Evaluating Auto-Generated Spacecraft Software Against a Manually Developed Reference

Youcef Bouziane[1], Abderrahmane Seddjar[1], Fayçal Bouchiba[1], and Amine Meghabber[1]

***Abstract*— Model-based code generation is increasingly promoted as a means to improve the development of safety-critical aerospace software, yet there is a shortage of empirical studies that directly compare generated output to conventional, manually written implementations under real mission conditions. This paper presents a two-part evaluation carried out on the Alsat-1B satellite's Attitude and Orbit Control System (AOCS), for which an ALF-centric (Action Language for Foundational UML) framework was used to produce the onboard code. On the process side, we mine the project's version control history to determine what fraction of the auto-generated code survived into the integrated system without modification and to pinpoint where manual changes were concentrated. On the product side, we conduct a comparative static analysis employing Lizard, SonarQube, Cppcheck, and Clang-Tidy to measure the generated artifacts against the satellite's original, flight-proven C codebase across structural complexity, reliability, and maintainability. The results show that the model-based workflow produces a large share of integration-ready code, yielding artifacts that are structurally simpler, present considerably fewer reliability risks, and satisfy recognised complexity thresholds. These outcomes provide spacecraft software engineers with practical, data-backed insight into the advantages and residual gaps of ALF-centric code generation within an operational satellite programme.**

***Index Terms*— Flight Software, Model-Based Development, Static Analysis, Code Quality, ALF, Alsat-1B**

I. INTRODUCTION

Flight Software (FSW) has become the most complex and rapidly evolving element in modern satellite systems. As onboard capabilities expand to include autonomous operations and real-time fault recovery, the verification burden on engineering teams grows proportionally. Conventional hand-coding practices, while well-understood, frequently produce monolithic, high-complexity modules that are difficult to test exhaustively and prone to latent defects.

Model-Based Development (MBD) has emerged as a promising alternative, enabling early simulation and automated code generation from high-level models. However, a persistent concern in the aerospace domain is whether automatically generated code can match or exceed the quality of manually written, flight-proven software. This concern is compounded by the "Code Generation Dilemma" [1]: as developers manually patch generated code for hardware integration, the semantic link between the model and the deployed software progressively degrades.

In prior work [2], we introduced an ALF-centric framework that uses the Action Language for Foundational UML (ALF) as a single source of truth, bridging the gap between high-level UML models and target code through a two-step transformation. We demonstrated the feasibility of this approach on the Alsat-1B satellite's AOCS process, including co-simulation via the Functional Mock-up Interface (FMI). Prior ALF-based studies, including the translational execution of ALF to C++ [3], bidirectional UML/ALF integration [4], ALF as an intermediary portability language [5], and the unification of modelling and programming through ALF [6], establish transformation feasibility but stop short of measuring the engineering value of the resulting artifacts. Our own preliminary study [2] is no exception: it demonstrates that the pipeline works, not whether its output is quantitatively better than hand-written flight code. The present paper closes precisely this gap by providing, to our knowledge, the first repository-mining and static-analysis comparison of ALF-generated code against an operational, flight-proven baseline.

The present paper shifts focus from framework feasibility to *quantitative evaluation*. We address two complementary questions:

1) *How much manual effort is required after code generation to reach an integration-ready state?*
2) *Does the generated code exhibit better structural quality and reliability than the hand-coded baseline?*

To answer these, we combine repository mining with comparative static analysis, benchmarking the generated Java code against the original handwritten C implementation of the same AOCS subsystem. The key contributions are:

- A repository-mining analysis quantifying post-generation effort and identifying where manual intervention concentrates.
- A controlled static analysis comparison between the generated artifacts and a flight-proven manual baseline.
- Empirical evidence of improvements in complexity and reliability achieved through the model-based workflow, alongside an analysis of the associated maintainability trade-off.

The remainder of this paper is organized as follows. Section II provides background on the framework and the case study. Section III describes the evaluation methodology. Section IV presents the results. Section V discusses the findings and threats to validity. Section VI reviews related work. Section VII concludes.

[1]All authors are with the Satellites Development Centre, Algerian Space Agency, 31130 Oran, Algeria. Corresponding author: ybouziane@cds.asal.dz

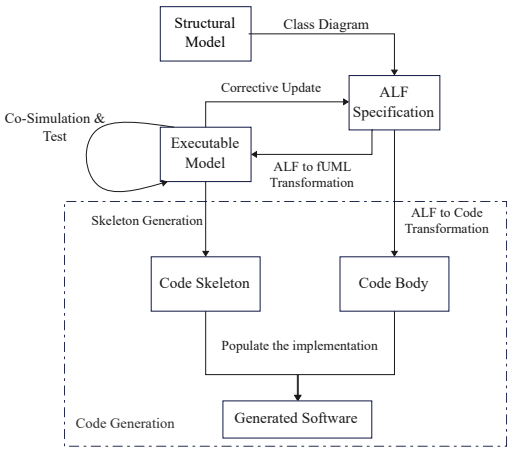

Fig. 1. Architecture of the ALF-centric model-based framework [2].

II. BACKGROUND

A. ALF-Centric Framework

The framework, fully described in [2], is structured around four stages: (1) *Structural Modelling* using UML class diagrams, (2) *Behavioural Specification* using ALF, (3) *Co-Simulation and Test* via fUML execution engines and FMI, and (4) *Code Generation* through a pipeline combining skeleton generation with ALF-to-code transformation. Fig. 1 illustrates this architecture.

The key design principle is that ALF acts as the single source of truth: the same ALF artifacts drive both the model-level simulation (through an ALF-to-fUML mapping) and the final code generation (through an ALF-to-target-language transformation). This ensures that the behavioural properties verified during simulation are preserved in the generated code.

B. Case Study: Alsat-1B AOCS

Alsat-1B [7] is a medium-resolution Earth observation satellite developed by the Algerian Space Agency (ASAL) and Surrey Satellite Technology Ltd (SSTL), launched in September 2016. The AOCS Process Shell operates on a strict 1 Hz control cycle, orchestrating sensor acquisition, algorithm execution, and actuator dispatching through a Finite State Machine (FSM) with three primary states: IDLE, TLMR, and RUNA.

The framework was applied to model, simulate, and generate the code for these AOCS periodic activities. The generated project comprises 13 source files totalling 4,233 Source Lines of Code (SLoC) across 13 components.

III. EVALUATION METHODOLOGY

The evaluation is organized around two complementary pillars, each addressing a distinct facet of the framework's practical value.

TABLE I
STATIC ANALYSIS TOOLS AND SCOPE

Tool	Target	Metrics
Lizard [9]	MC & GC	Cyclomatic complexity, SLoC, function counts
SonarQube [10]	MC & GC	Reliability rating, technical debt, code smells
Cppcheck [11]	MC (C)	Memory safety, buffer handling
Clang-Tidy [12]	MC (C)	Type safety, implicit conversions

A. Pillar 1: Post-Generation Effort

To measure the gap between code generation and integration readiness, we mine the project's Git repository using a mining pipeline built on PyDriller [8]. The generated codebase was committed as a baseline, and all subsequent manual modifications were tracked through version control. We define the *Refactoring Ratio* (RR) as:

$$\text{RR} = \frac{\text{Lines Changed After Generation}}{\text{Total Lines at Deployment}} \times 100\% \quad (1)$$

This metric quantifies how much of the generated code required manual intervention to reach an integration-ready state. A lower RR indicates higher automation effectiveness. We further decompose the RR by component to identify where manual effort concentrates.

For reproducibility, we state the operational definitions precisely. Churn is the sum of lines added and lines deleted within a commit diff, as reported by PyDriller. For each tracked file, the first commit (classified by PyDriller as an addition) is recorded as creation churn and represents the initial automated output; every subsequent modification is accumulated as refactoring churn. Whitespace-only changes are discarded before aggregation. The Refactoring Ratio aggregates refactoring churn across all commits between the generation commit C_{gen} and the final stable commit C_{final}, normalised by total generated SLoC. The analysis scope is restricted to source files within the target package directory, and commits unrelated to the AOCS logic are excluded by commit hash.

B. Pillar 2: Static Code Quality

The second pillar benchmarks the Generated Code (GC) against the original Manual Code (MC) of the Alsat-1B AOCS State Machine. The objective of this assessment is to quantitatively evaluate the structural integrity and maintainability of the automated design artifacts against a human-authored reference implementation.

To ensure an equitable comparison, the MC is strictly filtered to exclude hardware drivers and OS-specific boilerplate that have no equivalent in the generated logic. Both codebases are then subjected to an identical suite of industry-standard static analysers, as summarized in Table I.

The quality assessment is organized around three dimensions:

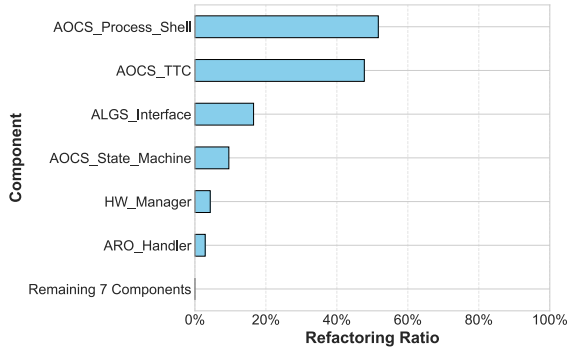

Fig. 2. Component-level Refactoring Ratio of the generated AOCS architecture, illustrating the distribution of post-generation manual effort.

Structural Complexity: Measured via Cyclomatic Complexity Number (CCN) using Lizard. The CCN quantifies the number of linearly independent paths through a function, directly correlating with testing effort. Industry safety guidelines typically recommend a maximum CCN of 15 [13].

Reliability: Assessed using SonarQube's reliability analysis, complemented by Cppcheck and Clang-Tidy. Issues are classified by severity and mapped to specific rules.

Technical Debt: Quantified by SonarQube as the estimated remediation time for all detected code smells and issues, expressed as a Technical Debt Ratio (TDR).

IV. RESULTS

A. Post-Generation Effort

The repository-mining analysis reveals a global Refactoring Ratio of 18.14%. In other words, approximately 82% of the generated code required no manual modification to reach integration-ready status. This result demonstrates a high degree of automation effectiveness for the core algorithmic logic.

However, this refactoring effort is not uniformly distributed across the architecture. A component-level decomposition, illustrated in Figure 2, reveals a clear dichotomy: manual modifications are overwhelmingly concentrated in hardware abstraction layers, such as OS shells and device drivers, where the generator lacks target-specific templates. Conversely, the state machine logic components, which represent the core AOCS algorithms, exhibit near-zero refactoring ratios. This concentration effect is significant because it empirically confirms that the ALF-to-code transformation is highly effective for platform-independent algorithmic logic, while the residual semantic gap is strictly confined to low-level hardware interfacing.

B. Structural Complexity

The Lizard analysis reveals a fundamental architectural improvement. Table II summarizes the key metrics.

TABLE II
COMPLEXITY AND VOLUMETRIC COMPARISON

Metric	MC	GC
Total SLoC	5,760	4,233
Avg. Cyclomatic Complexity	4.4	1.4
Max Cyclomatic Complexity	71	11
Functions > CCN 10	3	1

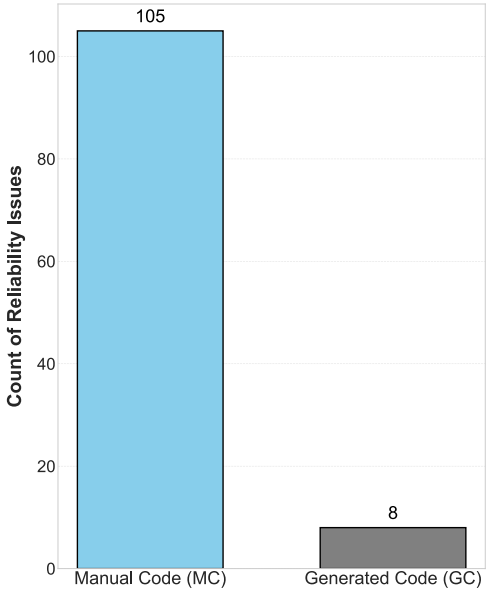

Fig. 3. Overall comparison of detected reliability issues between the manual code (MC) and the generated code (GC).

The generated code achieves a 26.5% reduction in code volume and a 68% reduction in average cyclomatic complexity. The most critical finding concerns extreme complexity hotspots. The MC contains the function `aocs_can_thread` with a CCN of 71: nearly five times the industry safety threshold of 15. Such monolithic "God Functions" are extremely difficult to test exhaustively and are highly error-prone during maintenance. The GC, by contrast, enforces strict modularity; its most complex function peaks at CCN 11, ensuring full compliance with safety standards.

This improvement stems directly from the framework's generation strategy. The ALF-to-code transformer synthesizes state machine logic into well-separated, modular switch-case structures rather than the interleaved, deeply nested control flow that emerges from manual development under schedule pressure.

C. Reliability

The reliability profile, evaluated using SonarQube complemented by Cppcheck and Clang-Tidy, reveals a substantial safety gap between the two implementations. Fig. 3 illustrates the overall disparity.

The MC contained 105 reliability issues (SonarQube Rating: B), while the GC contained only 8 minor issues, representing a 92% reduction in detected reliability risks. The issues detected in the MC span multiple categories, including

memory safety warnings, implicit type handling, and control-flow concerns. In contrast, the residual issues in the GC are confined to minor, non-critical findings.

It is acknowledged that a portion of the observed improvement is attributable to the transition from C to Java, which inherently prevents certain classes of memory violations. However, the elimination of control-flow and logic errors in the generated code, where the MC exhibited multiple such issues and the GC exhibited none, is directly attributable to the deterministic nature of the ALF-to-code transformer. The framework synthesizes complex state machine logic into structured, error-free code, whereas the equivalent manual implementation resulted in a CCN of 71 with associated logic risks.

D. Technical Debt

While the GC outperforms the MC in reliability and complexity, the SonarQube maintainability analysis reveals a counterintuitive result that is commonly observed with automatic code generation [14], [15]:

- **MC:** Technical Debt Ratio 0.4% (Rating A)
- **GC:** Technical Debt Ratio 8.2% (Rating B)

Despite having fewer bugs and lower complexity, the generated code shows a higher Technical Debt. A detailed inspection reveals that this debt is driven almost exclusively by *code smells* (stylistic issues such as variable naming conventions, empty statements, and limited inline documentation), rather than functional defects. The auto-generator prioritizes execution correctness and structural consistency over human-readable styling conventions.

This finding illustrates an important nuance for the MBD context. In a conventional workflow, a "B" rating would trigger remediation. However, in a model-based workflow, the source code is an intermediate compilation artifact, not the primary maintenance target. The *model* remains the source of truth; consequently, stylistic deviations in the generated code do not impact the long-term maintainability of the system.

V. DISCUSSION

A. Interpretation of Results

The two evaluation pillars converge on a consistent picture. The post-generation effort analysis shows that the framework automates approximately 82% of the integration-ready code, with the remaining manual effort strictly confined to hardware abstraction layers. The static analysis confirms that the automated portion is not only voluminous but also of higher structural quality: simpler, more modular, and substantially more reliable than its hand-coded counterpart.

While manual implementation allows for tailored optimization, it inherently accumulates structural entropy and control-flow inconsistencies over iterative development cycles and patch. Conversely, the automated generation enforces rigid structural uniformity. This modularity improvement is particularly critical for flight software, where every independent execution path must be rigorously validated. A function with a Cyclomatic Complexity (CCN) of 71 requires orders of magnitude more test cases than one with a CCN of 11 to achieve equivalent structural coverage.

The higher stylistic technical debt observed in the generated code does not diminish this safety advantage. Within the model-based paradigm, the generated source code occupies a role analogous to object code in a traditional compiler workflow: it is an intermediate compilation target, not the primary artifact that engineers maintain.

B. Threats to Validity

Cross-language comparison: The MC is written in C and the GC in Java, which introduces confounding factors. Memory management issues detected in C may not apply to Java regardless of the generation framework. We mitigate this by focusing on language-independent quality dimensions (complexity, logic errors) alongside language-dependent ones.

Single case study: The evaluation is based on one satellite subsystem. While the Alsat-1B AOCS is a real, flight-proven system, generalizing these results requires replication across other missions. To bound this threat, the chosen subsystem is not a reduced example: it is the complete operational AOCS Process Shell and exercises the full finite-state machine, the 1 Hz timing chain, and the event and error-handling paths. The baseline is the code that actually flew, which removes the idealised blank-page assumption, and cross-mission generalisation is stated as an explicit future-work item rather than claimed here.

Scope limitation: The static analysis comparison focuses on state machine logic. Hardware abstraction layers, which require manual integration, are excluded from the quality comparison but accounted for in the post-generation effort analysis. This exclusion is conservative rather than favourable: it removes precisely the components where the generator is weakest, so it cannot inflate the framework's measured advantage. The excluded effort is not discarded but quantified separately as the hardware-concentrated churn reported in Pillar 1.

Tool bias: Different analysis tools may have varying detection sensitivities. We mitigate this by using multiple complementary tools and focusing on converging evidence rather than individual tool outputs. Tool versions and rule sets are held fixed across MC and GC and are reported with the availability statement so that the measurements can be reproduced.

VI. RELATED WORK

The use of MBD in aerospace has a well-established lineage. NASA's adoption of MATLAB/Simulink for the Orion Crew Vehicle [16] and JPL's application of Model-Based Testing [17], [18] demonstrated the potential for reducing development time and improving robustness. State transition models have been used to enhance flight software validation [19], [20], while commercial toolchains like Rhapsody and SCADE have been applied in automotive and avionics systems [21], [22].

Within the domain of standardised modelling, fUML and ALF have been explored as tool-agnostic alternatives. Ciccozzi [3] demonstrated automated translational execution of ALF to C++, while Schröpfer and Buchmann [4] proposed approaches to overcome the Code Generation Dilemma through bidirectional UML/ALF integration. Recent work by Fuksa et al. [5] highlighted ALF's role as an intermediary language enhancing portability.

However, a gap persists in the literature: existing studies focus overwhelmingly on the *feasibility* of model-to-code transformation rather than the *quality* of the resulting artifacts. Neither the aerospace MBD literature [23] nor the ALF-specific studies [3], [24]–[26] provide quantitative static analysis comparisons against handwritten baselines. The parallel question of generated code quality has been explored recently for LLM-generated code [14], [15], revealing similar trade-offs between functional correctness and maintainability. This paper directly addresses this empirical gap for model-generated flight software.

VII. CONCLUSION

This paper presented a dual-perspective evaluation of an ALF-centric model-based framework applied to the Alsat-1B AOCS subsystem. The repository-mining analysis reveals a Refactoring Ratio of 18.14%, confirming that over 81% of the generated code reaches integration without manual modification, with residual effort strictly concentrated in hardware abstraction layers. The comparative static analysis demonstrates that the generated code achieves a 26.5% reduction in code volume, a 68% reduction in average complexity, the complete elimination of highly complex monolithic functions, and a 92% reduction in detected reliability issues—at the cost of higher stylistic technical debt that remains functionally negligible in a model-based context.

Together, these results provide flight software engineers with empirical evidence that ALF-centric code generation can substantially reduce both development effort and residual defect risk in mission-critical systems. Future work will extend this evaluation to additional satellite subsystems and investigate strategies to reduce stylistic technical debt through optimized code generation templates.

CODE AND DATA AVAILABILITY

The repository-mining pipeline is built on PyDriller [8], and the static analysis uses Lizard [9], SonarQube [10], Cppcheck [11], and Clang-Tidy [12] with fixed, reported versions and rule sets. The analysis scripts and the extracted metric datasets are available from the corresponding author upon reasonable request; the underlying flight source code cannot be released for confidentiality reasons.

ACKNOWLEDGMENT

The authors thank Sébastien Revol and the team at CEA LIST for providing the specialized Eclipse Papyrus distribution and technical resources.

REFERENCES

[1] E. Kindler, "Model-based software engineering and the code generation dilemma," in *Proc. Int. Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, 2010, pp. 1–6.

[2] Y. Bouziane, "Enhancing flight software development through a model-based approach: A case study on Alsat-1B," in *Proc. European Data Handling & Data Processing Conf. (EDHPC)*. IEEE, 2025, pp. 1–5.

[3] F. Ciccozzi, "On the automated translational execution of the action language for foundational UML," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1311–1337, 2018.

[4] J. Schröpfer and T. Buchmann, "Overcoming the code generation dilemma: Bidirectional transformations for UML/ALF," in *Proc. ACM/IEEE Int. Conf. Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2020, pp. 616–625.

[5] M. Fuksa, T. Buchmann, and B. Westfechtel, "Leveraging ALF as an intermediary language for enhanced portability and sustainability of behavioral models," in *Proc. Int. Conf. Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2024.

[6] T. Buchmann and A. Rimer, "Unifying modeling and programming with ALF," *SOFTENG*, 2016.

[7] SSTL, "AlSat-1B – small satellite supplier," Surrey Satellite Technology Ltd, 2016. [Online]. Available: https://www.sstl.co.uk/media-hub/latest-news/2016/sstl-announces-successful-launch-of-alsat-1b

[8] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proc. ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 908–911.

[9] T. Yin, "Lizard: A simple code complexity analyser," 2026. [Online]. Available: https://github.com/terryyin/lizard

[10] SonarSource, "SonarQube: Automatic code review tool," 2026. [Online]. Available: https://www.sonarsource.com/products/sonarqube/

[11] D. Marjamäki and Cppcheck team, "Cppcheck: A tool for static C/C++ code analysis," 2026. [Online]. Available: https://cppcheck.sourceforge.io/

[12] LLVM Project, "Clang-tidy: A clang-based C++ linter tool," 2026. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[13] J. Squire, C. Pate, and C. Rupprecht, "An empirical study of function and cyclomatic complexity thresholds in NASA spaceflight software," *J. Aerosp. Inf. Syst.*, vol. 17, no. 8, 2020.

[14] B. Idrisov and T. Schlippe, "Program code generation with generative AIs," *Algorithms*, vol. 17, no. 2, p. 62, 2024.

[15] A. S. Molison, M. Moraes, G. Melo, F. Santos, and W. K. G. Assuncao, "Is LLM-generated code more maintainable & reliable than human-written code?" arXiv preprint arXiv:2508.00700, 2025.

[16] G. E. Prokop, "Core flight software projects on Orion multi-purpose crew vehicle," in *Annual Workshop on Spacecraft Flight Software*, 2018.

[17] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," *IEEE Software*, vol. 28, no. 3, pp. 62–69, 2011.

[18] J. M. C. Oh, P. J. Watney, and E. P. Benowitz, "Software assurance for model-based design," in *Proc. IEEE Aerospace Conf.*, 2008.

[19] D. Dong, Y. Fang, and Y. Chen, "Application of test method based on state transition diagram in flight control software," in *Proc. Int. Conf. Dependable Systems and Their Applications (DSA)*, 2020, pp. 495–496.

[20] D. Dong, W. Hua, Z. Liu, Y. Fang, and Y. Hou, "Test method of flight control software based on input field model," in *Proc. Int. Conf. Dependable Systems and Their Applications (DSA)*, 2021, pp. 706–711.

[21] T. Tsujimoto and Y. Yamamoto, "Application of model based development to automotive embedded systems," in *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, 2014.

[22] G. Sivakumar and R. Vinu, "A model-based design approach for avionic systems," in *Proc. IEEE Int. Conf. on Electronics, Computing and Communication Technologies (CONECCT)*, 2015.

[23] K. Goseva-Popstojanova and S. Tyo, "Experience report: Assurance case methodology and its application to software-intensive systems," *IEEE Trans. Software Eng.*, vol. 47, no. 12, pp. 2872–2890, 2021.

[24] S. Guermazi, J. Tatibouet, A. Cuccuru, S. Gérard, and E. Seidewitz, "Executable modeling with fUML and ALF in Papyrus: Tooling and experiments," *EXE@ MoDELS*, 2015.

[25] V. Samonov, "Executable modeling language—ontological analysis and best practices," in *Proc. Federated Conf. on Computer Science and Information Systems (FedCSIS)*, 2018, pp. 865–873.

[26] K. Kocataş and H. Oguztuzun, "On the usability of foundational UML," in *Proc. ACM/IEEE Int. Conf. Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023.