNC+): this metric counts the number of distinct classes navigated in an OCL expression while combining it with a weighting system to reflect the navigation complexity more accurately. Unlike traditional metrics proposed by Reynoso [5], which treat all class navigation equally, NNC+ assigns higher weights to classes accessed through multiple intermediate associations (e.g self.membership.card) and lower weights to directly referenced classes (e.g self.customer). This approach reflects the added effort required to resolve deeper navigational paths and helps capture the structural complexity of OCL expressions more accurately. By distinguishing these navigation patterns, NNC+ provides a finer-grained view of class-level complexity in constraint evaluation.

- **Depth of Navigation Context aware** (DN-CA): This metric measures the depth of navigation in an OCL expression. It also considers the impact of collection operations, nested constraints, and implicit dependencies. Unlike traditional depth metrics that count only direct navigation chains, DN-CA assigns higher weights to navigations within collection-based expressions such as $forall$, *select* and *collect*. It also distinguishes between direct property accesses and deeply nested navigations. As a result, constraints that span multiple model elements or involve iterative operations receive a higher complexity score.

- **Weighted Navigation Chains** (WNC): This metric is designed to measure the complexity of OCL expressions by evaluating the depth and structure of the navigation chain within an expression. Unlike traditional navigation metrics that treat all navigation independently, WNC assigns higher weights to longer and more deeply nested navigation chains (e.g $self.a.b.c$ is considered more complex than $self.a.b$). The reason is that long dependencies introduces extra cost of de-referencing, verification complexity and execution effort, as deeper chains require more intermediate object resolutions. WNC also distinguishes between independent and dependent navigation. Navigation that is part of a continuous dependency chain (i.e long nested navigation chains) are assigned a higher weight than single navigations that do not follow such a chain. Additionally, when navigation is nested within collection operations ($forAll$, *select*, *collect*), their weights are multiplied to accurately represent the increased complexity introduced by iterative evaluation. By introducing chain depth, dependency structure and collection-aware weighting, the Weighted Navigation Chains (WNC) metric provides a more precise estimation of navigation complexity. This makes WNC valuable for analyzing the scalability and verification difficulty of OCL constraints.

- **Total Navigation Complexity** (TNC): We introduce the Total Navigation Complexity (TNC) metric to provide an accurate and redundancy-free evaluation of navigation based complexity in OCL expressions. TNC combines three metrics: NNR-C, DN-CA, and WNC. Each of these measure aspects such as navigation depth, collection multiplicity (e.g $1..*$) and navigational dependency. By combining these metrics, TNC avoids counting the same navigational structures multiple times. This ensures a more precise and accurate measure of complexity. If measured independently, these metrics will count the same navigational features multiple times, inflating the total complexity score. By combining them with carefully chosen weights, TNC provides a more balanced and non-redundant measure of navigation complexity in OCL constraints.
  The formula for TNC is defined as:

$$TNC = \alpha \cdot NNR\text{-}C + \beta \cdot WNC + \gamma \cdot DN\text{-}CA$$

  The choice of weights α, β and γ in the TNC metric is designed to provide users with a configurable approach that balances complexity contributions. Specifically, these weights allow users to balance the complexity arising from multi-valued relationships (NNR-C), sequential dependencies (WNC) and navigation depth (DN-CA). The main aim is to prevent redundancy in the complexity measurement. For example, the weight configuration $\alpha = 0.3$, $\beta = 0.4$, $\gamma = 0.3$ prioritizes sequential dependency complexity, making it suitable for models that have deeply nested navigation chains ($A \to B \to C \to D$) to contribute more to complexity than collection-based operations. This is particularly relevant in highly interconnected models where verification difficulty increases due to long dependency chains. Meanwhile, the configuration $\alpha = 0.4$, $\beta = 0.3$, $\gamma = 0.3$ prioritizes collection-based complexity and making it ideal for models where multi-valued ($1..*$) relationships significantly impact the verification. This can be seen in constraints that frequently iterate over large *collections* Operations. The weight γ (DN-CA) is designed to take depth complexity into account introduced by collection-based operations and nested navigations. The introduced weights are user-defined that allows customization based on model structure and user requirement. This ensures that TNC accurately reflects real-world complexity variations in OCL expressions.

- **Variable Reference Count** (VRC): This metric measures the number of distinct variables referenced within an OCL expression. A higher VRC indicates more variables used in an OCL expression, which introduces greater complexity as more variables require tracking and interpretation. The greater the number of variables introduced in an OCL expression, the more difficult the expression is to understand as this contributes to the complexity. An example which is given below.

  $$self.books- > forAll(b|b.author.name)$$

  here variable b is introduced by the $forAll$ operation. It is used multiple times within the body of the expression (for $b.author.name$), contributing to the VRC.

3.2. Computational Complexity

Computational Complexity in OCL refers to the effort required to verify or statically analyze an expression in the context of formal tools such as SMT solvers. OCL constructs such as the quantifiers $forAll$ and $exists$, significantly increase this complexity because they require repeated evaluations of logical conditions across all elements of a collection. We define three key metrics to quantify computational complexity: Operator Complexity (OC), Type Conversion Complexity (TCC) and Collection Iteration Complexity (CIC). For example, $forAll$ is weighted more than $select$ because it quantifies every element in a collection. These metrics aim to capture the constructs that most significantly affect solver performance and verification.

- **Operator Complexity** (OC): This metric measures the verification effort introduced by different operators within an OCL expression. This metric assigns user-defined weights to various operator categories based on their impact on SMT solver performance, as observed in our experimental results 4. From a verification perspective, arithmetic operators such as $+, -, *, /$ are generally less weighted than logical operators such as and, or, not and quantifiers such as $forAll$ and $exists$. These require condition evaluation over collections which significantly increases verification time. By distinguishing these categories, OC enables users to prioritize constructs that have the greatest impact on the verification effort. This approach provides a customizable and verification aware complexity measurement. It is particularly useful in scenarios where verification performance and solver efficiency are critical considerations.

- **Type Conversion Complexity** (TCC): This metric measures the computational overhead introduced by explicit and implicit type conversions in OCL expressions. Operations such as $asSet()$, $flatten()$ and $oclAsType()$ require additional computation time, when converting large collections, sorting elements, or enforcing type constraints. Simple conversions like $oclAsType(Integer)$ introduce relatively low complexity, while nested conversions—such as:

  $$self.items- > collect(i|i.values)- > flatten()- > asSet()$$

  substantially increase verification effort due to restructuring collection elements. TCC assigns higher weights to type conversions that significantly impact verification performance. This allows users to identify and optimize costly or redundant conversions within OCL expressions.

- **Collection Iteration Complexity** (CIC): This metric measures the computational cost associated with iterating over collections in OCL expressions. Higher-order quantifiers and collection operations such as *forAll*, *exists*, *select* and *collect* introduce additional overhead, especially when nested iterations occur, leading to greater complexity. CIC assigns greater weight to deeply nested structures, as they increase verification time.

3.3. Dependency Complexity

Dependency Complexity measures the degree of interdependence between model elements referenced within an OCL expression, assessing how constraints depends on previously defined invariants, derived attributes and transitive model relationships. High dependency complexity increases maintenance effort, as changes in one model element may impact multiple constraints and reduced modularity. To systematically capture dependency complexity, we introduce the following metrics:

- **Reused Constraint Count** (RUC): This metric counts the number of reused constraints within an OCL expression. This reflects the expression's dependency on previously defined elements. Measuring RUC helps to reveal how much an OCL expression depends on other constraint. This makes it easier to spot a constraints that may need extra attention during maintenance or updates.

4. Experiments & Results

Experiment Objective: The main objective of this experiment is to assess the complexity of OCL expressions by observing the time taken to verify them using SMT solvers. Rather than relying on traditional runtime complexity, we focus on verification complexity as a more suitable measure for OCL expressions. Mapping an OCL expression to a logical formula requires not only familiarity with OCL constructs but also a deep understanding of its underlying semantics. Hence, we believe that verification provides a more accurate reflection of the complexity of OCL expressions. It captures both structural and semantic challenges that may not be visible through execution alone. Using SMT solvers for verification of OCL constraints is a standardized practice. By analyzing verification times with SMT solvers across simple and complex expressions, we systematically measure how different OCL constructs and operators impact the overall complexity.

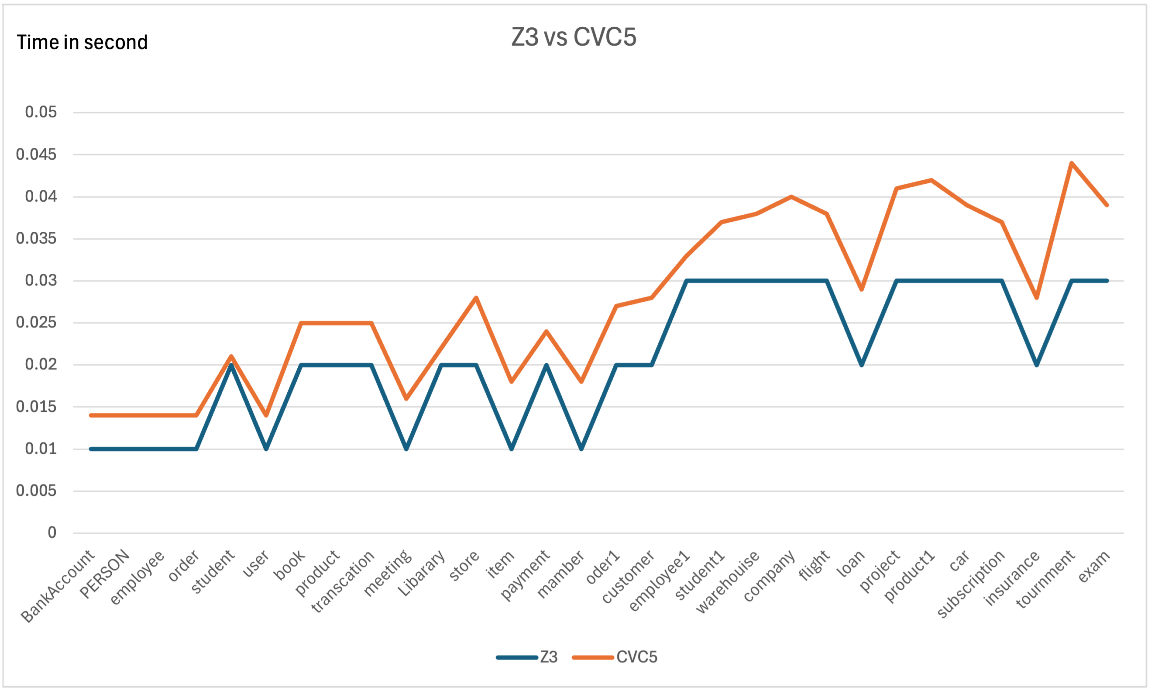
Experimental Setup: We run our experiment on an Apple Macbook air with (M1 chip) 8 GB memory. The SMT solvers used in our experiment are Z3 version: 4.14.0 [27], CVC5 version: 1.2.1 [28] and Ostrich version:1.3 [29]. We choose these SMT solvers mainly because of their popularity, performance, specialization and usability [6].

We formed a benchmark that has a total of 30 OCL expressions. 12 of them are collected from existing literature [30, 6, 15, 11], and 18 expressions are manually designed to ensurevariety in the OCL constructs used. We categorize these expressions into simple (15) and complex constraints (15) as follows.

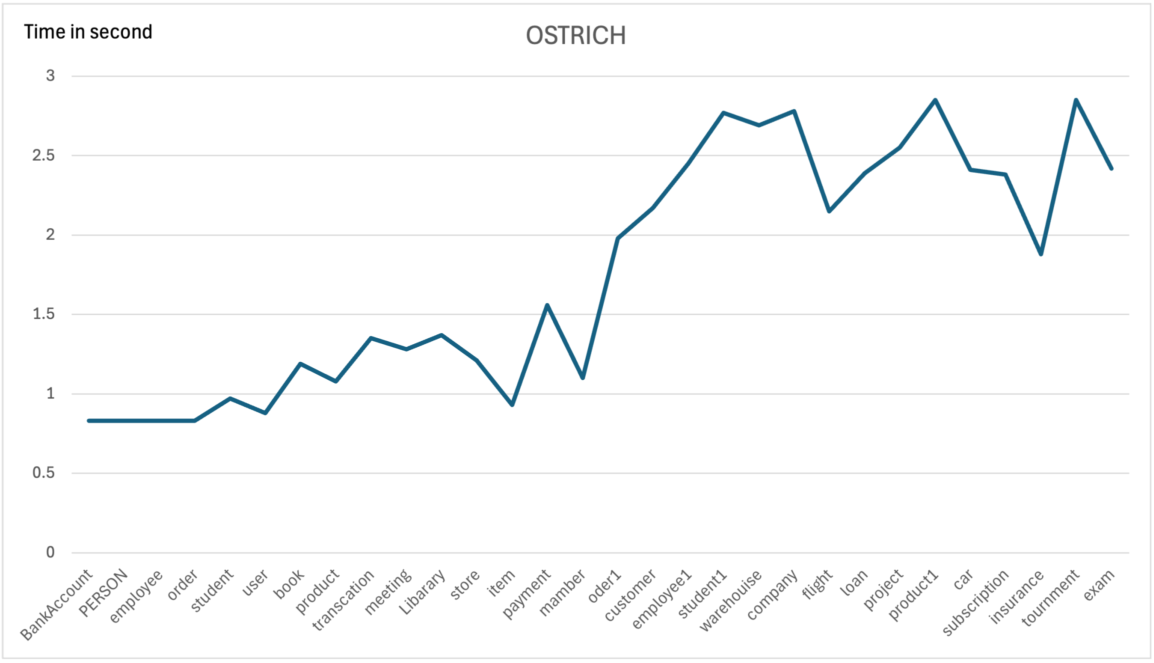
- **Simple:** Expressions with lower computational operator and fewer navigations.
- **Complex:** Expressions that involve deep navigation, multiple dependencies, extensive use of arithmetic, logical operators and collection operations.

We translate each OCL expression into an equivalent SMT formula and measure its verification time individually. Figure 1 shows the verification time trends across different SMT solvers, reflecting the underlying complexity of the OCL expressions. To clearly demonstrate how complexity impacts verification performance, we show results for representation of simple (From *BankAccount* to *member*) and complex (From *order*1 to *exam*) expressions illustrating the variation in solver behavior across different complexity levels.

Result: The experimental results indicate that all three SMT solvers exhibit a similar verification time trend, where simple OCL expressions are consistently verified faster than complex ones. Across all



(a) Verification time comparison between Z3 and CVC5 in seconds



(b) Verification time of Ostrich in seconds

Figure 1: Verification Time Comparison of SMT Solver

solvers, simple expressions, which mainly involve direct attribute comparisons and minimal navigation can be verified quickly (Time < 0.25 seconds). On other hand, complex expressions which are characterized by extensive use of collection operators ($forAll, exists, select$) and deep navigation, require significantly more time to verify (0.25 < Time < 0.45). This trend highlights that OCL constraints with a higher dependency on collection-based operations and deeply nested navigation introduce greater computational overhead. As a result, these expressions lead to increased solver verification time. Although the solvers may differ in absolute runtime, their shared similar trend confirms that verifying complex expressions demands more resources and run time than verifying simple expressions.

5. Total Complexity

In this section, we present an experiment to calculate the total complexity of OCL expressions using a combination of different metrics define above. This approach helps us capture various aspects of complexity in a single consolidated score. The Total Complexity (TC) metric is calculated by adding structural complexity, computational complexity and dependency complexity, ensuring a comprehensive assessment of OCL expressions. The goal of this experiment is to measure the total complexity of

Keyword	Weight	Reasons
= , <> , > , < , >= , <=	1.0	Simple operators, lightweight in verification.
and, or, not	1.5	Logical operators require evaluating multiple conditions, increasing complexity slightly.
implies, xor	2	These operators introduce logical dependencies, requiring additional verification steps to ensure correctness.
select, collect, reject, flatten	2	These functions involve traversing collections, which adds iteration complexity.
forAll, exists, closure, iterate	3	Higher complexity due to nested iteration, especially when applied to large collections.
oclAsType()	2	Type conversion; introduces validation overhead in type checking.
asSet	1.5	Converts a collection to a set, removing duplicates; requires a lightweight operation.

Table 1
Assigned weights for key Object Constraint Language (OCL) constructs based on their impact on verification.

OCL expressions by accumulating defined metrics. To support this evaluation, we assign weights to OCL keywords based on their impact on verification time as observed in our empirical analysis in Section 4. Table 1 summarizes the assigned weights. We define a lower bound of 1.0 for simple operators (e.g. $=, <, >=, <=, >$) that introduce minimal verification effort. It is serving as a baseline for the comparison. A maximum weight of 3.0 is assigned to constructs like $forAll$, *exists*. Intermediate weights (e.g. 1.5 or 2.0) are used for constructs that contribute a moderate verification effort such as logical operators (AND, OR) or type conversions ($oclAsType()$). The weights are placed relatively closer (generally in increments of 0.5), to capture minor but meaningful differences between constructs. This ensures that the metric remains sensitive to structural variations without disproportionately affecting the overall complexity score. Our proposed approach remains flexible, allowing users to adjust weights for domain-specific use cases or performance sensitive systems.

5.1. Discussion

The proposed complexity metrics make a significant step forward in the systematic evaluation of OCL constraints. Unlike previous work where weights were assigned without clear justification [1], we use verification time as an indicator for assigning weights to OCL constructs. The underlying assumption is that constructs with lower verification effort should receive lower weights than those with higher complexity. This establishes each metric in a concrete and quantifiable way. We also give a well structured methodology for evaluating OCL complexity across syntactic, semantic and computational dimensions. For example, the Total Navigation Complexity (TNC) captures depth and redundancy in navigation chains, while Operator Complexity (OC) reflects the overhead introduced by logical constructs such as $forAll$, *exists*, and *implies*. The flexibility to apply user-defined weights further enhances the applicability of these metrics, allowing users to prioritize constructs based on performance, maintainability or domain-specific constraints. More importantly, this work lays the foundation for automatic OCL benchmark generation. Our ultimate goal is not just to measure existing expressions but to generate OCL benchmarks with customization of complexity based on user-defined criteria. This will offer the community not only a better way to measure, test and analyze existing

OCL tools but also a powerful technique for generating configurable benchmarks. We believe that the metrics defined here can help us to better understand the necessities of OCL benchmark generation.

6. Conclusion

This paper presents a robust verification-driven approach for assessing OCL complexity across structural, computational and dependency dimensions. We believe that the proposed metrics can help OCL community to better understand the capabilities and limitations of existing OCL tools. In the future, we plan to investigate an approach that can generate OCL benchmark satisfying metrics defined with customizable weights.

References

[1] M. Gogolla, T. Stüber, Metrics for ocl expressions: development, realization, and applications for validation, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2020, pp. 1–10.

[2] C. Ebert, J. Cain, G. Antoniol, S. Counsell, P. Laplante, Cyclomatic complexity, IEEE software 33 (2016) 27–29.

[3] T. Hariprasad, G. Vidhyagaran, K. Seenu, C. Thirumalai, Software complexity analysis using halstead metrics, in: 2017 international conference on trends in electronics and informatics (ICEI), IEEE, 2017, pp. 1109–1113.

[4] J. Cabot, E. Teniente, A metric for measuring the complexity of ocl expressions, in: Model Size Metrics Workshop co-located with MODELS, volume 6, Citeseer, 2006, p. 10.

[5] L. Reynoso, M. Genero, M. Piattini, Measuring ocl expressions: a "tracing"-based approach, Proceedings of QAOOSE 2003 (2003).

[6] A. Jha, R. Monahan, H. Wu, Verifying uml models annotated with ocl strings, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 1106–1110.

[7] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.

[8] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

[9] J. B. Warmer, A. G. Kleppe, The object constraint language: getting your models ready for MDA, Addison-Wesley Professional, 2003.

[10] S. Serbout, C. Pautasso, Apistic: A large collection of openapi metrics, in: Proceedings of the 21st International Conference on Mining Software Repositories, MSR '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 265–277. URL: https://doi.org/10.1145/3643991.3644932. doi:10.1145/3643991.3644932.

[11] L. Reynoso, M. Genero, M. Piattini, Measuring ocl expressions: An approach based on cognitive techniques, Metrics for Software Conceptual Models (2005) 161–206.

[12] M. Gogolla, F. Büttner, M. Richters, Use: A uml-based specification environment for validating uml and ocl, Science of Computer Programming 69 (2007) 27–34.

[13] A. Shaikh, R. Clarisó, U. K. Wiil, N. Memon, Verification-driven slicing of uml/ocl models, in: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, ASE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 185–194. URL: https://doi.org/10.1145/1858996.1859038. doi:10.1145/1858996.1859038.

[14] T. Y. Kim, Y. K. Kim, H. S. Chae, Towards improving ocl-based descriptions of software metrics, in: 2009 33rd Annual IEEE International Computer Software and Applications Conference, volume 1, 2009, pp. 172–179. doi:10.1109/COMPSAC.2009.32.

[15] L. Reynoso, M. Genero, M. Piattini, E. Manso, The effect of coupling on understanding and modifying ocl expressions: An experimental analysis, IEEE Latin America Transactions 4 (2006) 130–135.

[16] L. Reynoso, M. Genero, M. Piattini, Validating metrics for ocl expressions expressed within uml/ocl models, in: International Workshop on Software Audit and Metrics, volume 2, SCITEPRESS, 2004, pp. 59–68.

[17] H. Wu, Qmaxuse: A new tool for verifying uml class diagrams and ocl invariants, Science of Computer Programming 228 (2023) 102955.

[18] P. Yang, L. Zhang, Q. Li, X. Gao, Y. Yang, Oclverifer: Automated verification of ocl contracts in requirements models, Science of Computer Programming 240 (2025) 103197.

[19] H. Wu, M. Farrell, A formal approach to finding inconsistencies in a metamodel, Software and Systems Modeling 20 (2021) 1271–1298.

[20] H. Wu, J. Timoney, Verifying ocl operational contracts via smt-based synthesising., in: MODELSWARD, 2020, pp. 249–259.

[21] M. Soeken, R. Wille, R. Drechsler, Encoding ocl data types for sat-based verification of uml/ocl models, in: International Conference on Tests and Proofs, Springer, 2011, pp. 152–170.

[22] N. Przigoda, M. Soeken, R. Wille, R. Drechsler, Verifying the structure and behavior in uml/ocl models using satisfiability solvers, IET Cyber-Physical Systems: Theory & Applications 1 (2016) 49–59.

[23] A. D. Brucker, B. Wolff, Hol-ocl: a formal proof environment for uml/ocl, in: Fundamental Approaches to Software Engineering: 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 11, Springer, 2008, pp. 97–100.

[24] H. Wu, Qmaxuse: A query-based verification tool for uml class diagrams with ocl invariants, in: International Conference on Fundamental Approaches to Software Engineering, Springer International Publishing Cham, 2022, pp. 310–317.

[25] H. Wu, A query-based approach for verifying uml class diagrams with ocl invariants., The Journal of Object Technology 21 (2022) 3.

[26] C. A. González, J. Cabot, Formal verification of static software models in mde: A systematic review, Information and Software Technology 56 (2014) 821–838.

[27] N. Bjørner, L. Nachmanson, Navigating the universe of z3 theory solvers, in: Formal Methods: Foundations and Applications: 23rd Brazilian Symposium, SBMF 2020, Ouro Preto, Brazil, November 25–27, 2020, Proceedings 23, Springer, 2020, pp. 8–24.

[28] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al., cvc5: A versatile and industrial-strength smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2022, pp. 415–442.

[29] B. Eriksson, A. Stjerna, R. De Masellis, P. Rüemmer, A. Sabelfeld, Black ostrich: Web application scanning with string solvers, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 549–563.

[30] A. Jha, Automatic benchmark generation for object constraint language, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2023, pp. 486–488.