Verifying UML Models Annotated with OCL Strings

Ankit Jha
Ankit.Jha.2021@mumail.ie
Maynooth University
Maynooth, Kildare, Ireland

Rosemary Monahan
Rosemary.Monahan@mu.ie
Maynooth University
Maynooth, Kildare, Ireland

Hao Wu
haowu@cs.nuim.ie
Maynooth University
Maynooth, Kildare, Ireland

ABSTRACT

The Object Constraint Language (OCL) is a specification language that allows users to write precise constraints or rules over models that are built using the Unified Modeling Language (UML). Many OCL constraints used in real-world models specify a set of rules over string data types. This makes reasoning about UML models that are annotated with OCL string constraints very challenging. In this short paper, we demonstrate the feasibility of using Satisfiability Modulo Theories (SMT) solvers for verifying OCL string-type constraints. Specifically, we compare the string reasoning capabilities of three SMT solvers in terms of their usability, performance, and the diversity of instances generated. We believe that the Model-Driven Engineering (MDE) community can benefit from our preliminary results in identifying the strength and limitations of the state-of-the-art SMT solvers for OCL string verification.

KEYWORDS

OCL, SMT solver, String Constraints

ACM Reference Format:
Ankit Jha, Rosemary Monahan, and Hao Wu. 2024. Verifying UML Models Annotated with OCL Strings. In *Proceedings of Model Driven Engineering, Verification and Validation (MODELS '24)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 MOTIVATION

The Unified Modeling Language (UML) is widely used in building different types of software models. It is the central element for Model Driven Engineering (MDE). The graphical notations of UML [17] not only allow software engineers to model different aspects of a complex system but also provide ways of communications among team members. Meanwhile, the Object Constraint Language (OCL) is a formal language that allows users to specify rules over UML models [6, 14, 20]. When a UML model is annotated with OCL constraints, the model possesses a high level of abstraction and precision for describing system properties and behaviors. Among different types of constraints, OCL string constraints are particularly important for writing and specifying rules associated with strings. Many business rules or system properties can be captured using OCL string constraints. For example, specifying that a username or password must follow a specific pattern can be efficiently done with OCL string constraints. These constraints are extremely helpful in defining such requirements. However, their complexity and expressiveness poses a big challenge to verification. Recently, efforts were made by [5] using an automated approach grounded in Constraint Programming to verify UML models annotated with OCL constraints including string. Simulink/Stateflow, with the aid of Simulink Design Verifier, has been used to verify the UML model of an automatic train control system [12]. An approach to manually translate natural language requirements into an enhanced UML model is proposed in [9] later which is translated and verified by means of the NuSMV model checker. These approaches struggle with scalability and complexity, are often domain-specific, require high manual effort, and can be prone to translation errors while requiring significant expertise.

With recent advances in Satisfiability Modulo Theories (SMT) solvers for solving string constraint [15], it is now possible to translate OCL string constraints directly to string functions for SMT solver. To formally verify a UML model that is annotated with OCL string constraints, we aim to generate an *instance* of that model that satisfy all possible OCL string constraints. In this paper, we investigate such translation and compare different SMT solvers in solving OCL string constraints. In particular, we report our findings and list the best SMT solver for verifying UML models are annotated with OCL string constraints. In short, our contributions of this paper can be summarised as follows:

- We define a mapping between common OCL string operations to SMT encodings.
- We performed a detailed comparison of 3 SMT (string) solvers to show their strengths and limitations in verifying OCL string data types.

2 BACKGROUND

2.1 OCL String

OCL allows users to define constraints and rules. Among OCL's arsenal of features, the OCL String [19] stands out as an important feature for modeling. String-based OCL constraints are widely used in several real-world example such as password, email, etc. Therefore, it is important to ensure their consistency. As a fundamental type within OCL, the OCL String represents a sequence of characters, providing a flexible and versatile method for expressing textual information. The introduction of the OCL String data type addresses several significant needs in modeling and constraint specification [13]. String manipulation operations in OCL allow us to concatenate, split, and match strings. This is essential when dealing with complex data transformations or when constructing strings based on certain conditions. It enhances expressiveness by allowing the specification of constraints concerning textual data, essential for modeling real-world entities and operations involving

MODELS '24, 22-27,September, Linz, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

text-based attributes such as usernames or messages. It facilitates enhanced verification by allowing constraints to validate object states at runtime or design time, ensuring fidelity to specified requirements such as non-empty strings or strings with predefined lengths [4].

2.2 SMT Solver

In recent years we have seen a abundance of research on string constraints, particularly SMT solvers demonstrate promising results in the area [1, 3, 16, 23]. In the context of the Object Constraint Language (OCL), string-based operations play a crucial role in expressing constraints and specifying conditions related to string manipulation [21]. However, there is still a gap in understanding and identifying the most appropriate SMT solver for solving string based constraints annotated with quantifiers efficiently. For this research, we initially selected five SMT solvers: z3-str [15], z3-noodler [8], z3 [10], cvc5 [2] and ostrich [7]. These SMT solvers were selected based on popularity, performance, specialization and usability. After careful investigation, we decided to eliminate z3-str and z3-noodler. This is because they have poor performance with quantified formulas and often fail to finish verification within acceptable time frame. Z3 provides a powerful string reasoning engine that allows users to reason about operations over strings such as concatenation, string matching and substring extraction. This is useful in detecting vulnerabilities in programs that handle user input, ensuring robustness and reliability [11]. On the other hand, CVC5 is capable of reasoning about word equations with code that is possible to efficiently convert between strings and integers [18]. Ostrich is a robust string solver known for its backwards propagation technique. It is based on the idea of computing pre-images of regular expression constraints. Its decision procedure is designed for straight-line formulas.

3 FROM OCL STRING CONSTRAINTS TO SMT FORMULAS

In this Section, we present how to map each OCL string operation to a SMT formula that can be verified/solved by any SMT solver that supports SMTLIB 2.0. In fact, the mapping is straightforward since many string operations are directly supported by SMT solvers. Table 2 summarises the mapping from OCL string operations to corresponding SMT string functions. With these mappings, we now can translate OCL constraints into SMT formulas.

String Operation	SMT String Function
Concatenation	($str1 \wedge str2$)
size	(len (str))
Index Of	(indexof ($str1$ $start\_length$))
toUpperCase	($str1$)$\wedge re.*$ ($re.range('A','Z')$)
toLowerCase	($str1$)$\wedge re.*$ ($re.range('a','z')$)

Table 2: Mapping table for translating OCL string operations to SMT string functions.

Now, we use a sample UML model shown in Figure 2 to illustrate detailed mappings. This model has one class called *Person* along with 9 OCL invariants, as listed below:

1. Valid First Name: Size of the first name is greater than 5 characters.
2. Valid First Name Character: First Name should be combination of lowercase characters a-z.
3. Valid Last Name: Size of the last name is greater than 5 characters.
4. Valid Last Name Character: Last Name should be combination lowercase characters a-z and uppercase characters A-Z.
5. Valid Email Address: Email should follow a pattern of first name "." last name "@gmail.com"
6. Valid ZIPcode Pattern: ZIPcode should be 5 character in length and should be a combination of uppercase characters A-Z and digits 0-9.
7. Valid Area Code: First 2 characters of ZIPcode.
8. Valid Street Code: Last 2 characters of ZIPcode.
9. Valid Account Number: A number between 10000 and 99999.

We now take invariants (1) and (2) as our mapping example. In order to map the two invariants to SMT formulas, we first introduce two additional functions (1) $O_{\text{id}} : INT \rightarrow INT$ indicates the identification of an object in memory and (2) $T_{\text{Person}} : INT \rightarrow BOOL$ takes an identification of an object and returns whether this object is a type of *Person*. This follows the encoding proposed in [22]. In general, each invariant is mapped to a first-order formula. For example, invariant (1) can be mapped to the following SMT formula with a string function $str.len$ to constrain the length of a string:

$$\forall x \in \text{Int}(\text{Person}(O_{\text{id}}(x)) \rightarrow (\text{str.len}(\text{first\_name}(O_{\text{id}}(x))) \geq 5))$$

Invariant (2) involves a constraint stating a combination of lowercase character a and z. This can be directly mapped to the string function $re.range$. Hence, the final formula for invariant (2) is as follows:

$$\forall x \in \text{Int}\,(\text{Person}(O_{\text{id}}(x)) \rightarrow (\text{str.in.re}(\text{first\_name}(O_{\text{id}}(x)) \wedge$$
$$\text{re.}^{*}(\text{re.range}('a','z')))))$$

Here $re.range('a','z')$ gives a regular expression matching any single character from a to z.

Let's take invariants (3), (4) and (8) as further examples. These invariants use string operations *UpperCase*, *LowerCase* and *Substring*. With help of string functions in SMT, we can map them to the following formulas:

$$\forall x \in \text{Int}\,(\text{Person}(O_{\text{id}}(x)) \rightarrow (\text{str.len}(\text{last\_name}(O_{\text{id}}(x))) \geq 5))$$

$$\forall x \in \text{Int}(\text{Person}(O_{\text{id}}(x)) \rightarrow \text{str.in.re}(\text{last\_name}(O_{\text{id}}(x)) \wedge$$
$$\text{re.}^{*}(\text{re.union}(\text{re.range}('a','z') \wedge \text{re.range}('A','Z')))))$$

$$\forall x \in \text{Int}(\text{Person}(O_{\text{id}}(x)) \rightarrow \text{AreaCode}(\text{str.at}(\text{ZIPCode}(O_{\text{id}}(x)), 4)))$$

The formulas above represent that for every person, their last_name must be longer than 5 characters (implemented with $str.len$). Here $re.range('a','z')$ gives a regular expression matching any single character from 'a' to 'z' and $re.range('A','Z')$ gives a regular expression matching any single character from 'A' to 'Z', which is

Model Name	OCL String Operations								
	Size	Concate	Substring	toInteger	toReal	toUpperCase	toLowerCase	IndexOf	toBoolean
Account	✓	✓				✓	✓		
Research	✓					✓	✓		✓
Animal	✓					✓	✓	✓	✓
Atom	✓		✓			✓	✓	✓	✓
CivilStatus	✓	✓	✓			✓	✓		✓
eShop	✓	✓	✓			✓	✓		✓
Person	✓	✓	✓			✓	✓	✓	✓
Sales	✓		✓			✓	✓		
Partnership	✓	✓	✓			✓	✓		✓
Student	✓	✓				✓	✓		

Table 1: Our benchmark consists of 10 UML class diagrams with a total of 55 OCL string invariants. Each class diagram uses different OCL string operations.

Model	Z3				CVC5				Ostrich			
	1	5	10	15	1	5	10	15	1	5	10	15
Account	0.17	0.78	3.53	10.42	0.30	433	time out	time out	0.78	21.05	time out	time out
Research	0.55	1.53	4.50	19.25	2.13	time out	time out	time out	0.81	19.68	time out	time out
Animal	0.16	0.37	1.14	2.21	0.26	1.26	317.81	time out	0.63	480	time out	time out
Atom	0.06	0.11	0.38	0.91	0.17	1.14	45.45	197.51	0.83	19.28	time out	time out
CivilStatus	0.07	0.13	0.49	0.78	0.29	32.42	time out	time out	1.84	123.17	time out	time out
eShop	0.16	0.30	1.02	2.06	0.50	5.08	time out	time out	2.42	119.10	time out	time out
Person	0.11	0.51	1.59	4.50	0.44	time out	time out	time out	2.55	284	time out	time out
Sales	0.13	0.76	2.53	17.72	2.24	502	time out	time out	1.53	time out	time out	time out
Patnership	0.38	4.72	18.10	174.82	0.30	136	time out	time out	1.97	583	time out	time out
Student	0.07	0.18	0.57	1.12	0.59	330	time out	time out	1.64	178	time out	time out

Table 3: Time taken in seconds by Z3, CVC5, Ostrich to generate 1,5,10,15 instances for our benchmark with a timeout of 600 seconds.

held with union between them. The area code is determined by the help of *str.at* that performs substring operation.

4 EXPERIMENTS & RESULT

Benchmark: To evaluate the efficiency of each SMT solver, we have created 10 models and added 55 OCL string invariants (constraints) as our benchmark. Table 1 lists our benchmark that contains different OCL string operations. For each model in this benchmark, we add different string operations such as *Concatenation*, *LowerCase*, *UpperCase*, *Substring* and *Size*. We did not cover *toInteger* and *toReal* string operations because there is no direct function support to convert a string into an *Integer* or *Real*. Our main reason for including these operations is (1) to cover all OCL string operations as many as possible and (2) to impose a significant challenge on the solvers. This ensures a comprehensive evaluation of each solver's capabilities in handling complex and diverse string constraints. For instance, some string constraints might involve concatenating two strings, while others ensures that the resulting string contains a specific uppercase character. Another type of challenging string constraint involves the length of a string must fall within a predetermined range and contain a specific pattern of uppercase/lower case.

Setup: To compare different SMT solvers, we run our experiment on an Apple Macbook air with specification as an Apple M1 chip, an 8-core GPU, and 8 GB memory. The SMT solvers used in our experiment are Z3 version: 4.12.2, CVC5 version:1.0.5 and Ostrich version:1.3. We have written a simple python script that can automatically produce a set of first-order formulas from OCL string operations (as mentioned in Table2).

Experiment: The experiment is performed on our benchmark shown in Table 1. Our aim is to determine not only if the solver can find a solution but also to measure the time required to solve these constraints. We ask each solver (with a timeout of 600 seconds) to generate 1, 5, 10 and 15 instances for each model, and we then record the time taken by each solver.

Results: The results shown in Table 3 demonstrate the efficiency and effectiveness of each solver in successfully finding solutions. Overall, Z3 excelled consistently by solving all string constraints without timing out, showcasing its efficiency and reliability. CVC5 manages to generate a smaller number of instances but often timed out on larger ones. Ostrich frequently timed out on larger number of instances, performing well enough on simpler models but struggling with more complex string constraints. All solvers encounter significant challenges with complex models like *Account*, *Sales*, and *Partnership*, especially for larger number of instances. CVC5 and Ostrich frequently timeout due to the intricate string operations and constraints, indicating scalability issues and inefficiencies in handling extensive string manipulations and higher computational complexity. Based on the experimental results, Z3 has the best performance, consistently handling all instance sizes without timing out. It proves to be the most capable and efficient in processing OCL string operations within the allocated time frame. This makes Z3 a reliable choice for verifying UML models annotated with OCL string constraints. For the detailed results and encodings, please see the GitHub Repository.[1]

[1] https://github.com/AnkitMU/OCL

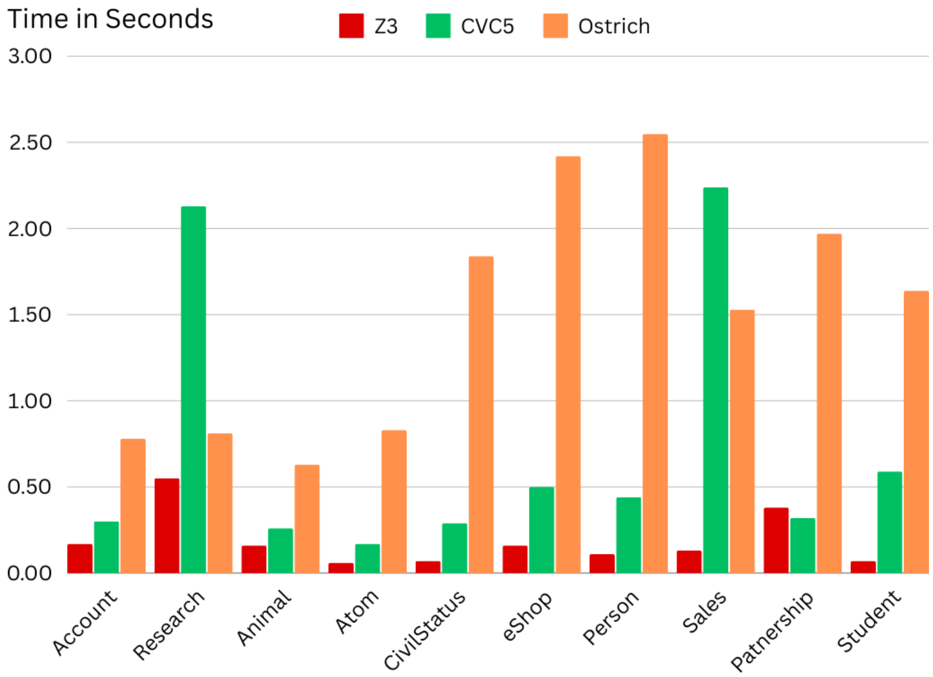
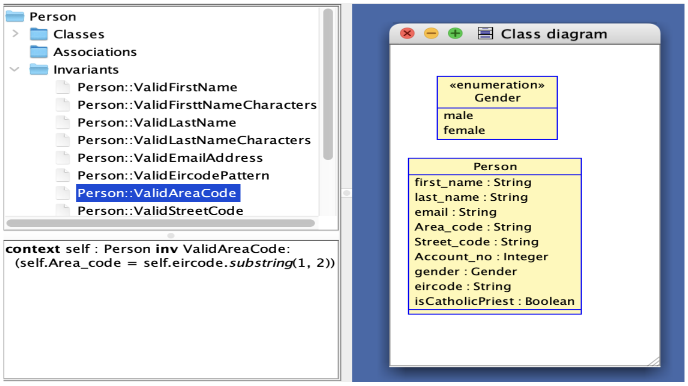


Figure 1: Comparison of Time(Seconds) taken by Z3, CVC5, Ostrich to generate 1 instance for our benchmark.



```
context self : Person inv ValidAreaCode:
  (self.Area_code = self.eircode.substring(1, 2))
```

Figure 2: An example of a class diagram annotated with 9 OCL string invariants.

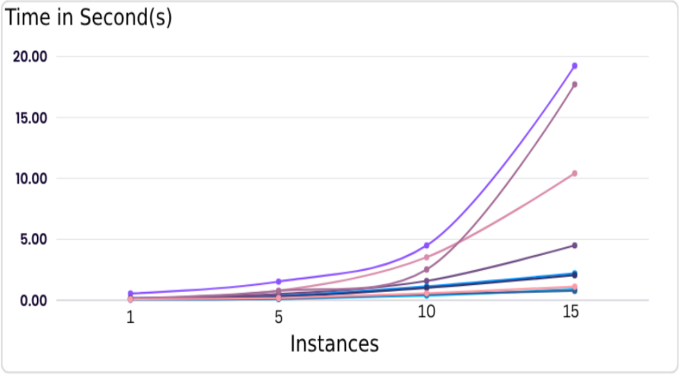


Figure 3: Performance comparison of Z3 to generate 1, 5, 10 and 15 instances for given class diagram models

5 DISCUSSION

Performance: Figure 1 plots the time taken by Z3 for generating 1 instances. Our analysis suggests that on average, Z3 is 200x more efficient than CVC5 and upto 4500x more efficient than Ostrich. The time taken to solve constraints using SMT solvers is significantly influenced by the number of quantifiers in a formula. In general, the nested quantifiers cause more challenging for the solvers.
From Figure 3 we conclude that the time required to generate models for 5, 10, and 15 instances increases significantly. This is mainly due to generating unique instances. This means each instance generated from solvers must be different from previous ones.
Usability: In term of popularity and usability, we conclude that documentation for Z3 and CVC5 are comprehensive, providing detailed user guides, API references and number of tutorials that cover a wide range of use cases including support for multiple programming languages such as Python, C++, and Java. Users can easily integrate Z3 into their own projects. The active community around Z3 and CVC5 is a significant advantage, with numerous discussions and support available on platforms like GitHub, Stack Overflow, and specialized forums. Compare to Z3 and CVC5, Ostrich has a moderate amount of documentation, mostly from research initiatives and academic papers. Although its API documentation is not as comprehensive as those of its more well-known peers, its speciality in string constraint solving makes it remain as a good

solver. Despite its small community and documentation. In case of ostrich, we had to contact specific developers to get detailed technical details in order to retrieve models from solver.

Diversity: By diversity, we refer to the varieties of instances generated by the SMT solvers. This simply means that different characters in a string. We believe this is an important character. This is because (1) it is useful to use these strings as test cases for testing OCL static analysers or parsers. (2) it is easier to further generate meaningful strings for the real-world scenarios. For example, a string “jack” is much meaningful than a string “aaaa” in the context of generating person’s name. When generating diverse strings we notice that Z3 produces more interesting strings than CVC5 and Ostrich. For instance, when generating strings for a person’s name for the first time, Z3 outputs "lhphps," whereas CVC5 and Ostrich both produce "aaaaaa." For the second time, Z3 provides "hsPsxp," demonstrating a greater variety within the specified constraints. In contrast, CVC5 and Ostrich only yield "aaaaab." Although the strings produced by the three solvers satisfy defined OCL invariants, the strings generated by Z3 are much more helpful if they are considered to be test cases for string testing. Further, it is easier for users to tune these string into actual meaningful strings in a particular context by adding additional context constraints.

Lesson learned:

From our analysis and experimentation, several key insights emerged:

- **Quantified formula Complexity**: Solving quantified string formulas imposes significant challenge on SMT solvers.
- **Superior Performance of Z3**: Z3 consistently outperforms other solvers, demonstrating efficiency and reliability in handling OCL string constraints.
- **Scalability Issues**: The solving time is proportional to the number of instances (5, 10, 15) as expected. As the number of instances increases, solving times increase substantially, indicating scalability issues in producing large number of instances.
- **Usability and Documentation**: Z3 and CVC5 offer comprehensive documentation and community support, while Ostrich, despite its specialization, lacks extensive user resources.

6 CONCLUSION

In this paper, we present a mapping from OCL string constraints to SMT formulas and a comparison of three SMT solvers in OCL string verification. Our preliminary results help us and the MDE community to identify the strength and limitations of the best three string solvers in verifying OCL string constraints. In the future, we aim to develop a technique for the automatic generation of test cases for OCL, or other specification languages based on user-specified scenarios. This will further help the MDE community to develop more reliable software models.

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String constraints for verification. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 150–166.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. 2017. Z3str3: A string solver with theory-aware branching. *arXiv preprint arXiv:1704.07935* (2017).

[4] Fabian Büttner and Jordi Cabot. 2012. Lightweight string reasoning for OCL. In *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings 8*. Springer, 244–258.

[5] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2009. Verifying UML/OCL operation contracts. In *International conference on integrated formal methods*. Springer, 40–55.

[6] Jordi Cabot and Martin Gogolla. 2012. Object constraint language (OCL): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems*. Springer, 58–90.

[7] Taolue Chen, Riccardo De Masellis, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W Lin, Oliver Markgraf, Philipp Rümmer, et al. [n. d.]. The OSTRICH String Solver. ([n. d.]).

[8] Yu-Fang Chen, David Chocholatỳ, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2024. Z3-noodler: An automata-based string solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 24–33.

[9] Angelo Chiappini, Alessandro Cimatti, Luca Macchi, Oscar Rebollo, Marco Roveri, Angelo Susi, Stefano Tonetta, and Berardino Vittorini. 2010. Formalization and validation of a subset of the European Train Control System. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 109–118.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[11] Andreas A Falkner, Alois Haselböck, Gerfried Krames, Gottfried Schenner, Herwig Schreiner, and Richard Taupe. 2020. Solver Requirements for Interactive Configuration. *J. Univers. Comput. Sci.* 26, 3 (2020), 343–373.

[12] Alessio Ferrari, Alessandro Fantechi, Stefania Gnesi, and Gianluca Magnani. 2013. Model-Based Development and Formal Methods in the Railway Industry. *IEEE Software* 30, 3 (2013), 28–34. https://doi.org/10.1109/MS.2013.44

[13] Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. 2019. OCL FO: first-order expressive OCL constraints for efficient integrity checking. *Software & Systems Modeling* 18, 4 (2019), 2655–2678.

[14] The Object Management Group. 2014. https://www.omg.org/spec/OCL/2.4/PDF

[15] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An efficient SMT solver for string constraints. *Formal Methods in System Design* 48 (2016), 206–234.

[16] Anthony W Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 123–136.

[17] OMG OMG. 1999. Unified Modeling Language Specification, report version 1.3, June 1999.

[18] Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2020. A decision procedure for string to code point conversion. In *International Joint Conference on Automated Reasoning*. Springer, 218–237.

[19] OMG Available Specification. 2006. Object constraint language.

[20] Jos Warmer and Anneke Kleppe. 1998. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc.

[21] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A systematic analysis of XSS sanitization in web application frameworks. In *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 150–171.

[22] Hao Wu and Marie Farrell. 2021. A formal approach to finding inconsistencies in a metamodel. *Software and Systems Modeling* 20, 4 (2021), 1271–1298.

[23] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 114–124.