Automated Test Generation for Medical Rules Web Services: A Case Study at the Cancer Registry of Norway

Christoph Laaber
Simula Research Laboratory
Oslo, Norway
laaber@simula.no

Tao Yue
Simula Research Laboratory
Oslo, Norway
tao@simula.no

Shaukat Ali
Simula Research Laboratory and Oslo
Metropolitan University
Oslo, Norway
shaukat@simula.no

Thomas Schwitalla
Cancer Registry of Norway
Oslo, Norway
thsc@kreftregisteret.no

Jan F. Nygård
Cancer Registry of Norway
Oslo, Norway
UiT The Arctic University of Norway
Tromsø, Norway
jfn@kreftregisteret.no

ABSTRACT

The Cancer Registry of Norway (CRN) collects, curates, and manages data related to cancer patients in Norway, supported by an interactive, human-in-the-loop, socio-technical decision support software system. Automated software testing of this software system is inevitable; however, currently, it is limited in CRN's practice. To this end, we present an industrial case study to evaluate an AI-based system-level testing tool, i.e., *EvoMaster*, in terms of its effectiveness in testing CRN's software system. In particular, we focus on *GURI*, CRN's medical rule engine, which is a key component at the CRN. We test *GURI* with *EvoMaster*'s black-box and white-box tools and study their test effectiveness regarding code coverage, errors found, and domain-specific rule coverage. The results show that all *EvoMaster* tools achieve a similar code coverage; i.e., around 19% line, 13% branch, and 20% method; and find a similar number of errors; i.e., 1 in *GURI*'s code. Concerning domain-specific coverage, *EvoMaster*'s black-box tool is the most effective in generating tests that lead to applied rules; i.e., 100% of the aggregation rules and between 12.86% and 25.81% of the validation rules; and to diverse rule execution results; i.e., 86.84% to 89.95% of the aggregation rules and 0.93% to 1.72% of the validation rules pass, and 1.70% to 3.12% of the aggregation rules and 1.58% to 3.74% of the validation rules fail. We further observe that the results are consistent across 10 versions of the rules. Based on these results, we recommend using *EvoMaster*'s black-box tool to test *GURI* since it provides good results and advances the current state of practice at the CRN. Nonetheless, *EvoMaster* needs to be extended to employ domain-specific optimization objectives to improve test effectiveness further. Finally, we conclude with lessons learned and potential research directions, which we believe are applicable in a general context.

CCS CONCEPTS

• **General and reference** → **Validation**; • **Software and its engineering** → **Software testing and debugging**; • **Applied computing** → **Health care information systems**.

KEYWORDS

automated software testing, test generation, REST APIs, cancer registry, electronic health records, rule engine

1 INTRODUCTION

Cancer is a leading cause of death worldwide, with nearly 10 million deaths in 2020 [20]. Consequently, most countries systematically collect data about cancer patients in specialized registries for the ultimate purpose of improving patient care, by supporting decision-making and conducting research. These registries maintain specialized software systems to collect, curate, and analyze cancer data. However, engineering such software systems poses many challenges, such as (1) collecting data for patients throughout their lives from diverse sources, e.g., hospitals, laboratories, and other registries; (2) dealing with continuous evolution, e.g., due to software updates, new requirements, updated regulations, and new medical research; and (3) increased incorporation of machine learning algorithms for decision support and production of statistics for relevant stakeholders including patients and policymakers.

Our context is the Cancer Registry of Norway (CRN), which has developed a Cancer Registration Support System (CaReSS) to support collecting, processing, and managing cancer-related data from various medical entities such as Norwegian hospitals and laboratories. Based on the processing, the CRN generates statistics that are consumed by external entities, including policymakers, hospitals, and patients. It further provides data for researchers

to conduct research. Naturally, the quality of statistics and data depends on how correct and reliable CaReSS is. To this end, testing is one method to ensure the dependability of CaReSS to a certain extent.

This paper reports on a case study in the real-world context of the CRN, focusing on testing one of the key components of CaReSS, called *GURI*– a rule engine responsible for checking medical rules for various purposes such as data validation and aggregation. Currently, *GURI*'s testing is primarily manual, as also reported by Haas et al. [25] that manual testing is a common practice in industry. Our study takes a popular open-source, AI-based system-level testing tool called *EvoMaster* [6], which has been shown to be superior to 9 other representational state transfer (REST) application programming interface (API) testing tools [28], and performs automated testing of *GURI* to assess its effectiveness in testing *GURI* from various perspectives.

We apply *EvoMaster*'s four tools, i.e., black-box and white-box tools with three evolutionary algorithms (EAs), to test *GURI*'s ten versions. In particular, we assess *EvoMaster*'s capability to achieve source code coverage, errors found, and domain-specific rule coverage. The results of our experiments show that all four tools' effectiveness is similar in terms of code coverage and errors found across all *GURI* versions. However, we observe that *EvoMaster*'s black-box tool is more effective for domain-specific coverage. We further compare the results with *GURI* in production, and the results of the black-box tool were closer to the production system.

Based on our results, we recommend using *EvoMaster*'s black-box tool as the starting point to automate the testing of *GURI* and CaReSS. However, this study also shows that *EvoMaster* must be customized for CRN's context by explicitly incorporating domain-specific coverage elements (e.g., coverage of different types of cancers and treatments) in the search process, e.g., encoding as new fitness functions, as also argued by Böhme et al. [11]. Finally, we provide detailed discussions and lessons from our industrial case study regarding its generalization to other contexts and point out key research areas that deserve attention from the software engineering community.

2 BACKGROUND AND CONTEXT

In this section, we first discuss the CRN context, *GURI* and the medical rules that *GURI* relies on, followed by background on *EvoMaster*.

2.1 Application Context

The CRN regularly collects cancer patients' data (e.g., diagnostic, treatment), based on which cancer research and statistics can be conducted. To ensure the quality of the collected data, the CRN's CaReSS has introduced several preventive efforts to discover and amend inaccurate or missing data. Patient data is submitted to the CRN as *cancer messages*, from which *cancer cases* are derived via a coding and aggregation process, representing a timeline of a patient's diagnoses, treatments, and follow-ups. The coding process relies on standard classification systems and depends on hundreds of *medical rules* for validating cancer messages and aggregating cancer cases. Consequently, these rules are of two categories: *validation rules* and *aggregation rules*, which are defined by medical experts and implemented in *GURI* for automated validation and aggregation of cancer messages and cases. These rules constantly evolve, e.g., due to updated medical knowledge and procedures.

Below is a validation rule, which states that for all the cancer messages of type *H*, if the *surgery* value is equal to 96, then the *basis* value must be greater than 32.

$$\forall\ messageType = H \implies (surgery = 96 \implies basis > 32)$$

The aggregation rule below determines the state of *Morphologically verified*, based on a given *Basis* value of cancer messages:

---

if Basis ∈ ['22', '32', '33', '34', '35', '36', '37', '38', '39', '57', '60', '70', '72', '74', '75', '76', '79'] **then**
    Morphologically verified = 'Yes'
else if Basis ∈ ['00', '10', '20', '23', '29', '30', '31', '40', '45', '46', '47', '90', '98'] **then**
    Morphologically verified = 'No'
else
    Morphologically verified = null
end if

These rules are stored in *GURI*'s internal database. New rules and updates to the rules are done through graphical user interface (GUI), which is available for medical personnel.

2.2 *EvoMaster*

EvoMaster [6] is an open-source, automated and search-based software testing framework. EA are employed to optimize search objectives involving coverage criteria such as line, branch, and method coverages. *EvoMaster* can be used with different programming languages, such as Java and Python. The black-box testing tool of *EvoMaster* [7] relies on random testing with multi-objective search to maximize black-box metrics (e.g., endpoint coverage and status code coverage) and fault detection capability (500 status code). *EvoMaster* also defines a series of white box testing tools [36, 47], combined with different multi-objective EAs (e.g., Many Independent Objective (MIO)), for achieving various test generation purposes by providing a comprehensive list of coverage criteria, including lines, branches, and faults.

3 EXPERIMENTAL STUDY

We perform a laboratory experiment [41] of automated test generation techniques for REST APIs at the CRN to understand their effectiveness of covering source code, revealing errors, and executing domain-specific elements, i.e., medical rules. Our experiment comprises a real-world study subject, i.e., CRN's medical rule engine *GURI*, and four REST API test generation tools.

3.1 Research Questions

Our study investigates the following four research questions (RQs) to assess the effectiveness of the test generation tools:

RQ 1   How much code coverage do the tools achieve?

RQ 2   How many errors do the tools trigger?

RQ 3   How many rules can the tools execute?

RQ 4   Which results do the rule executions yield, and how do these compare to production *GURI* rule execution results?

Table 1: *GURI* Meta Data

Metric		Value
Rules	Validation	71
	Aggregation	43
REST Endpoints	Rule Handling	2
	Total	32
Versions	Selected	10
	Total	28

Table 2: Rule Evolution

Version	Date	Rules	
		Validation	Aggregation
v1	12.12.2017	30	32
v2	30.05.2018	31	33
v3	06.02.2019	48	35
v4	27.08.2019	49	35
v5	11.11.2019	53	37
v6	25.09.2020	56	37
v7	24.11.2020	66	38
v8	20.04.2021	69	43
v9	13.01.2022	69	43
v10	21.01.2022	70	43

RQ 1 and RQ 2 are "traditionally" investigated research questions for evaluating test generation tools in terms of effectiveness, including REST API test generation [28]. RQ 3 and RQ 4 are domain-specific RQs and evaluate the test generation tools' effectiveness in testing *GURI*'s main functionality, i.e., validating and aggregating cancer messages and cancer cases. RQ 3 studies the test generation tools' capability to execute the medical rules with the ultimate goal of testing them. The goal of RQ 4 is to investigate which results the executed rules yield with test generation tools and how these results compare to the results from the production system.

3.2 Study Subject — *GURI*

GURI is implemented as a Java web application with Spring Boot, which exposes REST API endpoints to CRN's internal systems and provides a web interface for medical coders. Table 1 shows an overview of *GURI*. Although *GURI* has 32 REST endpoints, we only focus on 2 in this experiment, as these are the ones handling the rules, i.e., one for validating cancer messages with validation rules and one for aggregating cancer messages into cancer cases with aggregation rules. *GURI*'s most recent version consists of 71 validation rules and 43 aggregation rules. Since *GURI* was introduced, its source code has hardly changed; however, its rules have been subject to evolution due to updated medical knowledge, such as rule additions, deletions, and modifications. Consequently, based on these changes, we form ten rule sets as ten versions in this experiment. Specifically, out of 28 unique points in time where the rules were changed in *GURI*, we select 10 dates where the changes are most severe (the most rule additions and deletions occurred). Table 2 depicts this rule evolution.

3.3 Test Generation Tools

The automated REST API test generation tools form the independent variable of our experimental study. We select *EvoMaster* in version v1.5.0[1] with multiple parameterizations as the tools [6]. *EvoMaster* was recently shown to be the most effective tool, in terms of source code coverage and triggered errors, among ten different tools [28]. A tool, in the context of our study, is a specific *EvoMaster* parameterization, consisting of the testing approach and the employed EA.

In terms of testing approach, our experiments use both black-box and white-box testing. *EvoMaster-BB* relies on a random-testing approach to generate the REST requests [7]. On the other hand, *EvoMaster-WB* uses an EA to generate tests based on randomly

generated (initial) REST requests, coverage feedback, and mutation [6]. *EvoMaster* supports three EAs: (1) MIO [5], which is a many-objective evolutionary algorithm (MaOEA) focusing on scalability in the presence of many testing targets, that was specifically designed for REST API test generation and is *EvoMaster*'s default; (2) Many-Objective Sorting Algorithm (MOSA) [39], which is the first MaOEA that was designed for unit test generation with *EvoSuite* [21]; and (3) Whole Test Suite (WTS) [22], which is a single objective genetic algorithm (GA) that was designed for unit test generation and is the original EA of *EvoSuite*. Our experiment investigates all four *EvoMaster* tool parameterizations: (1) **EvoMaster-BB**: *EvoMaster* black-box; (2) **EvoMaster-WB-MIO**: *EvoMaster* white-box with MIO; (3) **EvoMaster-WB-MOSA**: *EvoMaster* white-box with MOSA; and (4) **EvoMaster-WB-WTS**: *EvoMaster* white-box with WTS. Beyond the testing approach and the EA, our experiment uses the default parameters of *EvoMaster*, similar to Kim et al. [28].

3.4 Evaluation Metrics

To evaluate the effectiveness of the tools and answer the RQs, we rely on the following evaluation metrics, which are the dependent variables of our study.

RQ 1 - code coverage. We rely on **line**, **branch**, and **method coverage** extracted with *JaCoCo*[2], similar to Kim et al. [28].

RQ 2 - errors. We use three types of 500 errors triggered by the tools, as defined by Kim et al. [28]: (1) **Unique Errors** are the number of errors grouped by their complete stack traces; (2) **Unique Failure Points** are the number of occurrences of the same error, i.e., the first line of a stack trace; and (3) **Unique Library Failure Points** are the number of errors that are unique failure points occurring in the library code.

RQ 3 - rule execution status. The first domain-specific metric is the number of executed rules during a tool's execution. We distinguish three types: (1) **Applied** is a rule that is fully executed at least once on an input, irrespective of the result (pass, fail, or warning); (2) **Not Applied** is a rule that is partially executed against an input, which only applies to validation rules of the form $apply \implies rule$,

---

where *apply* is the condition on which *rule* is applied; (3) **Not Executed** is a rule that is never executed on any input.

RQ 4 - rule execution results. The second domain-specific metric are the execution results of rules that have been applied (see RQ 3). We distinguish three types: (1) **Pass** results from a successful rule execution, i.e., the cancer message with all its data is valid, or the cancer case was successfully aggregated with a previous cancer case and a number of cancer messages; (2) **Fail** results from an unsuccessful rule execution, i.e., the cancer message is invalid, or the cancer case fails to be aggregated; (3) **Warning** is the result where the rule execution is successful; however, the data appears to be dubious (e.g., a patient's age is 120 years, which is theoretically valid but highly unlikely).

To answer RQ 4, we compare the results of the tools, based on these three categories, to the results of the production *GURI*, deployed at the CRN to determine if their similarity.

3.5 Experiment Setup

The experiment setup is concerned with the experiment execution settings and execution environment.

To deal with the stochastic nature of the EAs underlying *Evo-Master*, each tool is executed repeatedly for 30 repetitions [8]. The analyses then rely on the arithmetic mean across all the repetitions. Each tool is executed for 1 hour for each repetition and version by following the practice of Kim et al. [28]. They also observed that for complex and constrained input parameters, source code coverage and thrown errors hardly increase after 10 minutes, which is also our case. Beyond these settings, the experiments consider the tool parameterizations from Section 3.3. The experiment further randomizes the order of the tools and versions in each repetition, with randomized multiple interleaved trials (RMIT) [1], reducing potential confounding factors that stem from the execution environment or the execution order.

We executed the experiments on the Experimental Infrastructure for Exploration of Exascale Computing (eX^3) high-performance computing (HPC) cluster[3] hosted at the first author's institution, which uses Slurm 20.02.7 as its cluster management software. The experiments were scheduled on nodes of the same type (using the *slowq* partition of eX^3), with the whole node exclusively reserved. The nodes have 8 Intel(R) Xeon(R) Silver 4112 central processing units (CPUs) @ 2.60 GHz each with 4 cores, run Ubuntu 18.04.1, and have 40 GB total memory. The experiments were conducted in the first half of 2023. The Java tools were exclusively built and ran with OpenJDK 11.0.18 built by Adoptium[4].

3.6 Threats to Validity

We classify threats into construct, internal, and external validity.

The biggest threat to the **construct validity** concerns the choice of the metrics for evaluating the tools' effectiveness. We rely on metrics widely used in the API test generation research, in particular, adapted by a recent publication [28], to answer RQ 1 and RQ 2. For RQ 3 and RQ 4, we employ two sets of domain-specific metrics, i.e., executed rules in RQ 3 and rule execution results in RQ 4, which are of specific interest to the CRN. Nevertheless, it is

unclear whether these domain-specific metrics are correlated with "good" test cases for the domain experts or sufficiently targeted to assess test generation tools for the CRN. Further investigation via dedicated empirical studies is required to answer this question.

In terms of **internal validity**, a crucial threat is that *EvoMaster*white-box requires manually creating a subject under test (SUT) driver. Failure to do so concerning how *EvoMaster* expects the driver to be implemented and *GURI* requires to be controlled could threaten the *EvoMaster-WB* results. Similarly, an incorrect injection of the *JaCoCo* code coverage agent, implementation of the analyses scripts, and adaptation of *GURI* to retrieve rule executions could alter the study's results and implications. We thoroughly tested our implementations to validate the correct behavior to mitigate this threat. Further internal validity threats relate to the experiment design include: (1) the number of repetitions (30 in our study), (2) the time budget for test generation (1 hour), and (3) the tool parameters (see Section 3.3). These design decisions are based on previous research [8, 28]; however, different experiment design decisions might lead to different results. Finally, *EvoMaster* relies on *OpenAPI*[5] schema definitions to generate tests. An incorrect schema definition potentially leads to sub-optimal tests, which could impact the reported results. Our experiment uses the schema definitions generated by *springdoc-openapi*[6], which *GURI* already employs. Considering many parameters to configure, we try to build the base of our empirical study on the knowledge built by the existing literature to mitigate these internal threats.

The primary **external validity threat** is related to the generalization is inherent to the study design: a case study on a single study subject, i.e., CRN's rule engine *GURI*. All results are only valid in the context of our case study and are probably not transferable to other case studies. Nevertheless, we provide implications and "more general" conclusions in the discussion section. Moreover, our results are tightly coupled with the test generation tool(s), i.e., *EvoMaster* in its four parameterizations (see Section 3.3), and do not generalize to other REST API test generation tools. Finally, we perform a laboratory experiment on the research HPC cluster eX^3 with a standalone version of *GURI* and not a field experiment against the real-world *GURI* (or the testing environment) hosted at the CRN. Consequently, our results might not generalize to the real system. The extraction of the real-world *GURI* into a standalone version for the empirical study, was, however, done by the CRN developers, which should reduce this threat.

4 RESULTS

This section presents the results for each RQ.

4.1 RQ 1: Code Coverage

We study the code coverage achieved by the tools in terms of line, branch, and method coverages. Table 3 depicts the coverage results. Each coverage value consists of the arithmetic mean and standard deviation across all the versions and repetitions. We refrain from reporting coverages for each version, as the source code does not change between versions; only the rules change (see Section 3.2).

[3]https://www.ex3.simula.no
[4]https://adoptium.net
[5]https://www.openapis.org
[6]https://springdoc.org

Table 3: Source code coverage per tool across all the versions and repetitions. The values are arithmetic means ± standard deviations in percentages.

Tool	Line	Branch	Method
EvoMaster-BB	19.74%±0.44	14.24%±0.63	20.03%±0.18
EvoMaster-WB-MIO	18.68%±1.67	12.97%±1.88	19.45%±1.23
EvoMaster-WB-MOSA	18.14%±1.54	12.37%±1.77	19.13%±1.03
EvoMaster-WB-WTS	19.25%±2.26	13.76%±2.31	19.68%±1.92

We observe that all tools perform similarly, i.e., approximately 20% line coverage, 14% branch coverage, and 20% method coverage. Note that the relatively low (absolute) coverage values must be considered with caution because we generated tests for (only) 2 of 32 REST endpoints (see Section 3.2), as only these two handle the medical rules. However, the code coverage is still of interest as a relative comparison among the tools. With regards to it, the tools perform similarly, which is a different observation as obtained by Kim et al. [28], where *EvoMaster-WB-MIO* achieves a higher code coverage than *EvoMaster-BB* by 7.35 percentage points (pp) (line), 7.87 pp (branch), and 5.69 pp (method). In addition, Kim et al. [28] report that the line and method coverages are similarly higher than the branch coverage, which aligns with our findings.

A potential reason for the low coverage values is that *EvoMaster* produces many invalid requests, as *GURI* expects specific JavaScript Object Notation (JSON) input parameter values, i.e., medical variables contained in cancer messages and cases. In particular, *EvoMaster* often fails to provide valid date strings. This behavior is also observed by Kim et al. [28]. An alternative reason is that all the tools reach the maximum code coverage achievable through the two REST APIs used in our experiment. The remaining RQs will shed more light on these hypotheses.

> **RQ 1 Summary:** All tools achieve a similar line, branch, and method coverage. In the context of the CRN, opting for a simpler test generation tool, i.e., *EvoMaster-BB*, is preferred over a more complex tool, i.e., any *EvoMaster-WB*, to cover more source code.

4.2 RQ 2: Errors

The second RQ studies the thrown errors by the tools in terms of unique 500 errors, unique failure points, and unique library failure points. Table 4 shows the results for each tool across all the versions and repetitions. Similar to RQ 1, we refrain from reporting errors for each version. For each error category, the table depicts four values: (1) the total number of errors ("All"); (2) errors that are due to the tools' intervention, e.g., the attached Java agent ("Tool"); (3) input-output (I/O) errors that occur during the test generation ("I/O"); and (4) remaining errors that are not attributed to the Tool and I/O categories but actually due to errors thrown by the application, i.e., *GURI* ("Remaining").

Considering unique errors (i.e., errors with identical stack traces, see Section 3.4), we observe that the different tools trigger between 3 and 6.46 errors ("All"); however, on a closer inspection, we notice

that the three *EvoMaster-WB* tools experience a high number of tool-related errors, i.e., the coverage agent attached to the Java virtual machine (JVM) throws a `KillSwitch` exception when concurrent threads are running after the test generation has finished. This scenario inflates the "All" errors. Moreover, we can see that all tools suffer from a varying degree of I/O errors, most often due to broken I/O pipe exceptions. Finally, the number of "Remaining" unique errors is similar across all four tools, i.e., approximately 1. These results align with the code coverage results from RQ 1, i.e., no tool is superior over the others.

Regarding the unique failure points (i.e., errors where the top-most line of the stack trace is identical, see Section 3.4), we observe a similar trend: "All" errors are inflated by tool-related and I/O errors, leaving approximately 1 "Remaining" unique failure point, which is, again, consistent across all tools. In most cases, this 1 failure point is caused by a date parsing exception, e.g., when the diagnosis date of a cancer message is malformed. Although seldom, all four tools trigger a unique library failure point (i.e., a unique failure point where the cause is located in *GURI*'s source code, see Section 3.4). Upon closer inspection, we identify 1 *GURI* rule parsing error on specific inputs.

Compared to Kim et al. [28], the tools reveal fewer errors, and no tool is superior in the context of the CRN.

> **RQ 2 Summary:** All the tools reveal the same number of errors and failure points. Similar to RQ 1, this suggests employing the most straightforward tool, i.e., *EvoMaster-BB*, for its error-revealing capabilities.
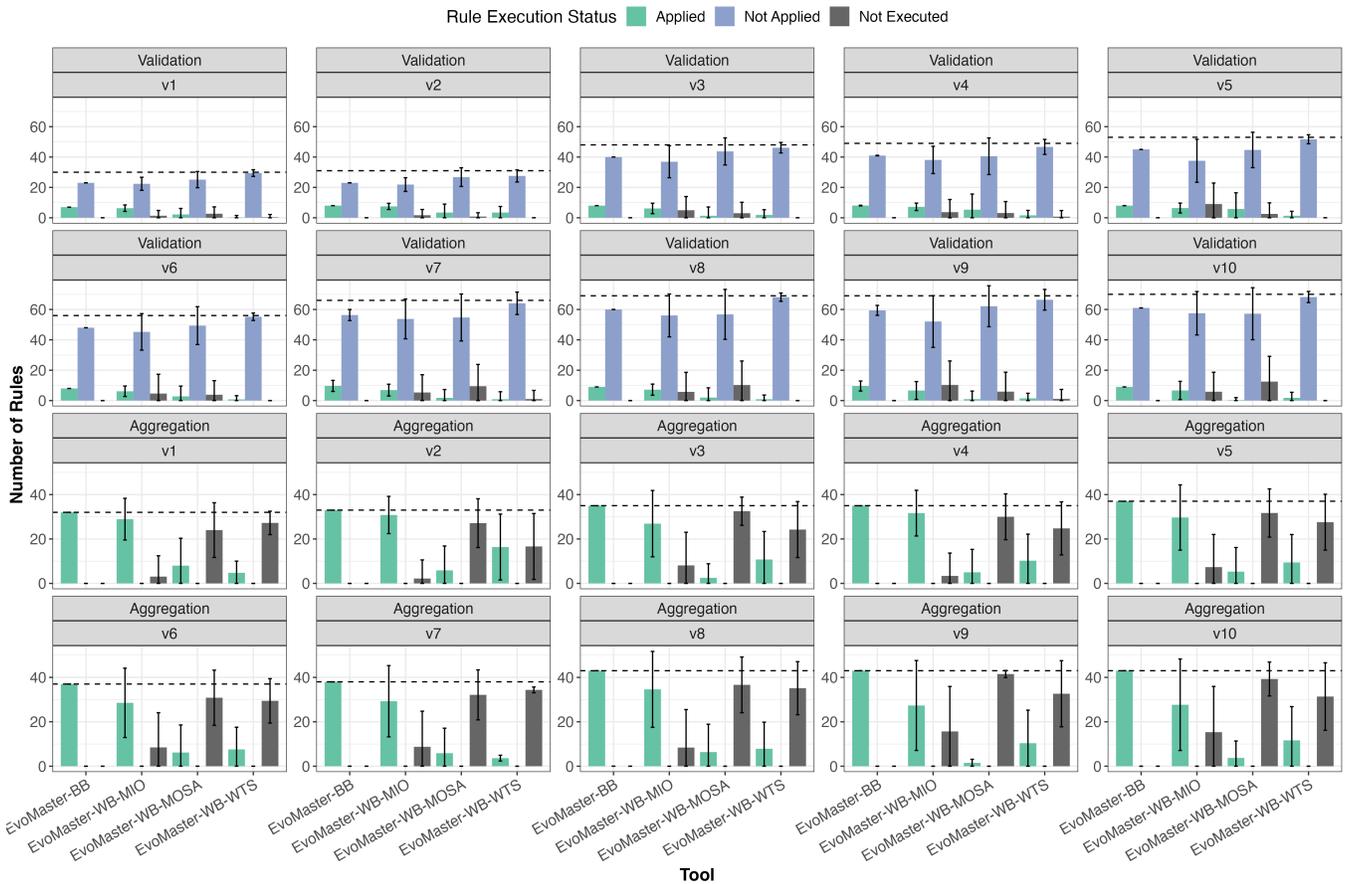
4.3 RQ 3: Rule Execution Status

This domain-specific RQ evaluates the tools' ability to execute medical rules. Figure 1 shows the number of (distinct) rules (on the y-axis) by each tool (on the x-axis) for each version of *GURI*. The dashed line depicts the total number of rules in the particular version. Each bar represents the arithmetic mean of applied, not applied, or not executed rules (see Section 3.4), with the error bars indicating the standard deviations. The first two rows depict the validation rules, whereas the last two rows show the aggregation rules.

We make three main observations: (1) there is a difference in effectiveness depending on which tool is employed, which is different from what we observe for RQ 1 and RQ 2; (2) the tools are not equally effective for validation and aggregation rules; and (3) the tool effectiveness does not change across the ten versions. We discuss these observations in detail below:

4.3.1 Observation 1: Tool Differences. *EvoMaster-BB* executes the most validation and aggregation rules, as the not executed category has near zero rules. *EvoMaster-BB* generate tests where *all* the aggregation rules (between 32 and 43) are applied (in all versions) and between 7.00 (12.86%) and 9.67 (25.81%) validation rules, depending on the version. Whereas *EvoMaster-WB-MIO*, the best-performing white-box tool, only generates tests for which between 26.90 (63.57%) and 34.60 (93.33%) aggregation and between 6.13 (9.52%) and 7.47 (24.09%) validation rules are applied.

Table 4: 500 errors per tool across all the versions and repetitions. The values are arithmetic means ± standard deviations.

Tool	Unique Errors				Unique Failure Points				Unique Library Failure Points
	All	Tool	I/O	Remaining	All	Tool	I/O	Remaining	
EvoMaster-BB	3±1.02	0±0	1.93±0.99	1.07±0.33	2.03±0.27	0±0	0.97±0.18	1.07±0.32	0.00±0.06
EvoMaster-WB-MIO	6.46±2.82	4.43±2.32	1.01±0.69	1.02±0.19	2.88±0.89	1.06±0.58	0.8±0.40	1.02±0.13	0.02±0.13
EvoMaster-WB-MOSA	4.16±2.39	2.92±2.12	0.2±0.48	1.04±0.31	2.17±0.82	0.97±0.57	0.17±0.38	1.03±0.21	0.02±0.15
EvoMaster-WB-WTS	6.04±1.75	4.78±1.53	0.18±0.38	1.08±0.52	2.5±0.65	1.29±0.47	0.18±0.38	1.03±0.20	0.03±0.16

Figure 1: Rule execution status for each rule per tool and version across all the repetitions. The bars are arithmetic means, the error bars are standard deviations, and the dashed line depicts the number of total rules of a version.

This is a surprising result because *EvoMaster-WB*, compared to *EvoMaster-BB*, is on par in our study (i.e., RQ 1 and RQ 2) and superior in a recent comparison of REST API test generation tools [28]. One reason is that the EA of *EvoMaster-WB* optimizes for "irrelevant" objectives. Once *EvoMaster-WB* finds better solutions for the source code coverage, it steers itself into a situation where it does not execute more (different) rules anymore. Conversely, *EvoMaster-BB*, with its simpler approach (concerning covering source code), exercises more diverse inputs (i.e., cancer messages and cases), which leads to more executed rules. This is particularly evident for aggregation rules, where *EvoMaster-BB* applies all the rules,

followed by *EvoMaster-WB-MIO* which applies between 63.57% and 93.33% of the rules, depending on the version.

Moreover, we notice that *EvoMaster-BB* has no variance in the number of executed rules among identical repetitions, whereas all the *EvoMaster-WB* experience a variance to a varying degree, as indicated by the error bars in Fig. 1. This means that the *EvoMaster-WB* tools cannot consistently execute the same rules for identical repetitions.

4.3.2 Observation 2: Rule Type Differences. This leads to the second observation, where the tools are not equally-effective in executing validation rules as they are for aggregation rules. *EvoMaster-BB*

executes all the aggregation rules but only achieves applying between 12.86% and 25.81% of the validation rules. It is worse for the *EvoMaster-WB* tools. We conclude that *EvoMaster* struggles to generate tests that cover considerably more validation rules. The reason is inherent to many validation rules, which are only applied if the left part of an implication is true (see Section 3.4). Kim et al. [28] observe a similar situation, where the tools generate many invalid requests (due to invalid parameter values) that are rejected by the APIs; however, in our case, the requests are not rejected, as they conform with the *OpenAPI*, but lead to rules that fall into the not applied category.

We further observe that the *EvoMaster-WB* tools suffer to a varying degree from rules not being executed, which is more pronounced for *EvoMaster-WB-MOSA* and *EvoMaster-WB-WTS* than for *EvoMaster-WB-MIO*. This means that the tests never reach a point (in the source code) where the rules are considered for execution. Arcuri [5] also finds that MOSA and WTS perform inferior to MIO on most (but not all) problem types for the source code coverage, which has the consequence that more rules are not executed.

4.3.3 Observation 3: Version Differences. Finally, we observe that, with changing rule sets due to the addition, deletion, and modification of rules, the tools' effectiveness is hardly impacted. For *EvoMaster-BB*, the standard deviation of the relative number of applied rules (with respect to the total number of rules) in each version is 0% and 4.42% for aggregation and validation rules, respectively. There is slightly more variation for *EvoMaster-WB*, i.e., up to 10.89% standard deviation, suggesting that the rule evolution at the CRN is not a factor for choosing a particular tool over the other.

> **RQ 3 Summary:** *EvoMaster-BB* is more effective in generating tests that apply the rules. In particular, aggregation rules are considerably easier to cover than validation rules. The *EvoMaster-WB* tools are less effective than *EvoMaster-BB*, and *EvoMaster-WB-MIO* performs the best among all the three white-box tools. All the tools are similarly effective in generating tests that lead to rules being applied across the rule set versions.

4.4 RQ 4: Rule Execution Results

This domain-specific RQ evaluates the tools' capabilities to generate tests that lead to the three rule execution results, i.e., pass, fail, and warning (see Section 3.4). Figure 2 shows the rule execution results for the applied rules relative to the total number of rules in a version (on the y-axis) per tool (on the x-axis), version and rule type. For this RQ, not applied and not executed rules are disregarded. Each bar represents the arithmetic mean, and the error bars are the standard deviations. The first two rows are for the validation rules, and the last two rows are for the aggregation rules.

We make five observations: (1) there is a high degree of variance among repetitions in terms of the execution results; (2) there is a difference in the prevalence of the individual execution result for aggregation and validation rules; (3) the tools follow the same effectiveness ranking as in RQ 3; (4) the tools are differently effective (although minor) for different versions; and (5) the result distribution of the generated tests is different from production *GURI*.

4.4.1 Observation 1: Repetition Differences. The first observation is that there is a high degree of variance for each execution result among the repetitions, irrespective of the tool, version, and rule type. This means that the generated tests yield different rule execution results in each repetition. This is caused by how *EvoMaster* generates (new) variable values of cancer messages and cancer cases: *randomly*. An effective test strategy should generate valid instead of random variable values, which leads to more rules being applied.

4.4.2 Observation 2: Tool Differences. EvoMaster-BB, again, is the tool that achieves the highest number of passes for both rule types. It reaches between 86.84% and 89.95% and between 0.93% and 1.72% for aggregation and validation rules, respectively. The best performing white-box tool overall is, again, *EvoMaster-WB-MIO* which passes for 57.25% to 81.79% (aggregation) and 0.07% to 0.39% (validation) of the rules. Both *EvoMaster-WB-MOSA* and *EvoMaster-WB-WTS* are inferior to *EvoMaster-WB-MIO* in terms of passes.

In terms of fails, the results are not as clear: *EvoMaster-WB-MIO* generates tests that can fail more aggregation rules, i.e., between 1.98% and 3.75%, than *EvoMaster-BB*, which fails for 1.70% to 3.12%. However, *EvoMaster-BB* fails considerably more validation rules (1.58% to 3.74%) than *EvoMaster-WB-MIO* (0.23% to 1.27%) and *EvoMaster-WB-MOSA* (0.02% to 1.53%). *EvoMaster-WB-MIO* is better than *EvoMaster-WB-MOSA* for some versions, whereas the opposite is true for other versions.

In terms of warnings, *EvoMaster-BB* yields more than all the *EvoMaster-WB* techniques with between 7.65% and 10.11% for aggregation rules, again followed by *EvoMaster-WB-MIO* with 3.79% to 7.80%. However, none of the techniques can execute warnings for validation rules.

4.4.3 Observation 3: Rule Execution Result Differences. We observe that depending on the rule type, the tests favor different execution results across all the tools and versions. For validation rules, fails are more common than passes (not considering warnings, as none are thrown). This is natural due to the random variable value generation of *EvoMaster*, which is the same reason why there is a high variance among identical repetitions. Nevertheless, the tools can still generate tests that pass many of the applied rules. For aggregation rules, we notice a different behavior, i.e., most of the rules pass, followed by warnings and fails.

4.4.4 Observation 4: Version Differences. Similar to RQ 3, we do not observe big differences across different versions for *EvoMaster-BB*. The standard deviation of the rules that pass is 1.00 and 0.30, fail is 0.41 and 0.66, and warning is 0.86 and 0 for aggregation and validation rules, respectively. The *EvoMaster-WB* tools only exhibit more variance across the different versions for aggregation rules that pass and yield warnings. Nevertheless, this strengthens our suggestion from RQ 3 that rule evolution is not a deciding factor for choosing one tool over another.

4.4.5 Observation 5: Comparison to Production GURI. Finally, we compare the rule execution results of the tools to those from production *GURI*, i.e., to real-world statistics of rule execution results. Table 5 depicts this for "Production" and compares it to all the four tools. The values are arithmetic means and standard deviations relative to the total number of rules. Note that we only consider
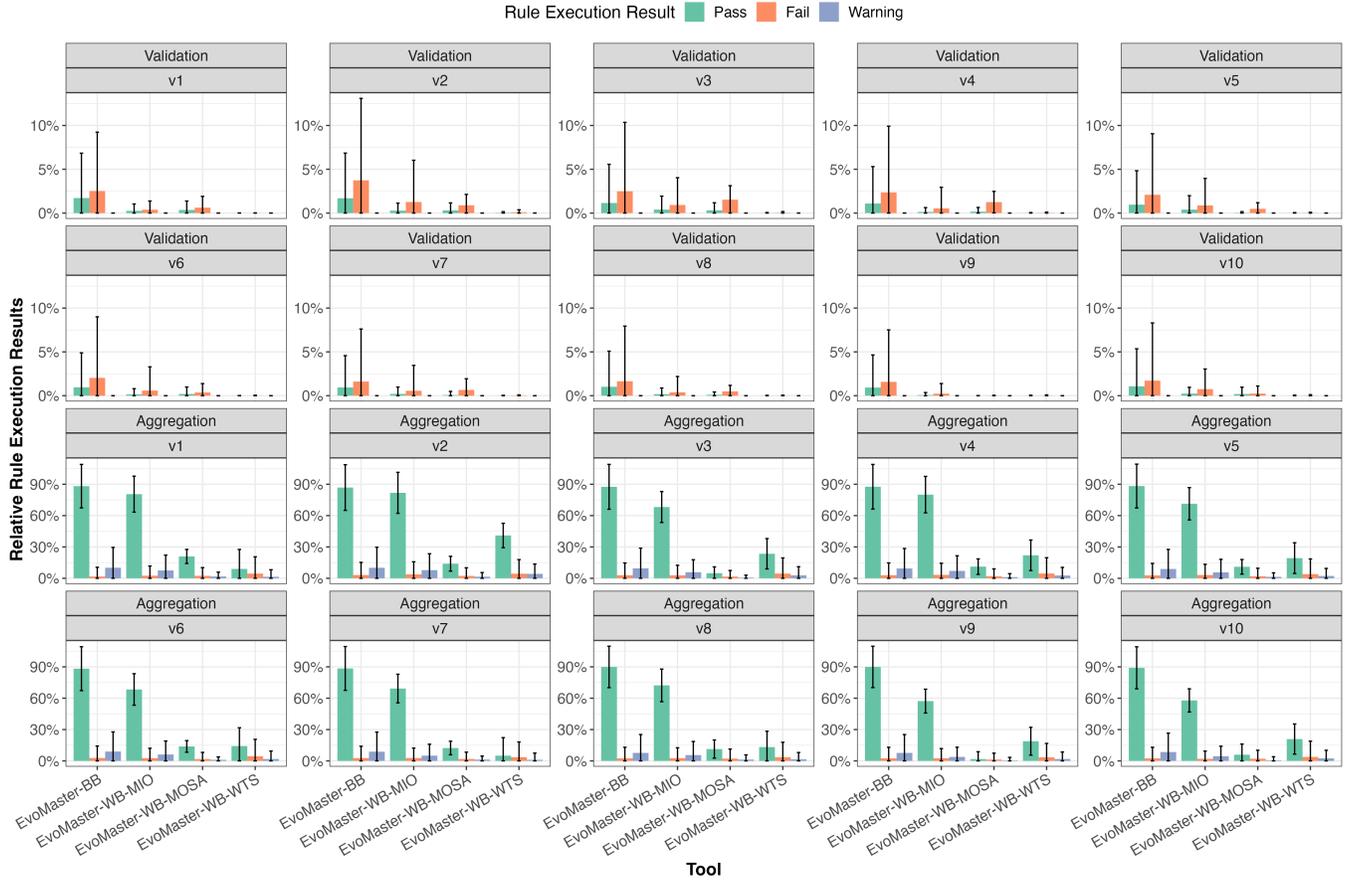
Figure 2: Rule execution results relative to the total number of rule executions for each rule, tool, and version. The bars are arithmetic means and the error bars are standard deviations.

Table 5: Rule execution results relative to the total number of rule executions for each rule and tool. The values are arithmetic means ± standard deviations summarized on rule type level across all the individual rules. *Production* corresponds to the rule executions from production *GURI*. The values for the test generation tools are for the latest version v10, averaged across all the repetitions.

Rule Type	Tool	Rule Execution Result				
		Pass	Fail	Warning	Not Applied	Not Executed
Validation	*Production*	26.19%±37.32	0.05%±0.15	$6.55 \times 10^{-6}\%\pm5.52 \times 10^{-5}$	73.75%±37.38	0%±0
	EvoMaster-BB	0.97%±3.92	1.61%±6.25	0%±0	97.42%±7.19	0%±0
	EvoMaster-WB-MIO	0.30%±0.88	1.41%±3.38	0%±0	93.29%±6.31	5%±5.04
	EvoMaster-WB-MOSA	0%±0	0%±0	0%±0	70.57%±20.14	29.43%±20.14
	EvoMaster-WB-WTS	0%±0	0%±0	0%±0	100%±0	0%±0
Aggregation	*Production*	99.88%±0.47	0.02%±0.08	0.10%±0.47	0%±0	0%±0
	EvoMaster-BB	89.02%±20.24	2.38%±10.65	8.59%±18.24	0%±0	0%±0
	EvoMaster-WB-MIO	64.47%±13.11	2.07%±7.63	6.02%±12.83	0%±0	27.44%±4.41
	EvoMaster-WB-MOSA	2.77%±10.61	1.92%±8.59	0.66%±4.32	0%±0	94.65%±16.38
	EvoMaster-WB-WTS	4.85%±18.59	3.86%±16.55	1.05%±6.89	0%±0	90.23%±29.40

the tool results for the latest version, i.e., v10, as the production results are only available for the current version.

We observe that the distributions of the tools are considerably different from production *GURI*. In production, most rules are not applied (73.75%), and of the applied ones, the vast majority passes (26.19%). Only a fraction fails, and even fewer yield a warning. The situation is even more extreme for aggregation rules: 99.88% of the rules pass, and only a negligible number fail or yield a warning. The closest tool to production, in terms of effectiveness, is *EvoMaster-BB*. However, none of the tools achieves a similar number of rules that pass for both rule types.

Interestingly, the tools often achieve higher failure and warning rates, as observed in production. In particular, *EvoMaster-BB* (1.61%) and *EvoMaster-WB-MIO* (1.41%) can fail more validation rules, and all four tools yield more failures and warnings for aggregation rules (1.92% to 3.86%).

We conclude that in terms of passing rules, *EvoMaster-BB* performs best, but there is much room for improvement to reach production *GURI* levels. In terms of failing and executing warnings, *EvoMaster-BB* performs best for validation and *EvoMaster-WB-WTS* for aggregation rules.

> **RQ 4 Summary:** *EvoMaster-BB* is the tool that yields the most passes, fails, and warnings among all the tools, except for failing aggregation rules where *EvoMaster-WB-MIO* is the preferred tool. For all the tools, the validation rules are the easiest to fail, and aggregation rules are the easiest to pass. Warnings can only be triggered for aggregation rules. Compared to production *GURI*, no tool can pass a similar amount of rules, but all are good at generating tests that lead to fails and warnings; *EvoMaster-BB* is the closest to production.

---

5 DISCUSSION AND LESSONS LEARNED

This section discusses the results and outlines lessons learned, which provide research opportunities.

5.1 Need for Domain-Specific Objectives, Targets, and Evaluation Metrics

From RQ 1 and RQ 2, we see that there are not a lot of differences among the tools; and from RQ 3 and RQ 4, we observe that the tools can lead to some rules being applied and obtain rule execution results: pass, fail, or warning. But, it is evident that the current tools do not support testing domain-specific targets well, i.e., they optimize for the "irrelevant" objectives, e.g., code coverage, and there is much room for improvement. Going forward, test generation tools require to (1) encode domain-specific objectives in their search, e.g., with added domain-specific search objectives (e.g., rule (result) count or distance metrics to applying rules [2]), or adding tests that reach unseen domain-specific targets to the archive (similar to Padhye et al. [38]); (2) keep tests after the search that cover each domain-specific target for regression testing scenarios; and (3) evaluate test generation tools with domain-specific metrics (rule execution status and rule execution results in our case) to show their effectiveness when traditional metrics do not show differences (also discussed by Böhme et al. [11]).

5.2 Oracle Problem for Rule Execution Results

The oracle problem is a well-known problem in software testing [10], which extends also to domain-specific goals. While we show that the current tools can apply rules with different results, it is unclear if, for a randomly generated test input, a rule is expected to pass, fail, yield a warning, or should not be applied. Current research does not offer a solution; there simply is no implicit oracle [10] for rule executions. Going forward, this needs to be addressed, and we see three potential aspects: (1) using tests that lead to a specific rule result (pass, fail, warning) and employ them in a regression testing setting, e.g., if a test passes a rule in version 1, it should also pass in version 2; (2) applying differential testing by comparing the outputs of the same random test input to a, e.g., reference implementation for the medical rules, as they should be standardized; and (3) devising metamorphic relations on the rules that are either semantics-preserving (similar to Lu et al. [35]) or are known to lead to invalid rules and comparing the outputs to the correct implementation.

5.3 Challenge of Generating Medical Data

Generating synthetic medical data is a challenge researchers from many fields are trying to tackle [17, 24]. In software testing, generating synthetic and valid medical data is equally important. The current test generation tools are good at generating syntax-compliant data (e.g., according to an OpenAPI schema definition). However, the individual variable values (i.e., medical variables) are randomly generated. This leads to many invalid cancer messages and cases to be checked by the rule engine. We identify four potential ways to address this challenge in the context of test generation in the future: (1) constrain the valid medical variables through, e.g., the OpenAPI schema definition, either manually or derived from documentation;[7] (2) use generative models such as generative adversarial networks (GANs) or variational autoencoders (VAEs) trained on real patient records to generate valid cancer messages and cases to directly test *GURI* [26]; (3) use generated cancer messages and cases in *EvoMaster* either as seeds or when new requests are sampled; and (4) employ large language models (LLMs), possibly trained on electronic health records (EHRs) or medical text [33, 40, 44, 45], to generate variable values.

5.4 Generality of the Results

While the results are specific to the case study, i.e., *GURI* at the CRN, the findings and lessons learned are likely applicable in other contexts. (1) Other countries also have medical registries similar to the CRN, which also deals with EHR. These can benefit from the challenges and findings outlined in this paper to introduce automated test generation tools. (2) Rule-based systems, in general, potentially face similar challenges. Once generated tests pass the input validation and execute rules, they will also have to deal with, e.g., domain-specific objectives, the oracle problem, and generating data that executes the rules. (3) Any system that dynamically loads targets of interest will likely also be affected by the need for domain-specific objectives and targets. (4) Researchers at a recent

---

[7] *EvoMaster* 1.5.0 does not support this; support was added in 1.6.0: https://github.com/EMResearch/EvoMaster/pull/709

Dagstuhl seminar[8] discussed similar challenges, such as domain-specific objectives and targets [13]; comparison to production [12]; domain-specific oracles such as reference implementations, differential and metamorphic testing [15]; and required evaluations that go beyond code coverage and errors [14]. This shows that while our results are specific to *GURI*, the challenges are important to the research community. Consequently, our paper is a valuable case study providing data for these challenges.

5.5 Call for Studies with Domain-Specific Goals

Based on our findings showing that code coverage and uncovered errors are insufficient to evaluate automated test generation tools, we conclude that there is a dire need for more industrial and public sector case studies like ours. As also outlined by Böhme et al. [11], code coverage is insufficient to validate test generation tools and fuzzers. For example, which tool is better when code coverage is equal, or no errors are found? Exactly this happens in the CRN's case. The research community needs to better understand domain-specific needs for test generation, objectives, and targets to optimize for, and evaluation metrics that are better aligned with stakeholders' interests; and, as a next step, evolve current tools to support these domain-specific needs better.

6 RELATED WORK
6.1 Test Generation for REST APIs

Many industrial applications, especially those built with the microservice architecture expose REST APIs. As a result, there is an increasing demand for automated testing of such REST APIs. Consequently, we can see a significant rise in publications in recent years [23]. Moreover, several open-source and industrial REST API testing tools are available such as *EvoMaster* [4–7], RESTler [9], RestTestGen [16], RESTest [37], Schemathesis [19], Dredd [18], Tcases [29], bBOXRT [31, 32], and APIFuzzer [3]. Even though any of these tools can be used in our context, we use *EvoMaster*, since it is open-source and has been shown to be the most effective regarding source code coverage and thrown errors among ten different tools in a recent study [28].

Generally, REST API testing approaches are classified into black-box (no source code access) and white-box (requires source code access) [23]. The existing literature has developed testing techniques from three main perspectives to evaluate testing effectiveness [23]: (1) coverage criteria, e.g., code coverage (e.g., branch coverage) and schema coverage (e.g., request input parameters); (2) fault detection, e.g., service errors (i.e, Hypertext Transfer Protocol (HTTP) status code 5XX) and REST API schema violations; and (3) performance metrics, i.e., related to the response time of REST API requests. To achieve these objectives, various algorithms have been developed in the literature. For instance, *EvoMaster* has implemented several EAs including random testing [5–7]. Various extensions have been also proposed to *EvoMaster*, such as handling sequences of REST API calls and their dependencies [49], handling database access through structured query language (SQL) [46], and testing remote procedure call (RPC)-based APIs [48]. In our case, we have access to the source code of *GURI*; therefore, we employ both the black-box

and the white-box (parameterized with three EAs) tools of *EvoMaster*. However, an extended investigation in the future may include other tools.

Compared to the literature, our main contribution is applying an open-source REST API testing tool in the real-world context of the CRN. We assess the tool's effectiveness in achieving code coverage, errors found, and domain-specific metrics (e.g., related to medical rules defined for validating and aggregating cancer messages and cancer cases).

6.2 Development and Testing of Cancer Registry Systems

In our recent paper with the CRN [30], we assess the current state of practice and identify challenges (e.g., test automation, testing evolution, and testing machine learning (ML) algorithms) when testing CRN's CaReSS. This paper is the first concrete step towards handling those challenges, particularly assessing the effectiveness of an existing testing tool in the CRN's context. Two other recent works build cyber-cyber physical twins for *GURI* [34] and incorporate ML classifiers into *EvoMaster* to reduce testing cost [27].

In the past, we developed a model-based engineering framework to support CaReSS at the CRN [42]. The framework aims to create high-level and abstract models to capture various rules, their validation, selection, and aggregation. The framework is implemented based on the Unified Modeling Language (UML) and Object Constraint Language (OCL), where the UML is used to capture domain concepts and the OCL is used to specify medical rules. The implementation of the framework has been incorporated inside *GURI*, which is the subject of testing in this paper. As a follow-up, we also developed an impact analysis approach focusing on capturing changes in rules and assessing their impact to facilitate a systematic evolution of rules [43]. Finally, we also developed a search-based approach to refactor such rules regarding their understandability and maintainability [35]. Compared to the existing works, this paper focuses on testing *GURI* as a first step toward building cost-effective testing techniques at the CRN.

7 CONCLUSIONS

This paper reports on an empirical study evaluating the test effectiveness of *EvoMaster*'s four testing tools on a real-world system, the Cancer Registry of Norway (CRN)'s Cancer Registration Support System (CaReSS). CaReSS is a complex software system that collects and processes cancer patients' data in Norway and produces statistics and data for its end users. Our results show that all the studied testing tools preform similarly regarding code coverage and errors reported across all the studied versions. However, in terms of domain-specific metrics, *EvoMaster*'s black-box tool is more effective; hence, we recommend it for the CRN as a starting point. We also provide lessons learned that are beneficial for researchers and practitioners.

---

[8]https://www.dagstuhl.de/23131 (the report has not been published at the time of writing)

REFERENCES

[1] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE 2017)*. Association for Computing Machinery (ACM), New York, NY, USA, 287–292. https://doi.org/10.1145/3030207.3030229

[2] Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C. Briand. 2013. Generating Test Data from OCL Constraints with Search Techniques. *IEEE Transactions on Software Engineering* 39, 10 (Oct. 2013), 1376–1402. https://doi.org/10.1109/tse.2013.17

[3] APIFuzzer. 2022. APIFuzzer – HTTP API Testing Framework. https://github.com/KissPeter/APIFuzzer Accessed 23.8.2023.

[4] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-Context Automated System Test Generation. In *Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST 2018)*. Institute of Electrical and Electronics Engineers (IEEE), 394–397. https://doi.org/10.1109/ICST.2018.00046

[5] Andrea Arcuri. 2018. Test Suite Generation with the Many Independent Objective (MIO) Algorithm. *Information and Software Technology* 104 (Dec. 2018), 195–206. https://doi.org/10.1016/j.infsof.2018.05.003

[6] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with Evo-Master. *ACM Transactions on Software Engineering and Methodology* 28, 1 (Feb. 2019), 1–37. https://doi.org/10.1145/3293455

[7] Andrea Arcuri. 2021. Automated Black- and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (May 2021), 72–78. https://doi.org/10.1109/MS.2020.3013820

[8] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. Association for Computing Machinery (ACM). https://doi.org/10.1145/1985793.1985795

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE 2019)*. Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/icse.2019.00083

[10] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525. https://doi.org/10.1109/tse.2014.2372785

[11] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*. Association for Computing Machinery (ACM), 1621–1633. https://doi.org/10.1145/3510003.3510230

[12] Marcel Böme. 2023. Tweet: Comparison to Production. https://twitter.com/mboehme_/status/1640743122681339905 Accessed 23.8.2023.

[13] Marcel Böme. 2023. Tweet: Domain-Specific Fuzzing. https://twitter.com/mboehme_/status/1640739828621795332 Accessed 23.8.2023.

[14] Marcel Böme. 2023. Tweet: Evaluating Fuzzers. https://twitter.com/mboehme_/status/1640365695211896837 Accessed 23.8.2023.

[15] Marcel Böme. 2023. Tweet: Oracles. https://twitter.com/mboehme_/status/1640705559879094272 Accessed 23.8.2023.

[16] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated Black-Box Testing of Nominal and Error Scenarios in RESTful APIs. *Software Testing, Verification and Reliability* 32, 5 (Jan. 2022). https://doi.org/10.1002/stvr.1808

[17] Fida K. Dankar and Mahmoud Ibrahim. 2021. Fake It Till You Make It: Guidelines for Effective Synthetic Data Generation. *Applied Sciences* 11, 5 (2021). https://doi.org/10.3390/app11052158

[18] Dredd. 2021. Dredd – HTTP API Testing Framework. https://dredd.org Accessed 23.8.2023.

[19] Dmitry Dygalo. 2023. Schemathesis: Property-Based Testing for API Schemas. https://schemathesis.readthedocs.io Accessed 23.8.2023.

[20] J Ferlay, M Ervik, F Lam, M Colombet, L Mery, M Piñeros, A Znaor, I Soerjomataram, and Bray Freddie. 2020. Global Cancer Observatory: Cancer Today. https://gco.iarc.fr/today

[21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. Association for Computing Machinery (ACM). https://doi.org/10.1145/2025113.2025179

[22] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (Feb. 2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[23] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. https://doi.org/10.48550/arXiv.2212.14604 arXiv:2212.14604 [cs.SE]

[24] A. Goncalves, P. Ray, B. Soper, J. Stevens, L. Coyle, and A. P. Sales. 2020. Generation and evaluation of synthetic patient data. *BMC Med Res Methodol* 20, 1 (2020), 108. https://doi.org/10.1186/s12874-020-00977-1 Goncalves, Andre Ray, Priyadip Soper, Braden Stevens, Jennifer Coyle, Linda Sales, Ana Paula eng England BMC

[25] Med Res Methodol. 2020 May 7;20(1):108. doi: 10.1186/s12874-020-00977-1..

[25] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery (ACM). https://doi.org/10.1145/3468264.3473922

[26] Mikel Hernandez, Gorka Epelde, Ane Alberdi, Rodrigo Cilla, and Debbie Rankin. 2022. Synthetic data generation for tabular health records: A systematic review. *Neurocomputing* 493 (2022), 28–45. https://doi.org/10.1016/j.neucom.2022.04.053

[27] Erblin Isaku, Hassan Sartaj, Christoph Laaber, Shaukat Ali, Tao Yue, Thomas Schwitalla, and Jan F. Nygård. 2023. Cost Reduction on Testing Evolving Cancer Registry System. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023)*. Institute of Electrical and Electronics Engineers (IEEE).

[28] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery (ACM), 289–301. https://doi.org/10.1145/3533767.3534401

[29] Kerry Kimbrough, Juglar, and Thibault Kruse. 2023. Tcases: A Model-Based Test Case Generator. https://github.com/Cornutum/tcases Accessed 23.8.2023.

[30] Christoph Laaber, Tao Yue, Shaukat Ali, Thomas Schwitalla, and Jan F. Nygård. 2023. Challenges of Testing an Evolving Cancer Registration Support System in Practice. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 2023)*. Institute of Electrical and Electronics Engineers (IEEE), 355–359. https://doi.org/10.1109/ICSE-Companion58688.2023.00102

[31] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access* 9 (Feb. 2021), 24738–24754. https://doi.org/10.1109/ACCESS.2021.3056505

[32] Nuno Laranjeiro, Carlos Francisco Fernandes Santos, and João Agnelo. 2022. EvoReFuzz – Evolutionary REST Fuzzer. https://git.dei.uc.pt/cnl/bBOXRT Accessed 23.8.2023.

[33] Valentin Liévin, Christoffer Egeberg Hother, and Ole Winther. 2023. Can large language models reason about medical questions? https://doi.org/10.48550/arXiv.2207.08143 arXiv:2207.08143

[34] Chengjie Lu, Qinghua Xu, Tao Yue, Shaukat Ali, Thomas Schwitalla, and Jan F. Nygård. 2023. EvoCLINICAL: Evolving Cyber-Cyber Digital Twin with Active Transfer Learning for Automated Cancer Registry System. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery (ACM), 11 pages. https://doi.org/10.1145/3611643.3613897

[35] Hong Lu, Shuai Wang, Tao Yue, Shaukat Ali, and Jan F. Nygård. 2019. Automated Refactoring of OCL Constraints with Search. *IEEE Transactions on Software Engineering* 45, 2 (Feb. 2019), 148–170. https://doi.org/10.1109/tse.2017.2774829

[36] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Transactions on Software Engineering and Methodology* 31, 3 (July 2022), 1–43. https://doi.org/10.1145/3491038

[37] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery (ACM), 682–685. https://doi.org/10.1145/3460319.3469082

[38] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360600

[39] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (Feb. 2018), 122–158. https://doi.org/10.1109/tse.2017.2663435

[40] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Scharli, Aakanksha Chowdhery, Philip Mansfield, Blaise Aguera y Arcas, Dale Webster, Greg S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. 2022. Large Language Models Encode Clinical Knowledge. https://doi.org/10.48550/arXiv.2212.13138 arXiv:2212.13138

[41] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 27, 3 (Oct. 2018), 1–51. https://doi.org/10.1145/3241743

[42] Shuai Wang, Hong Lu, Tao Yue, Shaukat Ali, and Jan Nygård. 2016. MBF4CR: A Model-Based Framework for Supporting an Automated Cancer Registry System. In *Proceedings of the 12th European Conference on Modelling Foundations and*

Applications (ECMFA 2016). Springer International Publishing, 191–204. https://doi.org/10.1007/978-3-319-42061-5_12

[43] Shuai Wang, Thomas Schwitalla, Tao Yue, Shaukat Ali, and Jan F. Nygård. 2017. RCIA: Automated Change Impact Analysis to Facilitate a Practical Cancer Registry System. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*. Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/icsme.2017.22

[44] Xi Yang, Aokun Chen, Nima PourNejatian, Hoo Chang Shin, Kaleb E. Smith, Christopher Parisien, Colin Compas, Cheryl Martin, Anthony B. Costa, Mona G. Flores, Ying Zhang, Tanja Magoc, Christopher A. Harle, Gloria Lipori, Duane A. Mitchell, William R. Hogan, Elizabeth A. Shenkman, Jiang Bian, and Yonghui Wu. 2022. A Large Language Model for Electronic Health Records. *npj Digital Medicine* 5, 1 (Dec. 2022). https://doi.org/10.1038/s41746-022-00742-2

[45] Li Yunxiang, Li Zihan, Zhang Kai, Dan Ruilong, and Zhang You. 2023. ChatDoctor: A Medical Chat Model Fine-Tuned on LLaMA Model using Medical Domain Knowledge. https://doi.org/10.48550/arXiv.2303.14070 arXiv:2303.14070

[46] Man Zhang and Andrea Arcuri. 2021. Enhancing Resource-Based Test Case Generation for RESTful APIs with SQL Handling. In *Proceedings of the 13th International Symposium on Search Based Software Engineering (SSBSE 2021)*. Springer, 103–117. https://doi.org/10.1007/978-3-030-88106-1_8

[47] Man Zhang and Andrea Arcuri. 2022. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology* 31, 1 (Jan. 2022), 1–52. https://doi.org/10.1145/3464940

[48] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology* (1–39 2023).

[49] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and Dependency Based Test Case Generation for RESTful Web Services. *Empirical Software Engineering* 26, 4 (June 2021). https://doi.org/10.1007/s10664-020-09937-1