Systematic Testing of a ROS Interface Specification Backend

Johannes Mey
Technische Universität Dresden
Dresden, Germany
johannes.mey@tu-dresden.de

Ariel Podlubne
Centre for Tactile Internet with Human-in-the-Loop (CeTI)
Technische Universität Dresden
Dresden, Germany
ariel.podlubne@tu-dresden.de

René Schöne
Technische Universität Dresden
Dresden, Germany
rene.schoene@tu-dresden.de

Paul Gottschaldt
Centre for Tactile Internet with Human-in-the-Loop (CeTI)
Technische Universität Dresden
Dresden, Germany
paul.gottschaldt@tu-dresden.de

Diana Göhringer
Centre for Tactile Internet with Human-in-the-Loop (CeTI)
Technische Universität Dresden
Dresden, Germany
diana.goehringer@tu-dresden.de

Uwe Aßmann
Centre for Tactile Internet with Human-in-the-Loop (CeTI)
Technische Universität Dresden
Dresden, Germany
uwe.assmann@tu-dresden.de

ABSTRACT

Code generators are frequently used when language-independent specifications are compiled into client libraries to support multiple languages. One example is the message definition specification of the Robot Operating System (ROS). This work discusses how a configurable code generator for reconfigurable hardware built using a model-based toolchain based on attribute grammars is tested during development. It supports multiple input and output variants for different source and target languages. To ensure the correctness of all potentially generatable code, a modular test toolchain is provided that can be extended to support different client libraries. Using it, we can identify bugs concerning specification divergence of the tool under test for all current ROS distributions. In this work, we present insights obtained during the design and execution of the test system.

KEYWORDS

Robot Operating System, Specification Testing, Code Generation

ACM Reference Format:
Johannes Mey, Ariel Podlubne, René Schöne, Paul Gottschaldt, Diana Göhringer, and Uwe Aßmann. 2024. Systematic Testing of a ROS Interface Specification Backend. In *2024 ACM/IEEE 6th International Workshop on Robotics Software Engineering (RoSE '24), April 15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3643663.3643964

1 INTRODUCTION

Robotics has become an important field in the research community and industry over the last decades, but there are still open challenges to solve [1]. The application fields range from manufacturing [2], collaborative robots (cobots) interacting directly with humans [3], biomedicine [4], drones for different application [5] as well as mobile robots [6], to name a few. Due to the wide range of applications, robotic platforms are becoming more complex (including heterogeneous sensors and actuators) even more when multiple ones are part of the same system, like an automated warehouse [7]. Hence, their middlewares are preferably vendor-independent.

The Robot Operating System (ROS) [8] is the most popular middleware, providing many open-source packages supporting all kinds of robots, data processing, and planning algorithms. ROS defines *nodes* where computations are performed. They communicate via *topics*, characterized by the *type of message* they transport. Nodes can be *publishers* (produce and broadcast data) or *subscribers* (consume data to process), and a combination of both. ROS provides all the communication mechanisms for all nodes in a system to communicate. They can be programmed in various languages, for which *client libraries* exist. Message types are defined by an Interface Definition Language (IDL). Therefore, each programming language can leverage specific *code generators* to generate language-specific bindings for each message type.

The System Under Test (SUT) in this paper is our previous work FPGA Interfaces for Robotic Middlewares (FIRM) [9], a model-based tool using Attribute Grammars (AGs) and logic-less templates, that consumes message type definitions of ROS as input and produces dedicated hardware components in Very High Speed Integrated Circuit Hardware Description Language (VHDL). The generated VHDL is automatically integrated into a provided hardware architecture that allows hardware accelerators running on the Field Programmable Gate Array (FPGA) to communicate directly with other ROS nodes. The tool was designed to be flexible and can be adapted to other middlewares or changes to the existing ones without much effort. Figure 1, adapted from [9], shows an overview of the system showing how the test system is integrated into the process. ROS messages are obtained from the runtime environment via the ROS command-line interface. These are then parsed and transformed into a model, which is iteratively transformed into VHDL code using model-to-model and model-to-text transformations. A second test generation component uses a similar process to construct a validation architecture using only parsed messages. This ensures that all subsequent transformation steps do not suffer from error propagation through shared components.

RoSE '24, April 15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0566-3/24/04
https://doi.org/10.1145/3643663.3643964

Lately, the industry has been paying more attention to ROS with the arrival of ROS 2, improving the quality of ROS 1 by relying on industry standards, thus enabling more commercial use cases [10]. Each field will impose specific requirements for the robotic systems, such as performance, energy consumption, or real-time guarantees. This poses two challenges for FIRM. First, the tool has to support multiple variants of ROS 1 and ROS 2 in all maintained versions. Second, running the tool in a production environment requires the correct behavior of the generated hardware components and communication architecture for every generatable module. However, ensuring correctness for code generators like our SUT is challenging because it requires validating the semantic equivalence between the input model and the corresponding generated code. Traditional testing techniques often involve simpler input and output domains than code generator models that can have complex syntactic and semantic structures [11]. This work explores the systematic testing methodologies for an experimental ROS client library, specifically emphasizing its message generation capability. Our research provides a systematic and methodical examination of the testing process, focusing on the following aspects.

Thus, we reflect on our efforts to systematically test an experimental ROS client library, focusing on the following questions.

RQ1 How to systematically test interface generator backends?

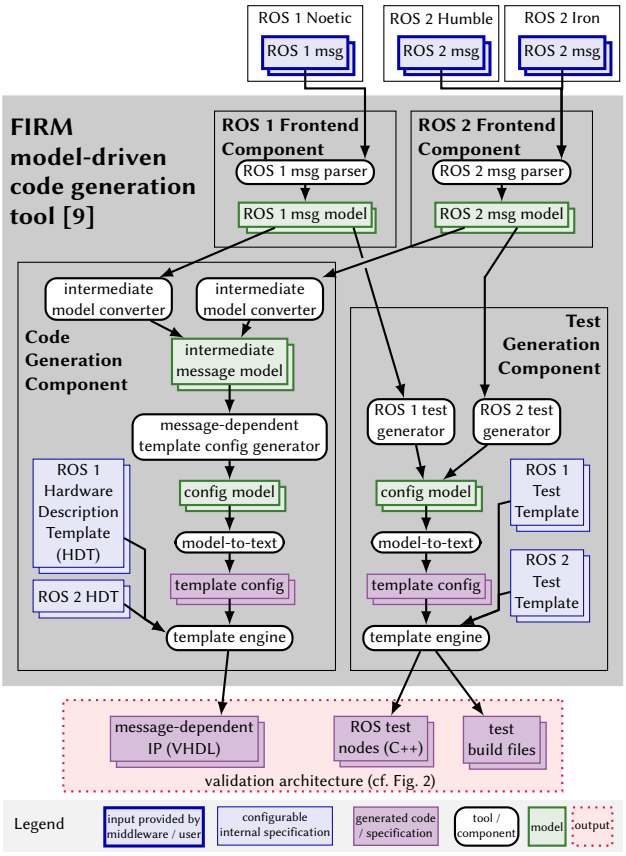


Figure 1: Code generation toolchain (figure is an updated version of [9]). Input files and steps taken depend on the ROS environment the tool is run in.

RQ2 Can the test process be structured to acknowledge the distribution of skills and responsibilities between the specification language and the backend language expert?

RQ3 Can an approach tailored to a code generator based on logic-less templates help improve test quality?

The remainder of this paper is used to argue for three separate test systems for different aspects, shows how these systems are used for our SUT, and discusses the results and insights obtained from the application of the test system (addressing the research questions).

2 RELATED WORK

We provide an overview of how similar testing challenges are addressed, ranging from very specific testing approaches for other ROS client libraries to more general testing approaches for (model- and template-based) code generators.

Most ROS client libraries are created using the *genmsg* tool [12] in ROS 1 and the *rosidl* framework in ROS 2, which both use Python programs and a Python template engine. For some officially supported languages, such as C++ and Python, some tests exist for the individual features of ROS messages. These are sufficient because message code generation happens extremely often during ROS build processes. Therefore, bugs and regressions are detected quickly - something that is not the case for unsupported and more challenging to support languages such as VHDL, which is one of the motivations for this work. Furthermore, other complex message processing tools, such as the message introspection *ros_babel_fish* [13], could benefit from systematic testing.

Further broadening the scope to Template-Based Code Generators (TBCGs), there are only a few testing approaches, such as [14]. Likewise, a survey in the context of MDE [15] identifies a lack of formal verification approaches for template-based code generation. This may be caused by the fact that TBCGs usually use complex DSLs or general-purpose languages in the templates. To address this shortcoming, the survey refers to verification approaches in the field of compilers, which also relates to our AG-based approach. Here, a recent survey [16] presents solutions for grammar-based testing and identifies efficiency and generalizability as interesting research topics.

Stuermer et al. [17] present a model-based testing approach. The challenges identified there, like a lack of specifications and high complexity of the involved systems, can also be found in this work and are investigated here.

Boussaa et al. [18] propose a container-based testing infrastructure for testing the non-functional properties of families of code generators. This approach has later been extended to use metamorphic testing [19]. The *Genesis* framework [20] is a model-based approach that uses execution traces of both input models and generated code to test code generators. The authors argue that using a model-based approach with a common formalism across all its parts enables three test dimensions, parametrized tests, testing multiple target platforms, and across multiple meta-levels.

Ecker et al. [21] present an industry solution for hardware and software code generation based on a model-driven architecture. They verify their generated designs via a separate semi-automatic

property generation flow that transforms temporal traces into testbenches. In conclusion, the review of the state of testing in ROS shows that there is an opportunity for improvement. The research community has identified and (in parts) addressed many aspects that are relevant to this case study, thus making it a topic of interest for further research.

3 MODULAR TEST TOOLCHAIN

Our test toolchain is designed to accompany a model-driven code generation toolchain. Such code generators are centered around a layer-based approach, relying on multiple internal models to generate the code. After parsing an initial model, intermediary representations are created via model-to-model transformations that transform the model in a well-defined way. In the final step, code is generated from the last model via a model-to-text transformation [22].

The proposed test toolchain uses artifacts derived from the intermediate models for test assertions and the generation of test artifacts. Even though it is currently only used to validate FIRM, these mechanisms are generic. They can be applied to similar model-driven tools that generate client libraries from interface specifications. The test toolchain is designed in a modular way similar to the original SUT to provide flexibility towards changes to the underlying interface specification or robotics middleware. It is split into three different test systems, TS1 to TS3, that target different test objectives. All test systems use a set of specifications that covers the specification language as completely as possible (**RQ1**).

Frontend Tests (TS1). This system ensures that the frontend of the SUT complies with the interface specification language. A correct parse tree is required for every input set specification. Therefore, more than a simple cycle of parse, reprint, and compare is required. Instead, we use an existing *oracle* that generates a unique output for each specification. Usually, in the ecosystem of a middleware, such oracles exist, e.g., in the form of introspections, analysis, or visualization tools. Such a test system not only ensures that the frontend complies with the interfaces but allows the identification of potentially tool-breaking bugs early. Additional negative tests can be added to strengthen the trust in specification compliance further. However, such tests require a precise specification, which does not exist for ROS 1 and has issues for ROS 2 (cf. Section 5).

Regression Tests for Semantics (TS2). A set of regression tests ensures that the semantics of the instance are preserved during model transformations and computations. Therefore, the regression tests are not (only) performed on the output of the SUT but on intermediate artifacts in the model-driven toolchain. Although a comparison between a set of input and associated expected output instances would be sufficient, testing intermediate artifacts provides the benefit of identifying bugs early and helps trace them down without knowledge of the target language (addressing **RQ2**).

Message-Level Acceptance Tests (TS3). The previous two test systems target the tool in a traditional software testing sense by ensuring the correct execution of the tool. However, the SUT is a code generator, meaning that the generated code is a system that must be tested. Therefore, our modular test toolchain includes the test system TS3 to ensure that each artifact appropriately handles the input of the provided specification instances. For each tested specification, a system can be *generated* to serve as a test oracle that reads generated example messages per message definition of the tested specification and compares its output with the expected output value. We found that the random example message generation approach works sufficiently well in our case because the actual data values are irrelevant to the generated artifacts but rather have to match. The generated code artifacts can also be embedded to transform the input back-and-forth, removing the need for expected output generation. Generating test cases via the same flow as the tested artifacts always implies a danger of concealing bugs in the test cases. This can be alleviated by solely sharing the frontend part, which is independently tested using TS1.

4 TESTING A ROS FPGA INTERFACE GENERATOR

The approach presented in Section 3 is now applied to the FIRM interface generator for ROS nodes running on FPGA. A test toolchain was constructed in parallel to its development to both enhance the development experience and increase the level of trust in the system to an industrial level. FIRM currently implements the specification of the IDL of ROS 1 and ROS 2, describing how message types can be structured. Individual instances of these specifications represent message definitions whose actual inputs are valid messages of the respective type. An overview of the toolchain of FIRM is shown in Fig. 2. For a detailed overview of the FIRM workflow, see Section IV in [9]. Both FIRM and the test system use analysis and model transformation based on AGs [23, 24] and code generation with the *logic-less* template engine *mustache*.[1] The input of the toolchain is a set of message definitions, using all publicly available definitions installable in the ROS distribution currently supported, identified via their distribution's name (e.g., *humble*), as well as the default set of messages in these distributions and a set of test messages. The output for every definition is a set of VHDL components corresponding to the respective message type. Additionally, ROS nodes and build files for the test infrastructure in TS3 are derived from the input, which is run using an FPGA simulator [25].

Frontend Tests (TS1). To ensure that our SUT parses the message types correctly, two different *oracles* are used for both ROS versions. In ROS 1, the tool `rosmsg show` provides a view of a message type, unrolling all referenced sub-types, that can also be derived from the model using analysis defined for this purpose. In ROS 2, the correctness is checked by constructing a *prototype* of the message type using the command `ros2 interface proto`. Additionally, we use attribute-based analysis to compute several message type metrics to gain insights into the kinds of types used, some of which are already shown in [9], e.g., message type nesting depth. This proved helpful when determining *why* a specific subset of message types failed a test.

Regression Tests for Semantics (TS2). We rely on our intermediary representations used as input for the template engine to test and analyze the transformations and message semantics. Since this engine is *logic-less*, all information is already contained in the template configuration. This configuration is a YAML document,

[1] http://mustache.github.io/

a concise format that is easy to compare automatically (e.g., using JSON Diff [26]) and manually. It is not only used in regression testing but also serves as a human-readable connection document between frontend and backend experts (**RQ2**, **RQ3**).

Message-Level Acceptance Tests (TS3). Since the SUT generates hardware modules written in VHDL, TS3 needs to generate a test infrastructure able to validate the logic of the generated components and check their integration into the ROS ecosystem. To avoid long hardware synthesis times and relatively short execution times (cf. Fig. 5), we use the GHDL [25] simulator instead of FPGA deployment. The generated hardware is tested in three steps for each message type (cf. Fig. 2). The first step (*generation*) involves the generation of all components: the VHDL modules (artifacts) and two ROS nodes, one that produces message instances and transfers them in *serialized* form and one that consumes the messages in a *serialized* form from the FPGA (test artifact). In the next step (*simulation*), the generated VHDL modules are integrated into a GHDL project to execute the corresponding simulation.

It takes the input generated by the first node, which is run through the two generated hardware converters (receiving and sending), connecting a no-op "accelerator". It generates the serialized output to be compared with the input of the simulation in the final step (*testing*). In case of problems, an FPGA expert can analyze and debug the generated hardware components and traces using domain-specific tools.

5 EVALUATION AND INSIGHTS

All test systems are run in a continuous integration system using Docker-based Gitlab CI and run on an AMD Ryzen 9 3900 CPU with 16G RAM for all currently supported distributions (ROS 1 *noetic* and ROS 2 *humble* and *iron*). A structure of the CI is shown in Fig. 3.

Tests are run sequentially to obtain some time measurements, but the entire process is parallelizable on a per-message level. In an additional step, various metrics of the system are computed. All tests are run for ROS 1 and ROS 2. To investigate the influence of the test data set, we tested with *all* installable and functional packages, the default *base* set of messages, and a custom set of *test* messages. For the most recent versions of ROS, 15808 individual tests for 3102 messages types (*noetic*) and 7030 individual tests for 1346 message types (*humble*) were executed. We gained the following vital insights using the presented test process.

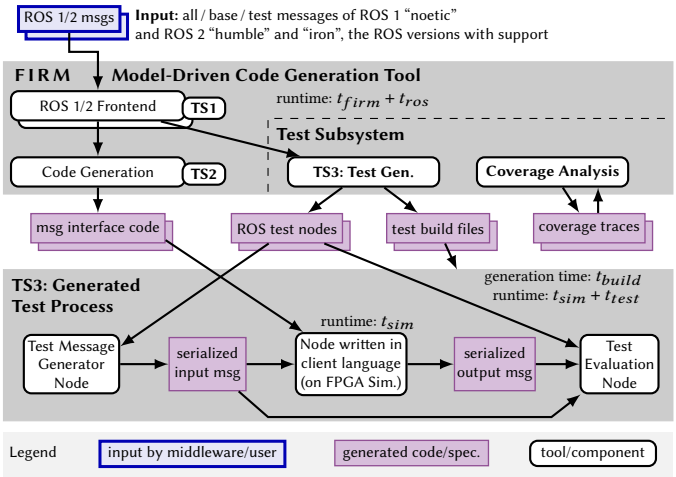


Figure 2: Test systems (TS) in the *FIRM* process.

Managing Test Effort. A primary target is to achieve efficiency in testing. As TS3 takes substantially longer than the other test systems,[2] its execution time was monitored. Figure 5 shows combined test runtimes of TS3 for the latest distributions of ROS 1 and 2. Figure 5 shows the runtimes of the test system for the individual stages of each message run on *noetic*. Within TS3, the *generation* phase takes longer as many artifacts are generated. Hence, the longer the input, the longer the simulation. The last phase is fast since it only compares the serialized input and output of the simulation. Using metrics computed by AG analysis, the runtime can be analyzed more fine-grained (cf. Fig. 6), relating properties of messages to the runtimes of the entire test procedure or individual parts. Using the obtained measurements, we know how long each testing stage takes for each message (**RQ1**). Combined with analysis and a model-based test case generation, this enables us to obtain a reduced number of messages, thus adding the potential for longer-running hardware-in-the-loop tests on the FPGA. The runtime results for *noetic* and *humble* are shown in Fig. 4a and Fig. 4b, with the share of completed tests within a given timeframe. The plots show that most tests take only a few seconds to run (including a constant setup time of ca. ten seconds). At the same time, few have long runtimes, showing opportunities for the automatic construction of test sets, an approach we are currently working on.

Test Specifications. One of the first issues we faced was an insufficient formal specification of the ROS message model, an issue also acknowledged by Stuermer et al. [17]. In ROS 1, the message format is presented informally, with little information about the serialization of messages, so reverse engineering was required. This shortcoming is addressed in ROS 2, which provides a mapping to OMG IDL [27]. However, the message format itself still needs to be formally described, so using it as an input to our toolchain is still problematic. Therefore, we plan to investigate direct IDL support. We generally see it essential to provide formally specified syntax and semantics for specifications.

Test in Stages. While we mostly show the final result of the approach and toolchain, looking at its development process is worthwhile. The staged transformation approach proved helpful for several reasons, providing answers to **RQ1**. First, it allowed us to spot errors and regressions early, which is vital since the models' complexity increases, resulting in hard-to-debug VHDL code. Furthermore, the development was a joint effort from collaborators

[2]Unfortunately, FIRM is also slow due to ROS system calls to obtain the messages specifications, especially with *all* packages installed (t_{ros} in Fig. 5).

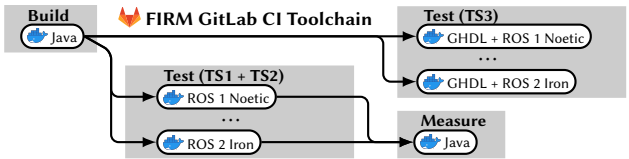
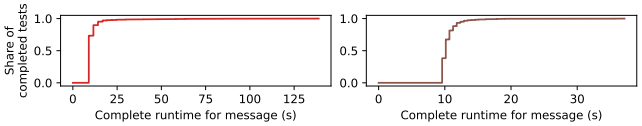


Figure 3: GitLab CI. Steps can use results from predecessors.

with a background in embedded hardware (the domain expert) and language engineering. Our template-based approach allowed the domain expert to specify and test templates without detailed knowledge of the model transformation and analysis mechanisms (**RQ2**), similar to the grey box testing approach by Sakamoto et al. [28]. Thus, we advocate for template-based code generation and logic-less templates specifically (**RQ3**), allowing lightweight template definition and an explicit and concise notation of the analysis results.

Assess the Coverage. The goal of testing is to ensure an (ideally quantifiable) level of quality. This is a two-fold problem in this case since both coverage in the language of *message types* and the set of *messages for each type* must be considered. For the former, we start the pragmatic approach of testing all messages in all current ROS distributions and can easily extend this to new distributions and user-defined messages, e.g., to consider messages obtained by crawling online repositories. Still, investigating coverage metrics and model-based test case generation is an important topic to reduce the enormous test efforts. Using template engines provides some opportunities to obtain coverage for generative tools. We have automatically assessed how much of the code contained in the fragments is used when performing the provided tests by tracing the fragments during template expansion as part of our CI pipeline. This process not only enabled us to eliminate all dead code from the fragments but also serves as an indication of coverage, albeit a weak one, mostly, since values contained in template configurations



(a) ROS 1 *noetic* (3681 msgs). **(b) ROS 2 *humble* (1992 msgs).**

Figure 4: Share of TS3 tests completed in a specific time.

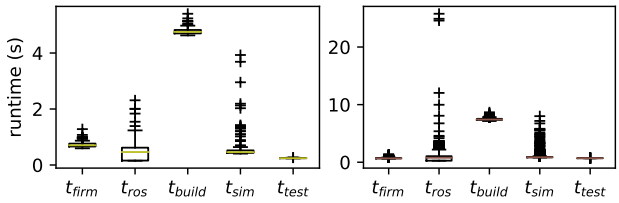
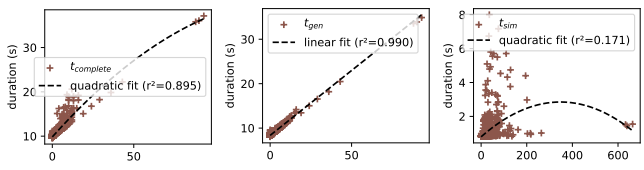


Figure 5: Runtimes for *humble* (base / all msgs.) (cf. Fig. 2).



(a) *Distinct contained msg types (total time).* **(b) *Distinct contained msg types (gen. time).*** **(c) *Number of data fields.***

Figure 6: Examples for correlation analysis of runtime and metrics for *humble*, with fitted functions; this data is available for all stages, metrics and data sets.

are not considered.[3] The coverage of message *instances* in VHDL is hard to define because the code handles signals rather than the control flow structures.

Use Analysis. The core process is controlled by analysis and transformations specified using AGs, a helpful approach for two reasons. First, AGs provide the integrated treatment of structure and semantics. This is in line with [20], discussing the benefits of using a common model in all aspects of the process. Secondly, the analysis gives insight into the SUT. We obtained many metrics from our input model. These are especially relevant when optimizing non-functional properties of code *and* test generation as suggested in [18, 19]. Furthermore, a deep understanding of the code helps to not only systematically evaluate but also to systematically design test data, addressing **RQ1**. While attribute-based analysis is not always an option, similar results can also be achieved with traditionally programmed analysis, possibly with more development effort.

Use Simulation. The process of testing the functionality of each converter directly on an FPGA would take immense time due to all the phases involved in obtaining a bitstream (all the programming information for FPGAs) from a VHDL module. That is why we chose to use a simulator for all messages and only manually test some on an FPGA with distinct characteristics. This helped to validate the logic design, which is then generalized with the simulation of all messages for that distinct characteristic evaluated on the FPGA.

6 CONCLUSION

This work presented the test infrastructure used to construct a complex code generator for FPGA components as ROS interfaces for hardware accelerators.

We showed the complexity of the use case and its relevance for multiple application areas, highlighting some important issues in testing model-driven code generators. Through this work, we obtained the presented insights into designing middleware test systems and were able to design the systematic and structured approach **RQ1** to **RQ3** inquire. The authors continuously apply the presented framework in their research [29, 30]. They are extending the test infrastructure for using an integrated automatic deployment system for ROS nodes on FPGAs with future industry applications as part of a research transfer endeavor. In this context, the application of hardware-in-the-loop (HIL) testing is investigated. While a very specific use case is presented, many aspects can be applied to a broader range of systems, such as code generators (staged testing), systems with long-running tasks (runtime analysis), and tools that use template-based code generation (coverage analysis).

ACKNOWLEDGMENTS

Funded by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt." Joint project 6G-life, project identification number: 16KISK001K and by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1

[3] Tracing was achieved by constructing variants of the templates with added markers. 89% of the fragments created at some point during development were not used in any tests and thus could be removed with minimal risk.

– Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

REFERENCES

[1] G.-Z. Yang, J. Bellingham, P. E. Dupont, P. Fischer, L. Floridi, R. Full, N. Jacobstein, V. Kumar, M. McNutt, R. Merrifield *et al.*, "The grand challenges of science robotics," *Science robotics*, vol. 3, no. 14, 2018.

[2] A. Kumar, "Methods and materials for smart manufacturing: Additive manufacturing, internet of things, flexible sensors and soft robotics," *Manufacturing Letters*, vol. 15, pp. 122–125, 2018.

[3] A. C. Simões, A. L. Soares, and A. C. Barros, "Factors influencing the intention of managers to adopt collaborative robots (cobots) in manufacturing organizations," *Journal of Engineering and Technology Management*, vol. 57, 2020.

[4] F. Soto and R. Chrostowski, "Frontiers of medical micro/nanorobotics: In vivo applications and commercialization perspectives toward clinical uses," *Frontiers in bioengineering and biotechnology*, vol. 6, p. 170, 2018.

[5] U. R. Mogili and B. Deepak, "Review on application of drone systems in precision agriculture," *Procedia computer science*, vol. 133, pp. 502–509, 2018.

[6] M. B. Alatise and G. P. Hancke, "A review on challenges of autonomous mobile robot and sensor fusion methods," *IEEE Access*, vol. 8, pp. 39 830–39 846, 2020.

[7] K. Azadeh, R. De Koster, and D. Roy, "Robotized and automated warehouse systems: Review and recent developments," *Transportation Science*, vol. 53, no. 4, pp. 917–945, 2019.

[8] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009.

[9] A. Podlubne, J. Mey, R. Schöne, U. Aßmann, and D. Göhringer, "Model-Based Approach for Automatic Generation of Hardware Architectures for Robotics," *IEEE Access*, vol. 9, 2021.

[10] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *13th International Conference on Embedded Software*, 2016.

[11] P. Sampath, A. Rajeev, S. Ramesh, and K. Shashidhar, "Testing model-processing tools for embedded systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, 2007.

[12] T. Straszheim, M. Kjaergaard, K. Conley, D. Thomas *et al.*, "genmsg: Standalone python library for generating ros message and service data structures for various languages," 2020. [Online]. Available: https://github.com/ros/genmsg

[13] S. Fabian and O. von Stryk, "Open-Source Tools for Efficient ROS and ROS2-based 2D Human-Robot Interface Development," in *2021 European Conference on Mobile Robots (ECMR)*, 2021.

[14] C. Kolassa, M. Look, K. Müller, A. Roth, D. Reiß, and B. Rumpe, "TUnit - Unit Testing For Template-based Code Generators," 2016.

[15] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems & Structures*, vol. 52, 2018.

[16] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A Survey of Compiler Testing," *ACM Computing Surveys*, vol. 53, no. 1, 2020.

[17] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper, "Systematic Testing of Model-Based Code Generators," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 622–634, 2007.

[18] M. Boussaa, O. Barais, B. Baudry, and G. Sunyé, "Automatic non-functional testing of code generators families," *ACM SIGPLAN Notices*, vol. 52, no. 3, 2016.

[19] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry, "Leveraging metamorphic testing to automatically detect inconsistencies in code generator families," *Software Testing, Verification and Reliability*, vol. 30, no. 1, 2020.

[20] S. Jörges and B. Steffen, "Back-To-Back Testing of Model-Based Code Generators," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer, 2014.

[21] W. Ecker, K. Devarajegowda, M. Werner, Z. Han, and L. Servadei, "Embedded systems' automation following omg's model driven architecture vision," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1301–1306.

[22] V. Guana and E. Stroulia, "Reflecting on model-based code generators using traceability information." in *P&D@ MoDELS*, 2015, pp. 12–15.

[23] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[24] T. Ekman and G. Hedin, "The JastAdd system — modular extensible compiler construction," *Science of Computer Programming*, vol. 69, no. 1, pp. 14–26, 2007.

[25] T. Gingold *et al.*, "Ghdl: free and open-source analyzer, compiler, simulator and (experimental) synthesizer for vhdl," 2022. [Online]. Available: https://ghdl.github.io/ghdl/

[26] P. Bryan (Ed.) and M. Nottingham (Ed.), "JavaScript Object Notation (JSON) Patch," Internet Requests for Comments, RFC Editor, RFC 6902, 2013.

[27] Object Management Group (OMG), "Interface Definition Language, Version 4.2," OMG Document Number formal/18-01-05 (https://www.omg.org/spec/IDL/4.2), 2018.

[28] K. Sakamoto, K. Tomohiro, D. Hamura, H. Washizaki, and Y. Fukazawa, "POGen: A Test Code Generator Based on Template Variable Coverage in Gray-Box Integration Testing for Web Applications," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg: Springer, 2013, pp. 343–358.

[29] A. Podlubne, J. Mey, S. Pertuz, U. Aßmann, and D. Göhringer, "Model-based generation of hardware/software architectures for robotics systems," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 1–7.

[30] A. Podlubne, J. Mey, A. Andreou, S. Pertuz, U. Aßmann, and D. Göhringer, "Model-based generation of hardware/software architectures with hybrid schedulers for robotics systems," *IEEE Transactions on Computers*, 2023.