Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments

Samar Mouchawrab, Lionel C. Briand, *Fellow*, *IEEE*, Yvan Labiche, *Member*, *IEEE*, and Massimiliano Di Penta, *Member*, *IEEE*

Abstract—A large number of research works have addressed the importance of models in software engineering. However, the adoption of model-based techniques in software organizations is limited since these models are perceived to be expensive and not necessarily cost-effective. Focusing on model-based testing, this paper reports on a series of controlled experiments. It investigates the impact of state machine testing on fault detection in class clusters and its cost when compared with structural testing. Based on previous work showing this is a good compromise in terms of cost and effectiveness, this paper focuses on a specific state-based technique: the round-trip paths coverage criterion. Round-trip paths testing is compared to structural testing, and it is investigated whether they are complementary. Results show that even when a state machine models the behavior of the cluster under test as accurately as possible, no significant difference between the fault detection effectiveness of the two test strategies is observed, while the two test strategies are significantly more effective when combined by augmenting state machine testing with structural testing. A qualitative analysis also investigates the reasons why test techniques do not detect certain faults and how the cost of state machine testing can be brought down.

Index Terms—State-based software testing, structural testing, controlled experiments, state machines.

1 Introduction

There is increasing interest [41] in model-driven development for object-oriented systems using, for example, the Unified Modeling Language (UML). In addition to being a key resource for designing object-oriented software and providing a means for communicating ideas among designers and customers, models are very useful to guide and automate the testing of object-oriented software. In particular, UML state machines are very useful to model the most complex and critical components in object-oriented software [33].

Other than being useful to support development, models can be used to support testing activities. Model-based testing has been assessed in a number of empirical studies and shown to be useful in systematically defining test strategies and criteria and deriving test cases and oracles [15], [19], [20], [23], [54], [56]. At the same time, a number of researchers conducted studies on the cost and effectiveness of conventional testing strategies: white box [31], [32], [40], [67] and black box testing strategies [15], [58], [68]. However, despite a growing number of studies [13], [15], [16], [19], [20], [23], [53], [54], more empirical evidence is required to assess the importance of models in improving testing cost-effectiveness, and to investigate how model-based testing can be combined or augmented with simpler, widely adopted testing techniques such as white box structural testing. Therefore, assessing the cost and fault detection effectiveness of testing techniques based on state machines and comparing them with simpler, code coverage-based techniques seems a logical investigation to undertake. Since the latter is a basic test practice automated by existing commercial tools, only a significant improvement in fault detection effectiveness or cost would justify the use of approaches based on state machines.

This paper focuses on the effectiveness of UML state machine testing when compared and augmented with white box, structural testing, which is deemed to be the most common, basic technique for testing (clusters of) components. The choice of UML as a language to model state machines is a practical one, as UML is becoming a widely known and applied standard in industry. Specifically, this paper focuses on a specific state machine test strategy, i.e., round-trip paths testing [12], an adaptation of the W-method [24] for UML state machines. The choice is driven by our previous work on the subject [20] showing that this is a reasonable compromise in terms of fault detection effectiveness and cost between cheap but inefficient criteria (e.g., all transitions) and efficient but expensive criteria (e.g., all transition pairs).

In this paper, based on controlled experiments involving trained participants, we perform both a quantitative analysis of differences in fault detection effectiveness and cost among test techniques and a qualitative analysis to understand the

- *S. Mouchawrab and Y. Labiche are with the Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive Ottawa, ON K1S5B6, Canada. E-mail: {samar, labiche}@sce.carleton.ca.*
- *L.C. Briand is with Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway. E-mail: briand@simula.no.*
- *M. Di Penta is with the Department of Engineering—RCOST, University of Sannio, Via Traiano, 1-82100 Benevento, Italy. E-mail: dipenta@unisannio.it.*

Manuscript received 18 Mar. 2009; revised 7 Oct. 2009; accepted 4 Jan. 2010; published online 9 Feb. 2010.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-03-0059.
Digital Object Identifier no. 10.1109/TSE.2010.32.

reasons for these differences and the variations observed across test drivers and component clusters. We aim to answer the following research questions:

- How does the fault detection effectiveness of test cases identified and manually generated by testers based on the state machine alone (functional testing) compare to that of test cases manually generated by testers based on the coverage analysis of source code (structural testing)?
- Are the sets of faults detected by state machine testing and structural testing techniques complementary? Does this suggest that somehow combining the two techniques is beneficial in terms of fault detection effectiveness? Given that test cases derived from state machines are available earlier, can state machine testing be effectively augmented, based on its code coverage analysis, with structural test cases?
- What are the different factors that impact the effectiveness of state machine testing?

As no analytical means can help obtain realistic, practically useful answers to these questions, we conducted a series of four controlled experiments—involving fully trained, undergraduate and graduate students—on three object-oriented class clusters[1] with a nontrivial state-dependent behavior modeled using UML state machines. Results show the following:

- Overall, techniques based on code coverage and state machines do not show practically significant differences in terms of fault detection effectiveness.
- The two techniques are, to a significant extent, complementary in terms of the faults they detect. When augmenting state machine testing with structural testing, significant improvements in fault detection can be observed.
- The real-time behavior of a class cluster negatively affects the effectiveness of both structural and state-based testing. This suggests that the techniques we used in the experiments need to be complemented with testing techniques specifically targeting real-time properties.

The remainder of the paper is organized as follows: Section 2 discusses the related literature. Section 3 provides a detailed description of the reported controlled experiments and Section 4 discusses the threats to validity. Section 5 presents the results, while Section 6 details the outcome of a qualitative analysis to understand the limitations of the state machine testing technique and the factors affecting the cost of developing test drivers. Finally, Section 7 concludes and summarizes the results.

2 Related Work

This section discusses related work on state-based and structural testing (Section 2.1), and empirical studies aimed at assessing the usefulness and the effectiveness of testing techniques (Section 2.2).

1. In the context of object-oriented unit testing, a class cluster is a set of related classes constituting a software system unit.

2.1 State-Based and Structural Testing

One of the earliest works on state-based testing is the work by Chow [24], who proposed the W-method for finite state machines (FSM). This method has been adapted to UML state machines by Binder [12] and renamed the round-trip paths (RTPs) strategy. In both techniques, the state model is traversed to construct a transition tree that includes all transitions in the state machine in such a way that traversing along a path stops whenever the state encountered is already present in the tree. When there are guard conditions on transitions and these guards are in a disjunctive form, then several transitions are warranted, one for each truth value combination that is sufficient to make the guard condition true.

Other state-based test criteria were proposed by Offutt et al. [56]: transition coverage, full predicate coverage, transition-pair coverage, and complete sequence. A case study was used to compare these criteria with a random selection of test cases. Results showed an improvement in fault detection when using the full-predicate coverage criterion, while transition coverage yielded a smaller number of test cases than random testing, with the same fault detection rate.

Additional testing strategies have been defined for state machines. Hong et al. [38] propose a technique to derive extended finite state machines (EFSMs) from state machines. The EFSM is then transformed into a flow graph modeling the control flow and data flow in the state machine, thus enabling the application of conventional control and data-flow analysis techniques. A modification of this method is described by Bogdanov and Holcombe [13] to address the compliance of an implementation of a system to its specification. This has been further extended to UML state machines with operations contracts defined in the Object Constraint Language (OCL) [18].

FSMs, EFSMs, and UML state machines have been widely used to model systems in various application domains, such as sequential circuits and communication protocols. The study in [43] provides a rather complete and clear review of the fundamental problems in finite state machines testing.

It is worth noting that any of the criteria discussed in the literature, including the W-method, specifies a test case as a (sub)sequence of transitions and states in the state machine. Though deriving those test case specifications can easily be automated, due to possibly complex guard conditions many sequences may turn out to be infeasible and identifying test (input) data for these test case specifications is, in practice, very difficult [17]. This problem is identical to the sensitization problem described by Beizer [10] for white box testing, unless the guard conditions are simple (see, for instance, [56]). This sensitization problem is undecidable, which explains why some approaches based on search-based optimization techniques,have been recently developed to derive test cases for object-oriented systems (e.g., [49]). For this reason, in this paper, we will focus on the assessment of state-based testing (and its comparison with structural, white box testing) when manually performed by testers.

2.2 Empirical Studies

From a more general standpoint, a growing number of empirical studies address the cost-effectiveness of testing strategies in various types of testing techniques: white box [31], [32], [67], black box [68], or model-based [6], [15], [20], [53], [60]. Many of these studies use the mutation strategy to seed faults and evaluate the fault detection effectiveness of the testing techniques. For instance, a simulation and analysis procedure [20] has been proposed and used to study the cost-effectiveness of four state machine coverage criteria, namely, all-transitions, all-transition-pairs, full-predicate [56], and round-trip paths [12]. Briand et al. [20] showed that the cost-effectiveness of these criteria depends to a significant extent on the characteristics of the state machine. For state machines labeled with numerous guard conditions, the round-trip paths strategy provides a good compromise between all-transitions and all-transition-pairs, the latter being far too expensive and the former rather ineffective.

An empirical study focusing on white box testing strategies was performed by Frankl and Weiss [31], where the all-uses and decision (all-edges) criteria were compared to each other and to the null criterion (random test suites). Results showed that all-uses was not always more effective than the decision and the null criteria, but in few cases where there was a large difference. In contrast, in cases where the decision criterion was more effective than the null criterion, the difference was small. The results of Frankl and Weiss were further confirmed by Hutchins et al. [40], whose experiments showed a better fault detection effectiveness of the all-uses criterion over the all-edges criterion, at the expense of larger test sets. The two techniques seem to be complementary in terms of the faults detected. This is yet another example of the benefit of combining different testing techniques on the overall fault detection effectiveness of test sets.

Pretschner et al. focused on the automatic generation of test cases on the grounds of symbolic execution with Constraint Logic Programming (CLP) [60]. The aim of the symbolic execution of the model is to find an execution trace—and therefore a test case—that leads to the state to be tested. A number of strategies are used to optimize the traversal of the state machine. For instance, a fitness function is defined to generate the shortest path to the destination state. Other strategies include attributing probabilities to transitions or storing visited states and transitions to prevent repeated visits. The study aims at comparing the use of behavioral models, namely, EFSM, to handcrafted tests generated based on requirements' message sequence charts (MSCs). Results show that the use of models significantly increases the number of detected requirement errors. However, the number of detected programming errors was unrelated to the use of models. Handcrafted, model-based tests were shown to detect as many errors as automatically generated tests, but the two sets of tests detect partially different sets of faults [60]. Our work differs in that it compares a more widely applied state machine testing technique (RTP) to a practically common and widely used structural testing technique (nodes and edge coverage). It does so by performing replicated, controlled experiments involving human subjects that aim at precisely understanding the limits of each technique and how they complement each other.

An approach in model-based testing was proposed and validated by Nebut et al. [53]. The approach consists in automating the generation of system test scenarios from use cases in the context of object-oriented embedded software. By using contracts with UML use cases, the authors apply Meyer's Design by Contract approach [50] at the requirement level. Executable contracts written in terms of logical expressions allow for defining valid sequences of use cases and extracting relevant paths, which are called test objectives. An empirical evaluation of the proposed approach was executed on three small case studies (800 LOC to 2,000 LOC) to assess the efficiency of the generated test cases in terms of statement coverage. The results showed that most of the functional statements in the code are covered by the proposed technique with relatively small sets of test cases.

Briand et al. [15] focused on the cost-effectiveness of the RTP technique [12]. They investigate, in controlled experiment settings, the fault detection effectiveness of state-based (RTP) testing for classes or class clusters modeled with state machines. They also investigate how to augment RTP with a well-known black box testing technique: category-partition (CP) [58], though this part of the study is very limited. Results showed that the RTP technique significantly benefits from CP in terms of fault detection. These results were one of the motivations for our study. The limitations of the RTP technique incited us to further identify the factors that affect its fault detection effectiveness. However, in contrast to Briand et al. [15], we choose to compare and complement the RTP technique with a structural testing technique rather than with a black box testing technique. Furthermore, whereas CP was applied to augment RTP on a small subset of methods, here, structural testing is fully applied to the same extent as RTP, compared, and combined. It is expected that structural testing can be helpful at better exercising those parts of the cluster under test that were not (sufficiently) tested by state-based testing and that can be identified by analyzing its code coverage. This is of practical importance as state models are rarely complete and fully defined in practice. Briand et al. [15] also noticed the significant difference in terms of fault detection and cost between two oracle strategies, one using precise oracles checking the concrete state of objects (i.e., checking all attribute values), and the other based on state invariants. We also address this issue and suggest ways to limit the cost of oracles without affecting their fault detection effectiveness [51].

3 Experiment Description

This section reports, following a specific template [69], the definition and planning of the experiments we performed.

3.1 Experiment Definition and Context

The goal of this study is to analyze the state machine-based, round-trip path testing strategy and the edge coverage structural testing technique for the purpose of comparing and assessing them, as well as their combination with respect to their fault detection effectiveness and cost from the point of view of the tester. The context consists of objects, i.e., source code and UML state machines of three Java software clusters, and participants, i.e., undergraduate students from the fourth year of software engineering at Carleton University, Canada, and graduate students from the Master in Software Technology of the University of Sannio, Italy.

TABLE 1
Size of Source Code and State Machines

	Cruise Control	OrdSet	Elevator
# classes	4	1	8
# operations	34	23	74
# attributes	14	5	37
# LOC	358	393	581
# control flow statements	33	36	56
# nodes	106	111	241
# edges	103	126	214
# transitions	17	22	50
# states	5	5	6
# events	7	5	10
# guard conditions	0	17	19

For the sake of brevity, we will refer to state machine testing (drivers) using the round-trip path strategy as state testing (drivers). The same applies to edge coverage structural testing (drivers), which is simply referred to as structural testing (drivers).

The study was conducted as a series of four experiments: two conducted at Carleton University, Canada, and two at the University of Sannio, Italy, since replication is imperative to increase the credibility of results and to allow more robust conclusions to be drawn. The experiments involved three Java class clusters that all have a state-driven behavior:

1. `OrdSet` is a Java class (of 393 lines of code—LOC) that was included in the original experiment and its first replication. Each instance of `OrdSet` represents a bounded, ordered set of integers. The `OrdSet` class provides methods for adding a single element, removing a single element and creating the union of two ordered sets.
2. `Cruise Control` is a cluster of four Java classes (of 358 LOC). It simulates a car engine and its cruising controller.
3. `Elevator` is a cluster of eight Java classes (of 581 LOC) that was included in the last two replications only as a way to make our results less dependent on the first two clusters. It consists of a number of elevators servicing a number of floors. An elevator accepts stop requests to travel to other floors. Users can also request service from floors to go up or down.

The above three class clusters were extracted from software engineering students' final year projects, where teams of students follow a rigorous, UML-based, development strategy. Requirements of the final projects were identified and described in use case diagrams. Other UML artifacts were also used to model the systems, including class diagrams, collaboration diagrams, activity diagrams, and state machine diagrams. The latter was used to model state behavior. These clusters represent two typical cases, where a state machine is used to model the behavior of a complex data structure (`OrdSet`) and a state-dependent control class in a real-time multithreaded control cluster (`Cruise Control` and `Elevator`). The implementations and corresponding state machines were simplified in order to give participants sufficient time for testing them within the duration of laboratory sessions. Source code and models of the three clusters used in this study can be found on the Software-artifact Infrastructure Repository (SIR) [4]. The main cluster characteristics are summarized in Table 1.

Though `OrdSet` is composed of only one class, one can note that its state machine and control flow are more complex than those of `Cruise Control`. Furthermore, the guard conditions in the `OrdSet` state machine add to the complexity of the class, whereas `Cruise Control` is event driven only. `Elevator` is far more complex than the other two clusters.

The choice of these three clusters was in part based on the fact that each state machine was created at a different level of abstraction. While the `Cruise Control`'s state machine is simple (no guard conditions, nor complex actions), it is restricted to the state behavior of the cluster without modeling its real-time behavior. The real-time behavior is implemented in two algorithms in the code. The first represents the relation between time and class attributes such as speed and distance. The second implements a relation between car throttle, time, and cruise control speed to control car speed while in the cruising state. The state machine therefore models the `Cruise Control`'s behavior at a high level of abstraction without modeling algorithms managing the cluster's real-time behavior and speed control. On the other hand, the `OrdSet`'s state machine is more complex than the `Cruise Control`'s state machine. It models the different functionalities of the system at a relatively low level of abstraction: All of the functionalities (algorithms) are specified under the form of states, transitions, guards, and actions (pre and postconditions). The third cluster, `Elevator`, has a rather complex state machine at a very low level of abstraction: Its real-time behavior is much simpler than `Cruise Control`'s and can be partly specified in the state machine; the `Elevator`'s state machine does not model all of the cluster's functionalities, for instance, avoiding the specification of interactions between different instances of elevators (e.g., the optimization algorithm for choosing the best

TABLE 2
Research Questions

Number	Research Question
RQ1	What is the difference, in terms of fault detection effectiveness, between test cases manually generated from state machines (Ts) and test cases manually generated based only on node and edge coverage of the source code control flow (Tc)?
RQ2	Are there interaction effects regarding fault detection effectiveness between code coverage, learning effects, subject ability and software properties (code, state machine properties) and the test technique applied?
RQ3	Are state testing and structural testing complementary in terms of fault detection?
RQ4	How does the cost between state testing and structural testing compare?
RQ5	How is the test cost and fault detection effectiveness affected by augmenting state testing with structural testing?

elevator to service a floor) and not differentiating between moving directions (up and down). In our experience, such variations in levels of abstraction are common in modeling practice, as a trade-off between modeling cost and completeness must be achieved.

The source code used in these experiments is admittedly small. However, it is important to note that state machines in standard UML-based development are mostly used to model the behavior of complex classes or class clusters [21], [33], [42], particularly complex data structures (usually referred to as entity classes) and control classes, for example, in reactive systems. They are rarely used to model the entire system, as this would result in large and unmanageable models for software engineers and testers, or in simplistic, incomplete models which do not adequately capture the system behavior. Furthermore, even when the source code is small, the state behavior can be quite complex when measured in terms of states, events, and transitions. This issue is further discussed in Section 4.

3.2 Experiment Planning

This section details the experimental plan, describing the context, the research questions, the variables, and the design.

3.2.1 Context Selection

The participants involved in the four experiments had the following characteristics:

- First and third experiments (Carleton 1 and Carleton 2): Participants are fourth-year students from a specialized, software engineering bachelor's program. They were well versed in Java and UML and were attending a course on software testing that covers different white box and black box testing techniques with a focus on object-oriented software. The experiments were conducted during the lab hours of that course as part of practical lab exercises. Forty-eight students participated in the first experiment and 19 in the third.
- Second and fourth experiments (Sannio 1 and Sannio 2): participants are graduate students studying for a master's in software technology. Master participants (about 30 every year) are selected from a population of 300 graduate students in computer science and computer engineering. The participants were students attending an intensive course on software testing. Their experience with software testing before attending the course varied from no experience to some experience with JUnit. Twenty-five students participated in the second experiment and 19 in the fourth one.

The method for the selection of participants follows a stratified random sampling[2]; participants were first assigned to blocks based, for the Carleton Experiments, on their background and knowledge of object-oriented design and development techniques,[3] and for the Sannio experiments, on *laurea* graduation score (since participants were graduate students). Then, participants were randomly selected from the different blocks to form four groups with a similar block distribution to ensure the results would not be affected by random variations in subject experience across groups. In addition, groups were defined to be of similar sizes to ensure a balanced contribution of test techniques/clusters combinations to the results.

3.2.2 Research Questions

Table 2 details the research questions addressing the goal listed in Section 3.1. The fault detection effectiveness of both state and structural test techniques is addressed in research question RQ1. Answering RQ2 helps us to identify factors that have an interaction effect with the test technique on

2. Stratified random sampling: The population is divided into a number of groups or strata with a known distribution between the groups. Random sampling is then applied within the strata [44].

3. The background and knowledge regarding object-oriented design and development techniques of the different subjects was assessed based on their grades in two advanced courses in software engineering and object-oriented design.

fault detection effectiveness. RQ3 investigates how complementary the two techniques are in terms of fault detection. Answers to RQ4 are used to compare test techniques with respect to their cost, both individually and when combined. RQ5 focuses on assessing the cost-effectiveness of augmenting state testing with structural testing. The reason why we do not investigate the reverse option, i.e., augmenting structural testing with state testing, is that state machine test cases can be derived earlier from design models, and therefore are available when the source code becomes available. In addition, we will investigate the reasons why test techniques differ across class clusters in terms of fault detection and cost.

3.2.3 Variable Selection

At a high level, our dependent variables are based on the following constructs: 1) fault detection effectiveness, overall and across different fault types; 2) cost for both test specification and execution. There is one independent variable of interest (treatment): the type of artifacts used as a basis for testing (i.e., state machine or code structure). Also, a number of other variables (cofactors) are accounted for to determine whether they interact with the effect of our independent variable: code coverage, learning effects, and subject ability.

3.2.4 Mutant Seeding

To evaluate the fault detection effectiveness of the experiment techniques, we executed the different drivers delivered by the experiment participants on a number of mutant programs (or mutants), i.e., versions of the program under test where one fault was seeded using a mutation operator [45], [46], and used different kinds of oracles (Section 3.2.5) to compare the behavior of the original program with those of mutants. Whenever the oracle indicates a difference, the mutant is considered killed. The fault detection effectiveness of a driver is measured by the percentage of mutants killed. The mutants are automatically generated using MuJava [47], which allows us to quickly generate a large number of faults, thus facilitating statistical data analysis [5].

One issue to be addressed is the detection of equivalent mutants, i.e., mutants that have the same behavior as the original program and therefore cannot be killed by test cases. There are studies proposing techniques to automate the detection of equivalent mutants [55], [57], and a commonly used heuristic is to consider live mutants not killed by any test case in the overall test pool as equivalent mutants [8], [27]. In our study, we cannot simply assume that mutants not detected by any driver are equivalent, as we know the testing performed by our experiment participants is incomplete (due to time limitations) and unlikely to kill all nonequivalent mutants. Therefore, we do not attempt to discard nondetected mutants as equivalent mutants but present our results based on all mutants and then perform a manual, qualitative analysis of all undetected faults to assess the potential impact of equivalent mutants on the fault detection effectiveness results.

3.2.5 Experimental Procedure

When state machines are used, participants are expected to generate test sets based on the RTP testing technique [12], a common state testing strategy that can scale up to large state machines but that is more demanding than simply covering all transitions. A state machine would be represented as a tree graph called a transition tree, which includes (in a piecewise manner) all of the transition sequences (paths) that begin and end with the same state, as well as simple paths (i.e., sequences of transitions that contain only one iteration for any loop present in the state machine) from the initial state to the final state. A procedure based on a breadth-first traversal of the state machine is used for deriving the transition tree. The Round-Trip Path testing technique corresponds to covering all paths from the start node to the leaf nodes in a transition tree. Since different transition trees can be generated from a state machine, we wanted to avoid having our results affected by variations due to alternative transition trees or trees possibly wrongly constructed by participants. In practice, such trees would be generated automatically from state machines using a dedicated tool. For these reasons, one tree per cluster was provided to all participants working with the state machine model.

Participants working with state machines were asked to manually generate test cases covering round-trip paths in the provided transition tree. They were also asked to use state invariants in their oracle assertions. After executing a transition, an oracle assertion checks the new cluster state with the expected state invariant. In replicated experiments, in addition to state invariants, participants were asked to implement contract assertions in their oracles. Contract assertions include class invariant, and methods' preconditions and postconditions. The different invariants and contracts are provided in OCL [66] in the documentation provided to participants.

For structural testing, participants were told to attempt covering all blocks (nodes, statements) and edges in the methods' control-flow graphs of the original (not mutated) source code. This is a common practice when testing classes and it is therefore a realistic baseline of comparison for state testing. By running their drivers on the instrumented code, the participants identify noncovered nodes and edges. This guides participants to identify new test cases to be added to their drivers to improve structural coverage. Participants using structural testing were advised to write oracles checking expected output/attribute values against actual ones.

3.2.6 Experiment Design

To avoid learning and fatigue effects or the specific class clusters having a confounding effect with our experimental treatment, each subject group performed the experiment in two separate labs with a different class cluster under test and a different treatment. Table 3 shows the distribution of treatments among groups of participants. Each treatment is executed by two different groups of participants, in the first or the second laboratory (lab order). As a result, each group executed different combinations of treatment and class cluster in each lab. Such an experimental design is standard and is referred to as a balanced $2 * 2$ factorial design [69].

Test drivers submitted by the participants were executed offline on a set of mutant programs (Section 3.2.4) to measure their mutation scores. Test drivers were also executed on an instrumented version of the original code

TABLE 3
Distribution of Experiment Treatments among Groups

	Group 1	Group 2	Group 3	Group 4
Lab 1	Cruise Control + State machine	OrdSet + State machine	Cruise Control + Code	OrdSet + Code
Lab 2	OrdSet + Code	Cruise Control + Code	OrdSet + State machine	Cruise Control + State machine

of the software under test to collect node and edge coverage data. The development and data collection were done on the Eclipse 3.0 platform [1]. Two Eclipse plug-ins, the Eclipse Test and Performance Tools Platform (TPTP) project [3] and the Eclipse Metrics plug-in [2], are used to collect cost-related data.

From a general standpoint, the notion of cost in the context of testing is complex and can be related to many factors, such as test suite size, test case identification complexity, CPU user time usage, and time-to-market. In our context, we focus on generation and execution cost and, in particular, we rely on the number of method calls measured by executing instrumented test drivers, on the CPU user time needed by the test driver execution, and on the size of the test drivers measured in LOC. Although these are clearly surrogate measures, number of calls [6], [11], [20], [40], [67] and CPU time [15] have been used in a number of testing studies. In these studies, one test case often corresponds to one execution of a function/program (e.g., [40]). This corresponds in our study to one execution of a method in the class cluster under test.

3.3 Experiment Operation

3.3.1 Preparation and Material

To prepare the students for the different tasks required for the experiment—and thus make the experiment realistic in terms of human factors—the experiment was preceded by a refresher course on the basics of software testing (e.g., test cases, testing criteria, and test drivers), structural and functional testing, and class testing. Students applied the concepts and techniques they were taught in assignments and laboratory exercises prior to the start of the experiment's tasks.

The following documents were prepared and provided to all participants in all groups:

1. printed list of instructions to guide students through the different tasks to complete,
2. high-level description of the class cluster,
3. Eclipse tutorial, and
4. driver template (with slight variations, depending on the testing strategy).

Depending on the treatment (testing strategy), participants were provided with the following:

- *Source code (for participants using white box testing)*. To allow participants to easily compute node and edge coverage, we provided them with an instrumented version of the code—packaged in a jar file—along with the original, noninstrumented, source code.
- *State machine diagrams (for participants using state testing)*, plus
  - class public interfaces,
  - a transition tree,
  - class diagrams, operations' contracts and state invariants in OCL [66], and
  - an executable jar file of the class cluster containing byte code only, and not the source code, as we want to ensure pure model-based testing.

3.3.2 Execution

In an experimental, artificial setting, the time allocated to an experiment is necessarily limited. Test techniques are compared assuming a limited, equal effort on the part of testers. Although this does not allow the assessment of the technique's maximum fault detection effectiveness, it is necessary to perform a fair comparison among participants and to investigate the effectiveness of the techniques when applied under time constraints [14], a realistic scenario in industrial development environments. In our experiments, each lab lasted three hours for Carleton's experiments and four hours for Sannio's experiments.

Right before the lab, students were introduced to the class clusters to be tested during the experiment to make sure that they relied solely on the documentation provided as part of the experimental material. Also, we explained the tasks to be performed during the experiment.

During each lab, students were first asked to read the documentation of the class cluster to understand the functionality it provides; then they were asked to write driver code, following precise instructions, by identifying test cases based either on the provided transition tree (covering round-trip paths) or based on structural coverage criteria, depending on the group to which they belonged.

After completing their task, the participants were asked to answer and return a survey questionnaire. The questionnaire addresses three areas: the tasks implemented, the testing technique used, and the work environment. Due to space constraints, details about the survey questionnaire results are provided in a technical report [51].

3.3.3 Data Collection

After the experiment, the test drivers produced by participants were executed on the original code of the class clusters to inspect their correctness and to eliminate inadequate drivers, which could not be used for the experiment analysis, e.g., state drivers that did not implement RTPs or did not have oracle implementations. The latter was necessary as it was impossible for us to ensure a

TABLE 4
List of Changes across Experiments

	Experiment			
Design or context attribute	**Carleton 1**	**Sannio 1**	**Carleton 2**	**Sannio 2**
Cluster under test in addition to Cruise Control	OrdSet	OrdSet	Elevator	Elevator
Participants level	Undergrad	Graduate	Undergrad	Graduate
State invariants in oracles (state testing)	Yes	Yes	Yes	Yes
Contract assertions in oracles (state testing)	No	Yes	Yes	Yes
Lab time length	3 hrs	4 hrs	3 hrs	4 hrs
Common documentation (for both techniques)	During lab	Before lab	During lab	Before lab
Use case diagrams added to documentation	No	Yes	Yes	Yes
Including oracle helper class to template driver	No	Yes	Yes	Yes
Implicit Sneak path testing (Cruise Control)	No	Yes	Yes	Yes

satisfactory compliance of the participants with the lab task instructions.

Perl scripts were used to automatically execute drivers on mutants and on code instrumented versions in order to collect data required to compute mutation scores, node and edge coverage, and distributions of undetected fault types. The Eclipse TPTP Project [3] plug-in was used to measure the number of method calls when executing test drivers. The Eclipse Metrics plug-in [2] was used for statically counting the number of statements in collected drivers.

3.4 Experiment Replication

There are many reasons why replications are necessary in experimentation, and in particular, for software engineering experiments [61]. In our context, replications were useful 1) to address or mitigate threats to validity discovered in the first experiment, 2) to use an additional class cluster, thus strengthening the external validity of our results, and 3) to collect results from at least two distinct geographical locations and organizational settings, once again improving external validity. Though the experimental design (shown in Table 3) did not change across replications, other changes are summarized in Table 4 and motivated below.

Using different objects. In both the Carleton 2 and Sannio 2 experiments, we replaced `OrdSet` with `Elevator`, which is a control class cluster (eight classes) for an elevator control system. This was motivated by the first two experiments, Carleton 1 and Sannio 1, which showed clear limitations for state testing when applied to real-time control classes (`Cruise Control`). Therefore, we wanted to further explore the matter on another, more complex instance of real-time class cluster to make sure that the limitations we had observed were not specific to `Cruise Control` and also investigate how increased complexity would affect the results.

Extending the RTP criterion to cover sneak paths. Our qualitative analysis of undetected faults in Carleton 1 (discussed in Section 6) pointed to a deficiency in the state testing technique. In its original form, as is common practice, `Cruise Control`'s state machine did not include implicit transitions (self-transitions with no actions, named sneak path in [12]), which are not taken into consideration by the RTP criterion. As a result, many faults remained undetected. Therefore, in all three replications, the testing of sneak paths was made part of our state test strategy in addition to round-trip paths.

Use of different oracles. In the last three experiments (Sannio 1, Carleton 2, and Sannio 2), participants working with state testing were instructed to use contract assertions in addition to state invariant assertions in their oracles. This is because a qualitative analysis of undetected faults (Section 6) from Carleton 1's results suggested that the use of contract assertions in oracles would have led to the detection of a significant number of these faults (the average mutation score of OrdSet drivers went from 50 percent in Carleton 1 to 72 percent in Sannio 1).

Providing support to write oracles. In Carleton 1, we noticed that the implementation of state invariants (from OCL to Java code) took considerable lab time. Therefore, to give participants more time to implement their drivers, we provided them with an oracle class helper, including methods for checking a class invariant, a state invariant, preconditions, and postconditions for methods. Participants working with structural testing were provided with an implementation of an oracle method comparing all attributes values against expected ones.

Using longer labs. Since survey questionnaires [51] from the Carleton 1 experiment suggested a lack of time to perform the experimental tasks, whenever possible, i.e., for the Sannio 1 and Sannio 2 experiments, we used laboratories of four hours instead of three. In addition, class clusters were introduced in the morning, before the four laboratory hours started, so that participants had four hours to be entirely dedicated to the experimental tasks.

In summary, given the replication classification provided in [9], our replications fall into the following categories: 1) Replications that do not vary any research hypothesis but that vary the experiment procedure and material (e.g., changing lab duration and class clusters) and 2) replications that vary research hypotheses by varying variables intrinsic to the object of study (e.g., contract assertions in oracles and sneak path testing).

3.5 Overview of Statistical Analysis

We provide here a short overview of the statistical techniques we applied to address our research questions. Univariate analysis was performed to assess the isolated effect of an independent variable on a dependent variable. In particular, two-sample *t*-tests were performed to compare test techniques in terms of fault detection effectiveness and cost, and determine whether differences in means could be due to chance. The level of significance is set to $\alpha = 0.05$ for all tests, though we also report p-values.

To avoid potential threats due to the violation of the *t*-test assumptions, equivalent nonparametric tests (Wilcoxon rank sum tests [28]) were also performed to verify that negative results are not due to strong departures from the normality assumptions, which are known to make t-tests conservative. In the rare cases where different results are observed, this is clearly stated. The *t*-test is a parametric test and requires data to be normally distributed. We use the Anderson-Darling test to check for normality [30] (H_0: *samples significantly deviate from a normal distribution*) to verify the normality of the data studied. We report normality results whenever samples deviate significantly from the normal distribution. Fortunately, in experiments where a significant deviation from normality was found, the number of participants (and thus, data points) is large enough (48 and 25) to make parametric tests robust to the encountered normality deviation [62].

Performing multiple tests on the collected sample data to address a number of distinct hypotheses may introduce chance capitalization of type I error. A number of statistical corrections were proposed to address this issue, among them the Bonferroni correction [34]. In our context, we apply the Holm's procedure [35] to the data pooled from all four experiments. It is suitable to small sample sizes (as in our case), is less restrictive than Bonferroni, and accounts for the correlation between tests. Specifically, it sorts p-values in ascending order, resulting from the n distinct tests, and multiplies the smallest p-value by n, the next by $n - 1$, and so on. Finally, to be compared to the significance level, given the resulting ordering index i, p-values are corrected as follows: $p_i = max_{\mathrm{j} \leq \mathrm{i}}(p_{\mathrm{j}})$.

When dealing with small sample sizes, it is also interesting to look at the *power* of statistical tests. The power of a statistical test is the probability that the test will reject a false null hypothesis (that it will not make a Type II error) and is function of three factors: the specific statistical test used (e.g., *t*-test), sample sizes, sample standard deviations for the different treatments, and the selected effect size[4] [52]. There are several potential applications for power analysis [29], among others to determine an appropriate sample size for an experiment. However, in our case, this is not possible as the sample size is dictated, as often in our field, by the availability of participants. Instead, we used the power analysis for a different purpose: to determine the minimum effect size above which we achieve a certain power, typically 80 percent, i.e., above which we can be confident about our conclusions. In other words, if we do not observe any statistically significant result for small sample sizes, we cannot confidently claim there is no effect of the treatment. We can, on the other hand, be confident that there is no effect above a certain effect size for which the power of our test reaches 80 percent or more.

To visualize distributions and allow for the comparison of results, we use box plots showing selected quantiles of continuous distributions and extreme values. In particular, boxes span between the 25th and 7th percentiles and the line in between the box indicates the median. The dashed lines, sometimes called *whiskers*, are placed at a distance equal to 1.5 times the interquartile distance below the 25th percentile and above the 75th percentile, respectively.

The analysis of cofactor effects, and their interaction with the treatments, is done using a two-way analysis of variance (ANOVA) or a bivariate least-squares regression [28].

4 Threats to Validity

This section discusses the main threats to validity [69] that can affect our experiments.

4.1 Conclusion Validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of the experiment [69]. In our experiments, threats to conclusion validity could be essentially due to low statistical power, resulting from the relatively small number of participants. We were limited by the number of students enrolled in the testing courses within which we conducted the experiments. To limit the impact of this threat on our conclusions, we designed the experiment in such a way that each group would work on a different treatment for two subsequent labs, and thus doubled the number of observations. Replicating the experiment four times also increased our capability to identify significant results. In addition, we performed a power analysis (fault detection analysis, item 3, in Section 5.1.1) with the aim of determining the minimum effect size that could be observed with 80 percent power.

Statistical conclusions were supported by proper tests, in particular, *t*-tests for pairwise comparisons and ANOVA for the analysis of cofactors. As discussed in Section 5.1.1, results of a *t*-test were also confirmed by an equivalent, nonparametric test (Wilcoxon) when required. To account for possible Type I error chance capitalization due to repeated significance testing, the significance level of the different *t*-tests was adjusted using Holm's procedure [35], which is less conservative than Bonferroni adjustment.

4.2 Internal Validity

An internal validity threat exists when the outcome of the experiment may not necessarily be caused by the treatment applied, but can be caused by another factor not controlled in the experiment. One example of internal validity threats is the learning and fatigue effects that can occur during experiments. This threat is addressed in our experiment by using different treatments and different class clusters in

4. Effect size is a measure of the strength of the relationship between two variables. It is useful to know not only whether an experiment has a statistically significant effect but also the size of any observed effects. Cohen's *d* [25] is an appropriate effect size measure to use in the context of a *t*-test on means, where *d* is defined as the difference between two means divided by the pooled standard deviation for those means.

each of the labs conducted for each group, and doing so in an order that prevents fatigue or learning effects being conpounded with our treatment.

To tackle the selection threat that is related to the variation in human performance, we identified a number of blocks to which the students were assigned. These blocks are based on marks achieved in earlier courses on software engineering and design. Students were selected from the different blocks to obtain a stratified random sampling over the different groups [69].

Another internal validity threat, the diffusion or imitation of treatments, was also limited by monitoring the labs and preventing access to the experiment material outside the lab hours and by other groups' members. Note that the experiment material was accessed through the course Website only during lab hours, with an address only known during the lab by members of the group working in that specific lab.

As opposed to participants working with structural testing, those working with state machines were not instructed to use equivalence class partitioning or boundary analysis to identify test cases parameters values. This was not necessary as, in this experiment, no boundary analysis was needed: The `Cruise Control`'s state machine has no parameters, and boundary analysis of the `OrdSet` and the `Elevator` clusters was already accounted for when exercising the guard conditions.

The allocated time for the experiments was fixed and limited: This sometimes caused limited code and state machine coverage. This is usually unavoidable in the context of controlled experiments in artificial settings. We address this issue in the data analysis by using coverage (state machine and code) as an interaction factor. From a more general standpoint, it is often the case that, in controlled experiments, experimenters should choose between assessing the impact of a treatment on either the time to perform the tasks or their effectiveness within specific time constraints, but it is usually impractical to address both [9]. The work presented here matches the latter case.

Although participants were aware of the laboratory objectives, they did not know exactly what hypotheses were tested. Evaluation apprehension is also avoided since participants were told that their labs were marked based on the capability of correctly following the lab procedure (and thus, also correctly applying the testing strategy), rather than on the performance of the test drivers they produced.

Another threat to internal validity is the use of the original, correct source code to generate their test cases based on code coverage analysis rather than using the faulty code on which the test cases are executed. The reason for this approximation is that it is impractical, if not impossible, to provide faulty code to subjects: The number of mutants varies from 382 to 1,176 in the three clusters, and this would have required generating different test cases for each mutant. Also, providing mutated code to participants would have allowed them to discover where the mutants were seeded by simply comparing the different files, and this would have been a major threat. It should be kept in mind, however, that the differences between each individual mutant program and the original program are very small.

4.3 Construct Validity

This type of threat is mainly related to our use of mutation analysis to measure the fault detection effectiveness of testing strategies as 1) the types of faults seeded may not be representative of "real" faults and 2) there might be faults not produced by the chosen mutation strategy [47]. However, the existing literature [5], [6], [26] suggests that faults seeded using mutation operators can be representative of real faults. Since, so far, no results contradicting the above studies have been reported, relying on mutation to compare test techniques is practical as it provides large, automatically generated fault samples. The fact that we used three class clusters with very different code characteristics—thus, leading to very different samples of mutants—should, however, limit the likelihood of this threat.

4.4 External Validity

External validity relates to the external aspects that interact with the treatments and limit the generalization of the results. The selection of fourth-year engineering students as participants in Carleton 1 and 2 and Master's students with little industry experience in Sannio 1 and 2 could be a threat to external validity as they may not be representative of the population of professional software developers. However, it is well known that productivity can vary a level of magnitude between the best and worst developers. Second, students at Carleton were overall good Java developers, as this is the main language used throughout their four years of study. Moreover, they are better acquainted with UML and, in particular, state machine modeling than most average practitioners: They passed two full-term courses on UML-based modeling. Students at Sannio already hold an engineering graduate degree and were also trained in Java programming and were carefully selected based on their academic track record. In addition, they all went through thorough testing training prior to the experiments. So, overall, for the specific tasks at hand, the participants used in our experiments can be considered competent. Issues related to the use of students in experiments are discussed in previous literature [7], [37], [39], which indicates that, often, advanced students and professionals are statistically similar in various performance measures. In conclusion, the existing literature suggests that though our results might not generalize to all experienced, professional developers, existing evidence suggests that they are probably representative for junior and intermediate developers. However, in some cases, the work pressure in industry—e.g., in the context of a major project releases—is different from that of an academic lab. To alleviate this problem, we used strictly enforced, fixed laboratory time.

The particular choice of class clusters to test in any experiment may be considered an external validity threat: Results can always be somehow specific to the software under test. Moreover, for controlled experiments, class clusters' size must be limited to allow enough time for performing the tasks in laboratory, controlled settings. However, while the class clusters we used could appear relatively simple and small compared to large, industrial systems or subsystems, we believe that they still can be considered representative of some of the situations where state testing would be applied. Though they differ in terms

TABLE 5
t-Test Results for the Mutation Score Comparison

Experiment	Cluster	DF	Mutation score mean		Mutation score variance		t-value	Pr > |t|
			Structural	State machine	Structural	State machine		
Carleton 1	OrdSet	31.6	56.15	50.27	399.42	295.97	0.93	0.359
	Cruise Control	15.8	27.69	24.47	92.07	2.86	1.35	0.197
Sannio 1	OrdSet	20.73	70.31	71.96	161.1	154.1	0.314	0.7567
	Cruise Control	12.97	44.65	35.65	257.5	23.3	-1.86	0.0853
Carleton 2	Elevator	5.46	40.54	35.44	110.9	78.15	-0.82	0.4483
	Cruise Control	13.77	50.13	30.8	219.4	55.17	-3.46	**0.0039**
Sannio 2	Elevator	15.92	36.97	35.06	198.9	172.7	-0.35	0.7325
	Cruise Control	9.37	46.89	34.55	173.4	36.79	-2.45	**0.0358**
All experiments	OrdSet	55.75	61.71	58.94	345.96	348.72	-0.56	0.5742
	Cruise Control	60.17	40.97	30.82	245.97	47.72	-4.04	**0.0001**
	Elevator	26.18	37.99	38.55	166.08	128.61	0.12	0.9033

of complexity, the `Cruise Control` and the `Elevator` clusters are representative of real-time, reactive classes with a state-dependent behavior, where class attributes are evaluated based on elapsed time between events and the current cluster state. The `OrdSet` class, on the other hand, is modeled by a large state machine with complex guard conditions. It is representative of classes encapsulating complex data structures. These two common categories of class clusters are usually modeled using state machines in UML-based development [21], [33], [42]. This is further supported by industrial case studies reported in the literature [22], [36]. Additionally, it is very uncommon in practice to model subsystems or entire systems using state machines, as this is far too complex in realistic cases. To assess testing techniques on such large programs, one would have to resort to industrial case studies. Controlled experiments involving humans in artificial settings necessarily use smaller programs, but they have, however, the advantage of achieving high internal validity. They must therefore often be complemented with industrial case studies that are usually strong on external validity (realistic, possibly larger systems, time constraints, etc.) but weak in terms of avoiding confounding factors.

The state machines for `Cruise Control` and `Elevator` do not fully model the two clusters under test. They only model the state behavior of the clusters. This is often the case in practice. State machines are rarely used to completely capture the system behavior as this would result in too large and unmanageable models. In conclusion, state testing results in practice can significantly depend on the level of abstraction of the models used, in other words, of the abstraction gap between the model and its implementation. This is particularly significant when modeling large programs, where abstraction becomes a necessity due to the complexity of the source code under test. It is therefore possible that our results would be significantly different on much larger programs and their corresponding models, which would be at a higher level of abstraction than that of the models used in our experiments. One plausible effect would be a decrease in the fault detection effectiveness of state testing and an even greater necessity to complement it with structural testing. In general, choosing an appropriate level of detail when developing state machines is an important decision which may be in part driven by testing objectives.

5 Experimental Results

This section presents the results obtained from the four controlled experiments. The results of each experiment are first presented separately, then followed by the identification and discussion of commonalities and differences. Wherever necessary, analysis of combined data sets from all experiments is also performed to increase the statistical power of the performed tests.

5.1 Impact of Test Techniques on Fault Detection Effectiveness

This section discusses the impact of the independent variable "test technique" (structural versus state machine) on the dependent variable "fault detection effectiveness" (RQ1, addressed in Section 5.1.1), and the possible interactions between the test technique and a number of cofactors in terms of their impact on fault detection effectiveness (RQ2, addressed in Section 5.1.2).

5.1.1 Univariate Analysis

To address RQ1, we perform a univariate analysis of the fault detection ratios of test drivers (fault detection analysis in Section 5.1.1), investigate the impact of cluster properties on the fault detection effectiveness of each test technique (impact of cluster properties on mutation scores in Section 5.1.1), and then discuss mutation score variations across drivers (understanding mutation score variations across collected drivers in Section 5.1.1).

Fault detection analysis

I. Comparison of mutation scores means across test techniques, clusters, and experiments. Table 5 reports the results of a two-tailed *t*-test [28] performed for each class cluster to assess the statistical significance of the difference in terms of mutation scores between the two test techniques. The following null hypothesis is tested: "*There is no*

TABLE 6
Estimated Effect Size for 80 Percent Power

Experiment	Cluster	Sample size		Mean difference	Standard deviation		Observed effect size	Effect size at 80% power
		Struct.	Stat.		Struct.	Stat.		
Carleton 1	OrdSet	17	18	5.88	19.99	17.2	0.32	0.98
	Cruise Control	15	17	3.22	9.59	1.69	0.47	1.03
Sannio 1	OrdSet	11	12	-1.65	12.69	12.41	0.13	1.23
	Cruise Control	12	12	9	16.05	4.83	0.76	1.20
Carleton 2	Elevator	4	8	5.1	10.53	13.31	0.42	1.90
	Cruise Control	10	7	19.33	14.81	7.54	1.64	1.48
Sannio 2	Elevator	10	10	1.91	14.1	13.14	0.14	1.32
	Cruise Control	8	11	12.34	13.17	5.95	1.21	1.38
All experiments	OrdSet	28	30	2.77	18.6	18.67	0.15	0.74
	Cruise Control	45	47	10.15	15.68	6.95	0.84	0.59
	Elevator	14	18	-0.56	12.89	12.83	0.04	1.03

significant difference between the number of faults detected by state test cases (Ts) and structural test cases (Tc)." Columns show the degree of freedom[5] (DF), the mutation scores' mean and variance per treatment, the calculated t-value, and the corresponding p-value ($\Pr > |t|$). The Anderson-Darling test [30] (H_0: samples significantly deviate from a normal distribution) indicated that mutation scores are normally distributed for the Carleton 2 experiment ($\text{p-value} = 0.07$) and Sannio 2 ($\text{p-value} = 0.19$), while they significantly deviate from normality in Sannio 1 ($\text{p-value} = 1\text{e-}6$) and Carleton 1 ($\text{p-value} = 0.002$). Therefore, in addition to *t*-tests, we performed a nonparametric Wilcoxon test to ensure the validity of the *t*-test results in cases where normality assumptions are violated, obtaining results consistent with those of Table 5. Detailed descriptive statistics of drivers' mutation scores are shown in a technical report [51].

For both `OrdSet` and `Elevator`, a *t*-test yielded a *p*-value greater than our selected level of significance, and therefore, the null hypothesis cannot be rejected. No statistically significant difference between state and structural drivers can be observed for these two clusters in terms of mutation scores. `Cruise Control` structural drivers had significantly higher mutation score means than state drivers in the last two experiments. When further investigating this difference between the results of the first two and the last two experiments, we note that the state machine mutation score did not vary significantly between Sannio 1 and Sannio 2, where all design factors were the same (same lab duration, oracle precision, and participants' skills). The mutation score mean dropped slightly in Carleton , from that observed in Sannio 1 and Sannio 2, and this may be attributed to the shorter lab time (three hours instead of four). The variance in mutation scores of state drivers varied significantly between Sannio 1, on one hand, and Carleton 2 and Sannio 2, on the other hand. This can be attributed to the varying number of implemented RTPs in Carleton 2 (few participants implemented all RTPs) and to the varying levels of implemented oracles in Sannio 2 (only state invariants, state invariants + class invariants, or state invariants + contracts checking). In Sannio 1, almost all participants implemented all RTPs from the `Cruise Control` transition tree and consistently used state and class invariants as oracles without resorting, in most cases, to pre and postconditions. On the other hand, structural mutation scores in the last two experiments were higher than in Sannio 1, especially in Carleton 2, with less variance.

5. The number of degrees of freedom is the number of independent observations for a data set. DF is explained as the number of independent components minus the number of parameters estimated [64].

The above suggests that both laboratory time and oracle precision have an important impact on mutation scores. This is clear when comparing the results of Carleton 1 to the results of the subsequent replications, where mutation scores for both clusters and treatments increased significantly compared to those in Carleton 1. Recall that laboratory time was increased in Sannio 1 and 2 and oracle precision was increased in the three replications by including contract assertion checking.

These results were also confirmed when applying the Holm's correction procedure (Section 3.5) to the pooled data from all experiments to account for possible type I error chance capitalization.

II. Accounting for trivial mutants in the calculated mutation scores means. To further investigate the fault detection effectiveness of each technique, we again computed mutation scores after removing "trivial mutants," i.e., mutants that are far too easy to detect and thus are not to be considered realistic and representative of real faults. In particular, we classified as trivial mutants killed by all drivers. The number of trivial mutants was 2 for `OrdSet`, 12 for `Cruise Control`, and due to its significantly larger size, was much higher (144) for `Elevator`. We recomputed *t*-tests considering mutation scores for nontrivial mutants only, and obtained results consistent with those of Table 5.

III. Power analysis. In some cases, the lack of significant results in Table 5 may be due to a lack of statistical power due to small samples. To this aim, we used power analysis to assess what would have been the minimum effect size for which we have a reasonable chance of detecting a statistically significant result.

Table 6 presents the power analysis results showing 1) the observed effect size based on the calculated mean difference and the pooled standard deviation and 2) the minimum effect size required to achieve a power of 80 percent at a significance level of 0.05.

We notice that for all four experiments, the minimum effect size corresponding to 80 percent power is large (0.98 to 1.90 across clusters), and this further justifies our decision to pool together the data of all experiments (last row), which shows significantly increasing power, that is, decreasing the minimum effect size matching 80 percent power: 0.59 to 1.03 across clusters. In fact, the cases where results are significant (Cruise Control for Carleton 2, Sannio 2, and all experiments combined) correspond to the cases where the actual effect size is close to or above the 80 percent power effect size threshold.

We can therefore conclude that these experiments do not allow us to draw conclusions about modest effect sizes. If there are significant differences, other than those already reported, between structural and state testing mutation scores, the unknown, actual effect size is significantly lower than 1, e.g., lower than 0.59, 0.74, or 1.03 for all experiments combined (the last three rows and last column in Table 6).

Impact of cluster properties on mutation scores. Here, we investigate whether the properties of the three class clusters used in the experiments had an impact on the fault detection effectiveness of the applied testing techniques.

For OrdSet, the high mutation score obtained with state testing is mainly due to the availability of a state machine providing an accurate description of the class behavior. For structural testing, the rather intuitive functionality of OrdSet leads to a high mutation score as well.

For Cruise Control, we observe a low mutation score and low code coverage for both state and structural drivers. For the former, this is mainly due to the real-time properties of Cruise Control not modeled in the state machine model. This was a choice of the original designers to avoid an overly complex and difficult-to-understand state machine that would attempt to specify actions on transitions triggered by timers to model the complex algorithms managing the variation of class attribute values over time, as discussed in [51]. We believe that this would often be the case in practice: Implicit transitions that do not affect current state and bear no actions—as for timer-controlled transitions of Cruise Control—are usually left out from the state machine to avoid cluttering the state machine diagram [12]. In many cases, developers and testers separate concerns, e.g., model real-time properties by means of other diagrams, such as activity or timing diagrams, or of many state machine diagrams instead of one complete model [65]. Participants working with the Cruise Control code also had difficulties—confirmed in survey questionnaire answers [51]—understanding the system real-time properties without the availability of proper documentation or a model. Basically, we did not provide any description of the clusters' real-time properties to any of the groups. Although a few participants noticed a relationship between the time factor and a number of class attributes, it was hard for them to understand the real-time algorithm that manages the class attributes solely based on code. A thorough understanding would have required the availability of models, such as communication diagrams or timing diagrams, thoroughly describing real-time properties.

The Elevator cluster also features a real-time behavior and includes concurrency. In contrast to Cruise Control, the real-time properties of Elevator not only affect class attribute values (concrete state), but also its current state as modeled by its state machine (abstract state). In other words, the real-time properties of Elevator are naturally modeled as state transitions. Despite that, state drivers' mutation scores in Elevator were relatively low (35.44 and 35.06 in Carleton 2 and Sannio 2). This may be due to 1) the high complexity of the state machine if compared, for example, to those of OrdSet and Cruise Control and 2) the fact that the state machine did not model concurrency, which is a peculiar aspect of the Elevator class cluster. Elevator's structural drivers did not do any better than state drivers (40.54 and 36.97 in Carleton 2 and Sannio 2) because Elevator's source code is by far more complex than that of the other two clusters (see Table 1).

Understanding mutation score variations across collected drivers. Results highlight a high variability of mutation scores achieved by structural drivers. This is not always the case with state drivers for which little variability is observed, at least for Cruise Control's drivers. The variation in the drivers' mutation scores was measured as a standard deviation and had the lowest value for state drivers in Cruise Control/Carleton 1. When applying RTP testing, participants had to produce test cases following a well-defined criterion and using a specific transition tree: This leaves little degree of freedom to the tester, also considering that—for Cruise Control—transitions have no guard conditions and require no parameter setting. The standard deviation of state drivers' mutation scores in the other three experiments was higher than in Carleton 1 (between 5 and 8 percent) and this is most likely caused by the increased, targeted precision of oracles. Different participants, however, implemented different levels of oracle precisions, ranging from state invariants, state invariants + class invariants to state invariants + contract checking.

As opposed to Cruise Control, the mutation scores' standard deviation for OrdSet state drivers is fairly large (17 percent as opposed to 1.5 percent for Cruise Control in Carleton 1). This can be explained as follows:

1. Only a few participants were able to cover all RTPs (35 percent RTP coverage, on average) and test cases in their drivers covered various numbers of RTPs [51].
2. The state machine has complex guard conditions and requires parameter settings which introduce variation in test cases.
3. Some faults can be detected only with very specific parameter values or set content.
4. Oracles often feature wrong or incomplete implementation of state invariants.

The standard deviation for OrdSet state drivers decreased in Sannio 1 to 12 percent, which can be attributed to the increased lab time, allowing participants to cover more RTPs.

For Elevator, the state drivers' mutation scores variation was limited (around 9 percent) and mainly due to the high complexity of the state machine.

Overall, for all class clusters and experiments, variations in structural drivers' mutation scores were higher than for

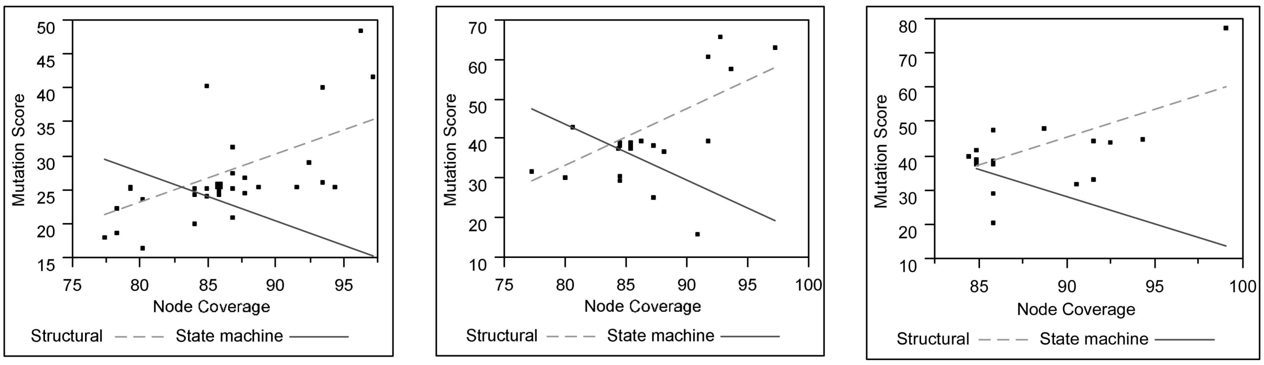


Fig. 1. Test technique and node coverage interaction effect graphs—cruise control. (a) Carleton 1. (b) Sannio 1. (c) Sannio 2.

state drivers' mutation scores. While participants working with state machines had a precise strategy to follow, and a predetermined number of test cases (corresponding to the number of paths in the transition tree) to implement, and predefined oracles, participants working with source code were only required to achieve edge coverage, without further guidelines or instructions.

5.1.2 *Interaction Effects*

To address RQ2, in this section we analyze, using a two-way ANOVA, the interaction effects on mutation score between the test technique and the following factors: code coverage, cluster, laboratory order, and subject ability.

Interaction effects with code coverage. There is a strong linear correlation between node and edge coverage ($R^2 = 0.9, 0.89, 0.93$, and 0.97 for the four experiments, respectively), and as a result, we limit any subsequent analysis to node coverage. For `Cruise Control`, the ANOVA tests showed significant main and interaction effects of test technique with node coverage on mutation score, with the exception of Carleton 2. This may be attributed to the relatively small number of observations (16) in Carleton 2. No significant main or interaction effect is observed for the other two clusters. Note, however, that when performing a two-way ANOVA of mutation score by test technique and node coverage without interaction, node coverage has a significant effect on mutation score: As expected, higher code coverage corresponds to higher mutation score.

Fig. 1 shows interaction effect diagrams of test technique and node coverage for `Cruise Control` for experiments where the interaction effect was found significant. Similar behavior is observed in the three diagrams.

When code coverage increases, structural drivers tend to have higher mutation scores than state drivers. This can be explained by the fact that participants working with state machines cannot infer information related to real-time behavior that, as discussed above, is not modeled by the state machines. On the other hand, for low code coverage, state drivers show higher mutation scores than structural drivers. This is likely due to the use of state invariant assertions in oracles. Note that the intersection between the interaction lines corresponding to structural testing and state testing occurs at around 85 percent node coverage in all three graphs (node coverage average across experiments and test techniques). This suggests that the structural testing technique starts to be more effective than the state technique when the coverage is rather high (i.e., above 85 percent). For Sannio 2 (Fig. 1c), the interaction lines did not go below 85 percent of node coverage as this threshold corresponds to the minimum node coverage across drivers.

No interaction effect was found between node coverage and test technique for Carleton 2. This may be due to the relatively small number of observations (16) in Carleton 2 (of which 6 were state drivers and 10 were structural drivers).

Interaction effects with lab order and participant ability. No significant impact of lab order and participant ability through interactions with test technique was observed on mutation scores. For lab order, a plausible reason is that participants were well trained for the tasks from the start and learning effects were therefore limited. Only one exception was observed in `Cruise Control`—Carleton 2, where a significant interaction effect was observed between lab order and test technique ($\text{p-value} = 0.04$). For participant ability, one possible problem is that ability as measured may not properly reflect testing skills, but rather the overall software engineering skills.

5.2 Combining Test Techniques Impact on Fault Detection Effectiveness

To address RQ3, we need to investigate whether it would be useful to complement test cases generated based on state machines (Ts) at design time with those generated based on code coverage analysis (Tc).

When combining test cases, all pairs of drivers (state drivers $\times$ structural drivers) must be taken into consideration to capture the variability among drivers written by different participants. Having m state drivers and n structural drivers implies that $m \times n$ combinations would be considered. However, we do not consider drivers with low coverage, as we want to consider only realistic scenarios with competent developers and a reasonably disciplined process. Some drivers had very low coverage due to a combination of poor development skills, lack of compliance with the provided task instructions, and limited time. Therefore, in order to improve the external validity of results to contexts with competent personnel and a reasonably disciplined process, we decided to compare only a subset of the drivers whose

TABLE 7
Drivers Selection Data for Combining Test Techniques

		# Selected		# Discarded		Total number of pairs to combine
		structural drivers	state drivers	structural drivers	state drivers	
Carleton 1	OrdSet	8	6	9	11	56
	Cruise Control	10	14	5	3	140
Sannio 1	OrdSet	9	12	2	1	99
	Cruise Control	9	11	3	1	99
Carleton 2	Elevator	3	3	1	4	9
	Cruise Control	9	3	1	3	27
Sannio 2	Elevator	5	4	5	5	20
	Cruise Control	7	10	1	0	80

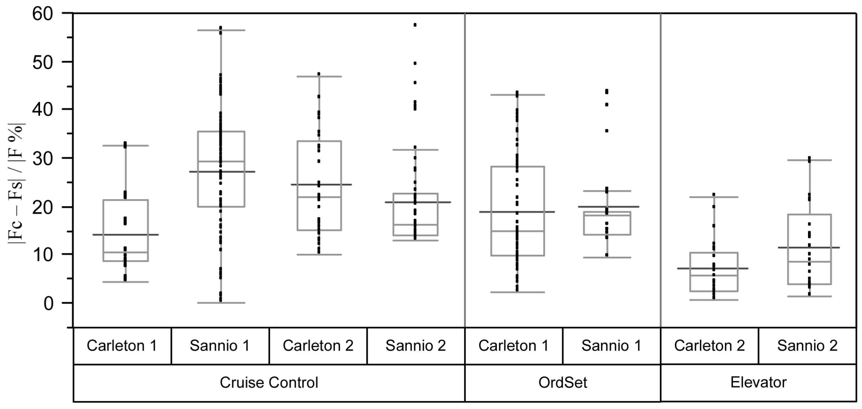


Fig. 2. Distribution of $|Fc - Fs|/|F|$ percent across experiments and clusters.

coverage was above a certain threshold, as described below. However, though it is common practice to seek high-statement coverage rates during testing in industrial test environments [59], this rate does not usually reach 100 percent due to budget and time restrictions, as well as the presence of unreachable code. Thus, we chose 85 percent as a reasonable threshold for the selection of structural drivers in `OrdSet` and `Cruise Control`, and we lowered it to 65 percent in `Elevator` as only low coverage rates were achieved because of its complexity.

As for RTP, the decision is more complex as there is not much reported practice in state testing. We chose a 100 percent threshold for `Cruise Control` as only a few participants did not cover all RTPs in their drivers. But, for `OrdSet` and `Elevator`, very few participants were able to cover all RTPs or even achieve high RTP coverage rates. In any case, we suspect that in a typical industrial environment, for a complex state machine with a large number of RTPs, only a subset of them is likely to fit within budgeted time and effort. Thus, we selected a 60 percent RTP coverage threshold for `OrdSet` and 45 percent for `Elevator` so as to obtain a reasonably large subset of almost half-complete drivers. Table 7 lists the number of selected and discarded drivers and the resulting total number of driver pairs that will be considered for the analysis in this section.

5.2.1 Complementarity of Testing Techniques

In order to address question RQ3, we analyze the set of mutants killed by one type of drivers and not the other. How complementary structural testing is to state testing is captured by $|Fc - Fs|/|F|$ percent.[6] In a realistic scenario, one would first generate state test cases, measure code coverage, and complement the test suite to achieve a certain level of structural coverage. The main motivation to follow that order is that generating large test suites being guided by code coverage analysis only is a highly tedious and time-consuming task [48]. Boxplots representing distributions of $|Fc - Fs|/|F|$ percent are provided in Fig. 2.

Results show that, on average, the percentage of mutants killed only by structural drivers ranges from 14 to 27 percent for `Cruise Control`, 16 to 17.5 percent for `OrdSet`, and 8.4 to 11.5 percent for `Elevator`. The highest increase in mutation scores brought by structural drivers is for `Cruise Control` for which real-time behavior-related code was not fully covered by state drivers. The lowest increase is for `Elevator`, which can be considered the most complex in terms of code and state machine when compared to the

6. *F* is the set of all faults (mutants), *Fs* is the set of faults detected by a state driver, and *Fc* is the set of faults detected by a structural driver (based on source code). $|Fc - Fs|/|F|\%$ is the percentage ratio of faults detected by a structural driver and not detected by a state driver out of the total number of faults.

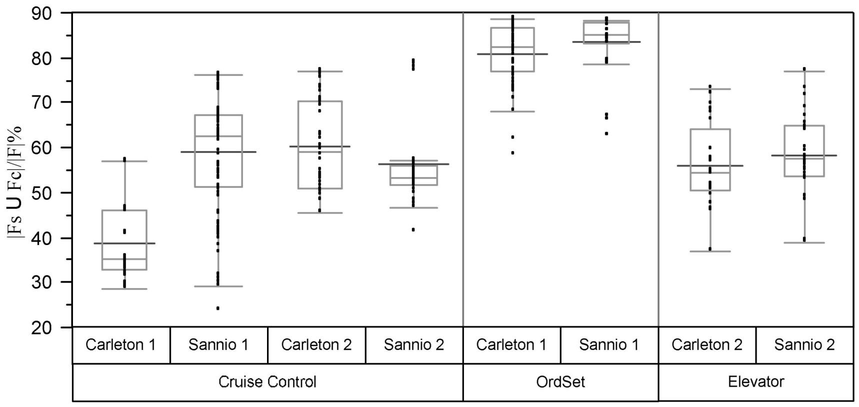


Fig. 3. Distribution of $|Fs \cup Fc|/|F|$ ratio across experiments and clusters.

other tested clusters. These results suggest that the contribution of structural drivers to a comprehensive global testing solution is significant but its extent depends on the complexity of the tested code.

5.2.2 Impact of Combining Test Techniques on Fault Detection Effectiveness

The $|Fs \cup Fc|/|F|$ percent measure represents the mutation score in percentage when combining state and structural drivers. Boxplots representing distributions of $|Fs \cup Fc|/|F|$ percent are shown in Fig. 3. As expected from the previous section, combining test cases from both test techniques clearly increased the mutation scores of combined drivers. The gain in mutation scores varied considerably across clusters and experiments. For instance, the mutation score of state drivers was, on average, doubled in Sannio 1/`Cruise Control`, while we only observe around 15 percent, on average, increase in Carleton 2/`Elevator`. When combining techniques, high mutation scores were achieved for `OrdSet` (e.g., an average of 85 percent in Sannio 1), while lower rates were obtained for `Cruise Control` (e.g., an average of 60 percent in Carleton 2) and for `Elevator` (e.g., an average 55 percent in Sannio 2). This again can be due to the already discussed real-time behavior and complexity of these clusters.

To formally address question RQ3, we need to analyze the statistical significance of the gain in mutation scores when combining state and structural test cases. The following null hypothesis is tested: "*The fault detection rate when combining state testing and structural testing is equivalent to that obtained with structural testing alone and to that obtained with state testing alone.*" Across the different clusters and experiments, 16 one-tailed *t*-tests for paired samples were performed to compare: 1) the means of mutation scores when including only structural test cases and after adding state test cases to them and 2) the means of mutation scores when including only state test cases and after adding structural test cases to them.

Results (p-values are reported in [51]) always indicate that the gain in mutation scores, from either structural testing or state testing, is statistically significant when combining test cases from the two testing techniques. The application of Holm's procedure to the pooled data from all experiments to account for possible type I error chance capitalization as described in Section 3.5 did not result in any change to the results presented and discussed in this section. Therefore, when testing clusters with state-dependent behavior, it is recommended to adopt a testing approach in which state testing is applied first and then complemented with structural testing.

In terms of practical significance, the improvements in mutation scores average between approximately 7 and 13 percent of all mutants across the three clusters when compared to structural testing alone and between 13 and 24 percent when compared to state testing. The improvements, for both test techniques, were higher in the replications for both `OrdSet` and `Cruise Control`. This indicates that the changes made to the experiment design improved not only the specific mutation scores of state drivers and structural drivers but also their combined mutation scores.

5.3 The Cost of Test Techniques

In this section, we address RQ4 by analyzing the difference between state testing and structural testing in terms of cost. This is complementary to the effectiveness analysis presented in the previous section as both differences in cost and effectiveness would have to be considered to select appropriate test strategies.

Testing cost can be generally divided into test generation cost and test execution cost. Regarding the former, we use as a surrogate measure the size of test drivers in terms of LOC to compare the effort required for generating the test cases. Keeping testers' skills constant, such a measure assumes that the test generation cost is proportional to the driver's size. Test cases execution cost would be measured with two different measures, capturing different aspects: CPU time and number of method calls in tested clusters. CPU time, measured in milliseconds (ms), provides an exact value for the time interval the CPU needed to execute the test cases. An alternative measure is the number of method calls, which focuses more on resources consumed—that are assumed to be proportional to the number of method calls—rather than on the CPU usage. Though this is clearly a strong assumption, for practical reasons it has been a common one in testing studies [6], [11], [20], [40], [67], where very often one test case corresponds to one execution of a function/program. Given that most methods in object-oriented software are small, as the number of methods called grows this count is likely to become a precise surrogate measure for test execution cost.

TABLE 8
Wilcoxon Test Results for Cost Analysis

Cluster	Experiment	CPU execution time			Nb of method calls			LOC		
		mean		Prob > |Z|	mean		Prob > |Z|	mean		Prob > |Z|
		Struct.	State		Struct.	State		Struct.	State	
Carleton 1	**OrdSet**	15.52	15.33	0.2562	382.43	502.17	0.6322	322.76	331.89	0.8045
	Cruise Control	8769.67	15.24	**0.0041**	600.64	875.55	**0.036**	260.87	372.94	**0.0055**
Sannio 1	**OrdSet**	15.54	22.25	**0.0263**	246.83	1298.50	**0.0061**	382.36	562.08	**0.0209**
	Cruise Control	23459.8	15.8	**0.0004**	787.82	1732.33	**<.0001**	187.5	715.58	**<.0001**
Carleton 2	**Elevator**	33316.5	21378.8	0.6711	455.25	867.43	0.2696	239.75	691.5	0.0643
	Cruise Control	39185.7	15.7	**0.0014**	684	1062.33	**0.0453**	280.9	616.28	**0.0072**
Sannio 2	**Elevator**	32478	18266	0.4869	728.7	1079.78	0.0757	254.6	550.22	**0.0128**
	Cruise Control	8342	15.8	**0.0002**	369.5	1103.6	**0.0003**	274.63	1358.5	**0.0004**
All Experiments	**OrdSet**	15.53	18.1	0.5334	674.18	1218.27	**0.049**	346.18	423.97	0.0652
	Cruise Control	19.370	15.6	**<.0001**	418.62	949.08	**<.0001**	248.2	711.28	**<.0001**
	Elevator	32717.6	22899.3	0.5684	650.57	1755.78	**0.0275**	250.36	691.11	**0.0004**

Two-sample *t*-tests were performed to obtain statistical evidence about the impact of test technique on the cost of test drivers (RQ4). Results show that the cost differences between the two test techniques are not consistently significant. We performed a nonparametric Wilcoxon test to ensure that this was not due to a violation of the *t*-test assumptions. Results are presented in Table 8.

For `OrdSet`, which has no real-time behavior, both state and structural drivers had almost instantaneous—and not significantly different—CPU execution time. The application of Holm's procedure (see Section 3.5) adjusted the p-value in Sannio 1 to 0.2, thus indicating that there is no significant difference in terms of cost between structural and state drivers. For `Cruise Control`, we found significant differences: While state drivers were almost instantaneous, it was not the case for structural drivers, as they more thoroughly exercise the cluster real-time behavior. For `Elevator` drivers, there is a difference between state and structural drivers, although not statistically significant. Differently from `Cruise Control`, the real-time behavior was modeled in `Elevator`'s state machine, and therefore it was accounted for by state drivers.

The results for method calls are not consistent across experiments and clusters. Significant differences in method calls were found for `Cruise Control`. This wasn't the case for `Elevator`. For `OrdSet`, the results were not consistent across experiments. The lack of significance in some experiments can be due to the small size of our samples. However, when combining data from all experiments, state drivers exhibit a cost significantly higher than structural drivers. An investigation of the high cost of state testing is presented in Section 6.2.

Comparing the test drivers' LOC, the results show that state drivers tend to have a higher cost than structural drivers, though this difference is consistently statistically significant for `Cruise Control` only. Also, state drivers' cost increased in the replication experiments for both `OrdSet` and `Cruise Control`. All of these observations lead to the following question: "What are the factors that have an impact on the cost of state drivers?" An attempt to answer this question is provided in Section 6.2.

5.4 Investigating the Cost-Effectiveness of Combining Test Techniques

This section addresses research question RQ5. We analyze the impact of augmenting state testing with structural testing on cost and compare the improvement brought to fault detection ratios to the cost increase, an analysis referred to as cost-effectiveness. Simply combining drivers by merging all test cases from both types of drivers may lead to erroneous conclusions regarding cost. The combined drivers may include a significant number of redundant test cases in terms of code coverage. In practice, a test driver would be first generated based on the state machine of the cluster under test. Next, when the code becomes available, based on coverage analysis, the state driver would be complemented to improve code coverage with structural test cases, keeping coverage redundancy among test cases to a minimum. The analysis in this section uses some of the drivers developed in this experiment to emulate and assess this procedure in terms of its impact on mutation score and cost.

Combining test drivers while eliminating redundant test cases has been performed as follows: For each pair of drivers `Ds` and `Dc` (state and structural drivers), we start by combining all test cases into one driver `Dsc` for which the node and edge coverage are calculated (`Nsc` and `Esc`). `Nsc` and `Esc` represent the highest code coverage that can be achieved by augmenting `Ds` with `Dc` (i.e., state machine testing with structural testing). However, this code coverage can perhaps be achieved with a subset of nonredundant (in terms of code coverage) test cases from `Dsc`. To remove any potential redundant test case from the augmented driver `Dsc`, we proceed by removing one structural test case at a time. If the code coverage measured after removing the test case does not change (from `Nsc` and `Esc`), then we consider the removed test case as a redundant test case which should

TABLE 9
Mutation Scores of Augmented Drivers

Cluster	Driver	Mutation scores			Relative increase of mutation scores	
		State	Structural	Combination	ΔMs	ΔMc
Cruise Control	Driver 1	39.53	61.58	63.6	0.61	0.03
	Driver 2	39.53	68.32	71.99	0.82	0.05
	Driver 3	39.53	75.13	76.96	0.95	0.02
OrdSet	Driver 4	79.17	71.96	86.7	0.10	0.20
	Driver 5	79.17	78.85	88.3	0.12	0.12
	Driver 6	79.17	83.33	86.7	0.10	0.04
Elevator	Driver 7	65.14	60.71	73.97	0.14	0.22
	Driver 8	65.14	49.83	70.41	0.08	0.41
	Driver 9	65.14	53.66	72.5	0.11	0.35

be eliminated from the augmented driver Dsc. Instead, if the code coverage drops, the test case should be kept in the augmented driver Dsc. We repeat this procedure for all structural test cases in Dc. This results in an augmented driver with no redundant test cases and the highest possible code coverage, given available structural test cases. The procedure presented above is intended to be representative of situations where RTP is adequately applied and code coverage is maximized within time constraints.

To address RQ5 using the test drivers developed in the experiment, the analysis proceeded by selecting three complete state test drivers (one for each cluster under test), implementing all RTPs with contract checking in oracle assertions. Similarly, we selected for each cluster three structural drivers to be used for augmenting the state driver among those collected drivers having the highest code coverage. Recall that our goal is to emulate the process of augmenting state drivers with structural test cases based on coverage analysis in order to maximize coverage based on realistic test cases and within time constraints. The selected high coverage drivers were generated by participants within the experiment time constraints and provide more opportunities (than other structural drivers) of finding test cases to augment and maximize coverage. The reason for selecting three structural test drivers for each cluster is to account for the significant variation observed among these drivers in terms of test cases.

Let us look first at the impact on mutation scores of using the procedure described above to form test drivers. Let Ms be the mutation score of state drivers and Msc the mutation score achieved by augmented drivers (i.e., after eliminating redundant test cases). Table 9 presents mutation score results of augmented drivers and their relative increase in mutation scores, i.e., $\Delta\text{Ms} = (\text{Msc} - \text{Ms})/\text{Ms}$ (when compared to state drivers) and $\Delta\text{Mc} = (\text{Msc} - \text{Mc})/\text{Mc}$ (when compared with structural drivers).

The relative increase in mutation scores of augmented drivers with respect to state drivers is the highest in Cruise Control (increase up to 95 percent), while it is much lower for OrdSet (increase up to 12 percent) and Elevator (increase up to 14 percent), for which state drivers had high mutation scores. The relative increase in mutation scores of augmented drivers with respect to structural drivers is the highest in Elevator (increase up to 41 percent), while it is much lower for Cruise Control (increase up to 5 percent).

Similarly to mutation scores, for each cost metric we compute the increase in cost due to augmenting drivers relative to the original cost of state and structural drivers. For example, let Cs be the cost of state drivers, Cc the cost of structural drivers, and Csc the cost of augmented drivers, then the relative increase in cost is computed as $\Delta\text{Cs} = (\text{Csc} - \text{Cs})/\text{Cs}$ (with respect to state drivers) and $\Delta\text{Cc} = (\text{Csc} - \text{Cc})/\text{Cc}$ (with respect to structural drivers). Finally, we define the *cost-effectiveness* of augmenting state with structural test cases as the ratios $\Delta\text{Ms}/\Delta\text{Cs}$ and $\Delta\text{Mc}/\Delta\text{Cc}$, considering the cost-effectiveness of augmented drivers with respect to state and structural drivers, respectively. The motivation is to compare the increase in mutation score to the increase in cost. For example, a cost-effectiveness of one tells us the relative increases in mutation score and cost are equal. A value below 1 suggests that the increase in mutation score is smaller than that of cost and a value above 1 suggests the contrary. Table 10 shows the cost-effectiveness results for state and structural drivers for the three cost metrics.

For Cruise Control, the cost-effectiveness of augmented drivers is higher with respect to state drivers when considering the cost metrics "Number of method calls" or "Lines of Code" (gray cells). The gain in mutation score is much larger than the increase in cost. This is not the case with respect to structural drivers where the increase in cost is larger than the gain in mutation score. This can be explained by the fact that Cruise Control's state drivers had limited code coverage as the real-time behavior of the cluster was not modeled in the state machine. When considering the "CPU execution time metric," results are different from the other two metrics. Though the gain in mutation scores of state drivers was high (Table 9), the cost increased so much that the cost-effectiveness of adding

TABLE 10
Cost-Effectiveness for Augmented Drivers with Respect to State and Structural Drivers

Cluster	Driver	Cost-effectiveness					
		CPU execution time		Number of method calls		LOC	
		ΔMs/ΔCs	ΔMc/ΔCc	ΔMs/ΔCs	ΔMc/ΔCc	ΔMs/ΔCs	ΔMc/ΔCc
Cruise Control	Driver 1	<0.01	-0.08	0.48	0.05	1.40	0.02
	Driver 2	<0.01	-0.07	2.30	0.04	2.43	0.13
	Driver 3	<0.01	-0.03	2.32	0.02	2.36	0.10
OrdSet	Driver 4	0.14	0.31	0.27	0.11	0.18	0.68
	Driver 5	0.16	0.19	0.75	0.02	0.67	0.05
	Driver 6	0.16	0.08	0.26	0.02	0.28	0.06
Elevator	Driver 7	0.15	0.93	1.08	0.09	0.76	0.24
	Driver 8	0.44	0.20	2.67	0.04	0.84	0.08
	Driver 9	0.06	0.90	1.06	0.07	0.40	0.11

structural test cases to a state driver was close to 0. With respect to structural drivers, the cost effectiveness is even negative. This is because augmented drivers led to a slight increase in mutation score (Table 9), while they exhibited a decrease in CPU execution time with respect to structural drivers. This is because augmented drivers did not contain the structural test cases which were redundant with respect to the chosen state drivers and particularly expensive in terms of CPU time.

For `OrdSet`, the cost-effectiveness of combining drivers is, in general, low. The gain in mutation score was small compared to the increase in cost, regardless of the specific drivers and cost metrics used. On one hand, `OrdSet`'s state machine fully modeled the functionalities of the cluster, leading to high mutation scores and high coverage in state drivers when covering all RTPs, and on the other hand, the rather intuitive functionality implemented in the source code of one class led to high mutation scores and high coverage in structural drivers. Therefore, the increase in mutation scores when augmenting state drivers was smaller than the cost increase to achieve higher coverage.

For `Elevator`, the cost-effectiveness of augmented drivers with respect to state drivers (ΔMs/ΔCs) was only high with respect to "Number of method calls." The increase in state drivers' mutation scores was related to covering specific functionalities of the cluster not modeled by the state machine (mainly concurrency). With dedicated test cases covering parts of the code implementing concurrency, structural drivers led to an increase in code coverage and mutation score while incurring a relatively small increase in cost.

We may not be able to generalize the results of this section, but we can identify a trend: The cost-effectiveness of complementing state drivers with structural test cases is relatively high when the state machine modeling a cluster under test does not fully model its functionalities. This is rather the general case as state machines are rarely used to model the entire system (further discussions of this point can be found in Sections 3.1 and 4.4). Structural test cases covering code implementing the nonmodeled functionality would bring an increase in the mutation score of the augmented driver with relatively low cost.

5.5 Equivalent Mutants' Analysis

While processing the data collected from the different experiments, we analyzed all undetected (live) mutants for the purpose of understanding the limitations of test techniques, with a particular focus on state drivers (Section 6.1). For each live mutant by state drivers, we created execution traces for the faulty program to understand why some faults were not detected. This helped us identify equivalent mutants that do not change programs' outputs and behavior. For instance, a mutant that deletes an initialization of an integer to zero is an equivalent mutant as the compiler would automatically assign an initial value of zero to uninitialized attributes. Other types of mutants are more complex and may, for example, require studying boundaries of attributes in some loops.

In total, 40 mutants were found to be equivalent mutants in `Cruise Control`, 63 in `OrdSet`, and 133 in `Elevator`. This corresponds to 10, 10, and 11 percent of the total number of generated mutants in the three clusters, respectively. Table 11 provides details on the percentage of live mutants from the total number of generated mutants and the ratio of equivalent to the live mutants for each treatment. We consider live mutants to be those still undetected when running all the available state drivers and structural drivers. $|F - Fc|$ percent represents the percentage of live mutants corresponding to faults undetected by structural drivers and $|F - Fs|$ percent represents the percentage of live mutants corresponding to faults undetected by state drivers. $|Ne|/|F - Fc|$ and $|Ne|/|F - Fs|$, where Ne is the set of equivalent mutants, represent the ratio of equivalent mutants out of live mutants undetected by structural drivers and state drivers, respectively.

The number of live mutants is much higher than the number of equivalent mutants in the tested clusters. This suggests that a heuristic—previously used on other empirical studies on software testing [5], [8]—considering all faults undetected by any driver as equivalent mutants and

TABLE 11
Count of Live Mutants per Experiment and per Treatment

Experiment	Cluster	|F – Fc|%	|F – Fs|%	|Ne|/|F – Fc|	|Ne|/|F – Fs|
Carleton 1	Cruise Control	28.5	74.8	30.3	11.5
	OrdSet	13.5	16.5	76.2	62.1
Sannio 1	Cruise Control	24.1	57.8	35.9	14.9
	OrdSet	11.5	10.9	88.9	94.1
Carleton 2	Cruise Control	20.4	60.4	42.3	14.3
	Elevator	45.1	34.9	22.8	29.5
Sannio 2	Cruise Control	20.7	57.3	41.8	15.1
	Elevator	32	29.7	32.2	34.7

eliminating them from the total set of mutants—cannot be applied in our case, especially in the case of `Cruise Control`. We tried to repeat the analyses of Section 5 after removing the equivalent mutants, and although mutation scores obviously increased, these changes turned out not to make any difference in terms of the conclusions we have drawn in the previous sections. A detailed, qualitative analysis of live mutants will be presented in Section 6.1.

6 Qualitative Analysis

Results in Section 5 indicated a number of unkilled mutants, higher for `Cruise Control` and `Elevator`, and lower for `OrdSet`. To better understand why certain faults are difficult to detect by state drivers and structural drivers, we report on a qualitative analysis to identify what execution conditions would be required to detect those faults and whether these conditions were likely to be fulfilled by either state testing or structural testing (Section 6.1). Also, we report on a qualitative analysis (Section 6.2) aimed at better understanding the higher cost of state drivers compared to structural drivers. Finally, we investigate reasons for the limited structural coverage achieved by state drivers (Section 6.3).

6.1 Qualitative Analysis of Live Mutants

The main motivation for the qualitative analysis we performed on live mutants is to identify ways to improve the state testing strategy to increase its fault detection effectiveness. Another motivation is to classify faults detected by one technique and not by the other. To this aim, we identified the following disjoint sets of faults:

1. $F - (Fs \cup Fc)$, the set of all faults not detected by any driver.
2. $Fc - Fs$, the set of all faults detected only by structural drivers.
3. $Fs - Fc$, the set of all faults detected only by state drivers.
4. $Fs \cup Fc$, the set of all faults detected by both types of drivers.

Then, we identified reasons for not detecting faults (Section 6.1.1), and finally, we classified undetected faults according to these reasons (Section 6.1.2).

6.1.1 Identifying the Reasons for Not Detecting Faults

We identified the reasons for not detecting faults with a focus on faults that were not found at all $F - (Fs \cup Fc)$, and those found by one technique and not the other: $Fs - Fc$ and $Fc - Fs$. This was done by executing the corresponding mutants and generating execution traces. If the fault does not affect the output, the trace can then help us identify the reason. Also, if the fault does indeed affect the output, the trace then helps us understand why the oracle did not detect any failure. An example of such a fault is one created by seeding a fault in the method `resizeArray()` of the `OrdSet` class, as shown below (the index `k` highlighted in the code was replaced by `k++`):

```
private void resizeArray() {
  int new_size = _set_size + min_set_size;
  if (new_size <=max_set_size && _resized_times<max_accepted_resizes)
  {
    int[] _new_set = new int[new_size];
    for (int k = 0; k < _last + 1; k++) {
      _new_set[k] = _set[k];
    }
    _set_size = new_size;
    _set = _new_set;
    _resized_times++;
  } else {
    _overflow = true;
  }
}
```

Method `resizeArray()` is called whenever an element is to be added to a full set and that element is not already in the set. A resize can occur if two conditions are true: 1) The resized set size does not exceed the maximum set size (a constant) and 2) the number of resizes done on the set does not exceed the maximum resizes allowed (a constant). The fault gets executed if those conditions are met, and when this is the case, the set is resized as expected, but with wrong content. To detect the fault, it is necessary to check the exact content of the set. This can be done by verifying the class invariant or the `resizeArray` postcondition in the oracle. An example of a test case that causes a failure if this fault is executed is to create an ordered set with content {1, 2, 3, 6}, and then add the element "4" to the set. The result one gets, assuming the two conditions mentioned above hold, is {0, 2, 0, 4, 6} instead of {1, 2, 3, 4, 6}. Such a fault was not detected by Carleton 1's state drivers, while it was detected by Sannio 1's state drivers, where oracle helpers contained implementations of contract assertions.

As a result of this process, live mutants were divided into two sets: 1) the set of equivalent mutants and 2) the set of remaining live mutants, which should be considered as undetected faults and which are classified into categories and discussed in Section 6.1.2. Note that equivalent mutants represent 19 percent of live mutants in `Cruise Control`, 84 percent in `OrdSet`, and 37 percent in `Elevator`. They are omitted from the remaining discussions as they do not represent faults.

6.1.2 Classification of Undetected Faults

This analysis was first performed on drivers collected from Carleton 1, with the aim of identifying changes to the experiment design and context for replications from Sannio 1, as described in Section 3.4. The qualitative analysis on the collected drivers from the replications aimed at identifying further ways to improve the state testing strategy. As an outcome of the qualitative analysis, the identified categories of undetected faults are:

1. *Faults requiring precise oracles in order to be detected*: Introducing such faults in the code causes changes in one or more attributes' computation that cannot be detected without precise oracles. This is the most frequent category in `Cruise Control` (Table 13). By using contract assertions in replications (Sannio 1, Carleton 2, Sannio 2), the number of undetected faults in this category dropped from the original experiment (Carleton 1), although a significant percentage of these faults (79 percent) remained undetected. To detect these faults, it would have been necessary to build an oracle that would nearly replicate the behavior of the code under test, which is not realistic in practice. The real-time behavior of `Cruise Control` was tested by experiment participants writing structural test cases. The availability of source code helped them to understand the algorithm managing the relationship between time and the value of class attributes such as speed and distance. In `OrdSet`, the use of contract assertions in oracles decreased the number of undetected faults belonging to this category from 32 (31 percent of undetected faults) in Carleton 1 to no fault in Sannio 1.
2. *Faults requiring specific scenario (e.g., a specific path in the state machine that is not covered by RTP) in order to be detected* such as repeating some command call in `Cruise Control`. Specific sequences of events would be required in order for test case executions to reach specific attributes' values. Detecting some faults of this category would have required triggering an event (or a sequence of events) several times in the same test case (tested path). This, however, is not accounted for by the RTP testing technique, which limits the lengths of paths traversing the state machine graph, and therefore can be considered as a limitation of that technique. Some paths, even when they represent common usage scenarios of the system, are not necessarily covered by transition trees generated with the RTP testing technique. An example is the scenario of a real journey of an elevator: getting a number of requests, servicing them one after the other, and finally, stopping in the Idle state. This scenario, as well as many others, is not present in transition trees derived from the `Elevator` state machine. In `Cruise Control`, in order to be detected, some faults require calling a command (firing a transition) multiple times, as accelerating many times to get to the maximum throttle attribute value. Again these scenarios are not part of the transitions trees. In the replication experiments, sneak path testing was included as part of state testing (in addition to RTP). This decreased the undetected faults of this category for `Cruise Control`.
3. *Faults requiring specific parameters to be passed in the test case* (such as specific set content in `OrdSet` or covering boundaries) *in order to be detected*. Such faults suggest that a combination of boundary analysis or category partition techniques with the state testing technique would be beneficial.
4. *Faults requiring specific execution time* (to allow for real-time attributes values to change*) in order to be detected*. These faults are specific to the real-time clusters. In `Elevator`, the number of undetected faults of this category was much lower than in `Cruise Control` because the real-time properties of `Elevator` are naturally modeled as state transitions and state actions in its state machine (impact of cluster properties on mutation scores in Section 5.1.1). This category of faults is mainly observed in `Cruise Control`, where state drivers did not execute for a long enough period of time (e.g., to reach the maximum speed) to detect faults in this category. This is the second important category of undetected faults in `Cruise Control` (25 percent of undetected faults— Table 13).
5. *Faults affecting static attributes*: modifying a class' static attribute into a nonstatic attribute or vice versa. Those faults were not detected in any experiment. Test cases always included one instance of the class under test at a time. For instance, in `Elevator`, these faults were not detected by state drivers as the state machine models only one elevator.
6. *Faults causing wrong intermediate behavior without affecting the final outputs*: These faults are observed in `Elevator` when the exact behavior of each elevator is not checked by state drivers. For instance, an elevator may service a floor by going down first, then going up instead of going up right away. The state machine does not distinguish between moving up or moving down and how many floors are visited before the requested floor is serviced (as long as the requested floor is indeed serviced).
7. *Faults not observable based on the state machine*: These faults can be mainly observed in algorithms implemented in the source code, but not modeled by the state machines. This category represents the most important category in the `Elevator` cluster (51 percent of undetected faults—Table 13). These faults are mainly observed in the algorithm managing the selection of the best elevator to service a job. This algorithm is not modeled by the state machine, and therefore state drivers could not

TABLE 12
Distribution of Categories of Undetected Faults among Testing Techniques

Category of undetected faults	**F – (Fs U Fc)**	**Fc – Fs**	**Fs – Fc**
Faults requiring precise oracles in order to be detected	x	x	
Faults requiring specific scenario	x	x	
Faults requiring specific parameters to be passed in the test case	x		
Faults requiring specific execution time	x	x	
Faults affecting static attributes	x	x	
Faults causing wrong intermediate behavior without affecting the final outputs	x	x	
Faults not observable based on the state machine	x	x	
Faults affecting state behavior			x
Faults affecting class invariants			x
Faults affecting method results			x

detect the faults. In the `Cruise Control` cluster, such faults occur in the cruise control algorithm and in the algorithm implementing the real-time behavior of the simulated car.

8. *Faults affecting state behavior:* As a result of the execution of the fault, the state of the cluster is different from the expected state result. These faults went mainly undetected by structural drivers in `Elevator`, where the state behavior is more complex than in the other two clusters. Implementing state invariants in oracle assertions of state drivers ensured the detection of those faults.
9. *Faults affecting class invariants*: As a result of the execution of the fault, the class invariant evaluates to a wrong value. A class invariant represents a set of constraints on class attributes values. Implementing class invariants in oracle assertions of state drivers ensured the detection of faults where an attribute's value is set out of specified boundaries. In `Elevator`, the number of elevators and floors defined in the group of elevators is specified in the class invariant. A fault changing such numbers is detected by state drivers. In `OrdSet`, the minimum and maximum size of an ordered set is specified as well in the class invariant. Having a set size smaller than the minimum size or greater than the maximum size is easily detected by state drivers.
10. *Faults affecting method results*: They may occur when the method postcondition depends on the object state. These faults might not be detected by structural drivers as they may require that the object be in a specific state to trigger failures.

Table 12 shows the distribution of undetected fault categories among the three subsets of undetected faults: 1) $\mathrm{F} - (\mathrm{Fs} \cup \mathrm{Fc})$, the set of all faults not detected by any driver; 2) $\mathrm{Fc} - \mathrm{Fs}$, the set of all faults detected only by structural drivers; and 3) $\mathrm{Fs} - \mathrm{Fc}$, the set of all faults detected only by state drivers.

The categories of faults detected by structural drivers and not detected by state drivers (Fc-Fs) appear to be a subset of the categories of faults undetected by any driver. In fact, while most of the faults belonging to these categories were undetected by any strategy, a number of them were, however, detected by structural drivers.

Table 13 reports, for each cluster, the counts of mutants corresponding to each category of undetected faults and what these counts represent as percentages of live mutants not including equivalent mutants.

6.2 Investigating the Variation in Cost

Results in Section 5.3 suggest that the cost—in terms of generation cost (LOC)—of structural testing is higher than that of state testing. The results of our qualitative analysis show that the main cause for cost variability in structural drivers is a high level of redundancy in test cases where multiple test cases partially cover the same nodes and edges. Redundancy is limited during state testing, as test cases are precisely specified by a test strategy (RTP), leading to a limited redundancy when coding drivers. Another source of variability is the ineffective use of the available public methods to implement pieces of functionality required to create test drivers. For example, in `OrdSet`, a set can be created with two constructors, one creates an empty set and another creates a set with content from an array of integers. Some participants did not use the second constructor to create a nonempty set. Instead, they created an empty set and iteratively added elements to it with the "add one element" method. This increased the number of called methods in their drivers considerably. One factor causing variability in state drivers' cost is the number of implemented RTPs—varying widely, especially for `OrdSet` (from 10 to 100 percent) and for `Cruise Control`, where, in order to cover sneak paths, the number of covered RTPs for experiment replications was 25, including 13 sneak paths, instead of the 12 of Carleton 1.

Another factor causing variability in state drivers' cost is the variation in precision of implemented oracles (state invariant assertions only, state invariant + class invariant assertions, or state invariant + contract assertions). When profiling the execution of a driver, we found that getters were the most called methods. This indicates that oracle precision heavily contributes to the high cost of state

TABLE 13
Mutant Count (Percentage) per Category of Undetected Faults and per Cluster

Category of undetected faults	**Elevator**	**Cruise Control**	**OrdSet**
Faults requiring precise oracles in order to be detected		65 (38%)	
Faults requiring specific scenario	18 (8%)	20 (12%)	
Faults requiring specific parameters to be passed in the test case	8 (3%)		3 (25%)
Faults requiring specific execution time		43 (25%)	
Faults affecting static attributes	14 (6%)	11 (6%)	2 (17%)
Faults causing wrong intermediate behavior without affecting the final outputs	35 (15%)	2 (1%)	
Faults not observable based on the state machine	116 (51%)	25 (15%)	1 (8%)
Faults affecting state behavior	6 (3%)	1 (1%)	
Faults affecting class invariants	7 (3%)	1 (1%)	6 (50%)
Faults affecting method results	25 (11%)	2 (1%)	

drivers. In fact, the cost significantly increased for replications where we increased the oracle precision by including contract assertions.

We can summarize the factors that contribute to the high cost of state drivers as follows:

1. The use of precise oracles after every event call (includes a high number of method calls).
2. A number of RTPs have common subpaths (i.e., the initial setting may be common to a number of RTPs and repeated in a number of test cases), and therefore common code coverage.
3. A high number of RTPs.
4. Poor usage of public methods, mainly in test case setup.

6.3 Analysis of Code Coverage

Another point worth being qualitatively investigated is the code coverage achieved by state drivers. Uncovered nodes and edges are due to the following:

1. *Methods not triggered by the state machine tests nor by oracles* (state invariants + contracts): Some of the public methods are not triggered by state machine events, actions, or activities, nor by oracles (state invariants and contracts checks), either directly or indirectly. These methods do not model functionality or behavior, but they implement helper methods (e.g., `toString()` methods) or getters.
2. *Untested functionality*: Some cluster functionalities are modeled in the state machine but not exercised, or at least not fully exercised from the state drivers. For instance, in `Cruise Control`, the functionality of "Cruising," i.e., keeping the speed of the car at a fixed value, is modeled in the state machine but not fully exercised by the state drivers. This mainly depends on the real-time characteristics of the `Cruise Control` cluster. In `Elevator`, the state machine does not distinguish between moving the elevator up or down as only one state "moving" models both behaviors. The state drivers covered that state, however, only letting the elevator move up because, when the system is initialized, the elevator starts its operation from the ground floor, and therefore moving would always start as moving up. The constraints of the RTP testing technique on the generation of paths in the state machine would not allow the generation of a path in which the elevator would move up to get to some floor, and then change direction and move down.
3. *Catching exceptions*: A number of the exception handling nodes and edges were not exercised by state drivers. Only two nodes and edges in this category are found in `Elevator`. Testing strategies dealing with exception coverage are proposed in [63].
4. *Unmodeled functionality*: The state machine does not model all cluster functionalities. This is the case for `Elevator`, where the concurrent behavior of elevators is not modeled by the state machine, which models only one elevator instance.
5. *Handling boundary cases and unexpected entries*: A number of nodes and edges handling boundary cases and unexpected entries have not been exercised in state drivers. For instance, the code handling a nonrecognized command in `Cruise Control` was not exercised in state drivers.

6.4 Lessons Learned

This section summarizes lessons learned about the combination of state-based testing and structural testing, based on both the results of our empirical study and the qualitative analysis presented earlier in this section. Although some of our conclusions may seem common wisdom, to the best of our knowledge this represents the first attempt to investigate them based on qualitative and quantitative empirical analyses.

The goal of the lessons we summarize below is to improve fault detection effectiveness and code coverage of state testing while limiting its cost. Empirical evidence of the effectiveness of the recommendations provided in this section and further recommendations on how to improve state-based testing strategies can be found in [51].

State testing is not a silver bullet as it cannot address all clusters' properties. A state machine is not able to represent all of the properties of a system that would be relevant for implementing it, and therefore, also for thoroughly testing it. In particular, it is often not practical to model all real-time and concurrency properties in a state machine (impact of cluster properties on mutation scores in Section 5.1.1). Real-time properties may be difficult to model in a state machine, especially when other factors affect attributes' variations over time (e.g., car speed is determined through a relationship involving air resistance, throttle, and current speed over time). Also, concurrency adds considerable complexity to state machine models. These limitations of state testing were outlined by the results of the qualitative analysis: Section 6.1.2 (items 2, 4, 6, and 7) and Section 6.3 (items 2 and 4). Separating concerns, i.e., state-dependent behavior versus time-dependent behavior or concurrent behavior, addresses this complexity issue by taking the modeling of real-time and concurrency properties away from the state machine and modeling them with other means. Therefore, we consider that state testing is sufficient by itself only when the state machine precisely and completely represents the behavior of a sequential component with no real-time and concurrent behavior. However, for complex real-time or concurrent components, a model-based testing strategy should not be limited to state testing. A complete model-based testing strategy would include other model-based testing techniques such as a use-case-based testing technique to cover common use case scenarios and high-stress use case scenarios not covered by RTPs. Techniques based on activity diagrams or sequence diagrams would be required to fully exercise methods with complex control flow or functionality representing the cluster's real-time behavior.

Sneak paths should be covered by state testing. For the sake of simplification, self-transitions with no actions are often omitted when modeling state machines. Qualitative analysis (Section 6.1.2—item 2) and the results of the replication experiments (fault detection analysis in Section 5.1.1) proved the importance of including sneak path testing to improve state testing fault detection effectiveness. Therefore, it is recommended to complement the original RTP technique with sneak path testing as it was originally suggested by Binder [12].

One must reach an appropriate compromise between highly precise oracles and testing cost. Increasing oracle precision proved to improve fault detection effectiveness but also increase the cost of state testing (Section 6.1.2—item 1 and Section 6.2—item 1). Testers might want to reach a compromise between having a highly precise oracle and keeping the testing cost down—in terms of resource consumption and time needed to run the test cases. For instance, we observed (see Section 6.2—item 2) that RTPs have common subpaths, and therefore oracle checks for events (and actions) in these common subpaths are performed several times (each time the common subpath appears in a test case). To reduce the testing cost, we recommend only checking oracles in a common subpath once. The oracle cost can also be reduced by simplifying Boolean expressions in oracle checks, i.e., by putting them in a disjunctive form.

Useful heuristics can be used to reduce the cost of state testing. State testing cost can also be reduced by reducing the number of RTPs in the generated transition tree (see Section 6.2—item 3). To this aim, a depth-first traversal of the state machine would often generate less, but longer, RTPs, with a smaller number of method calls in total, than breadth-first traversal. Common subpaths in depth-first traversal RTPs are expected to be reduced as well. This is supported by the examples in [51].

Complementing state-based testing with structural testing is cost-effective when state machines do not fully capture the class cluster behavior. As shown in Section 5.4, if state machine models are very detailed—e.g., as happens for `OrdSet`—the cost-effectiveness of complementing state-based drivers with structural drivers (ΔMs/ΔCs) is relatively low, as the added drivers are not able to exercise a substantial, additional portion of the source code (not covered yet by state drivers) and thus detect additional faults. These additional drivers might be expensive in terms of source code to be written and of methods to be invoked (as they require developing precise oracles), thus the overall cost-effectiveness of complementing state drivers with structural drivers is generally low. On the other hand, the cost-effectiveness of complementing state drivers with structural drivers is higher (measuring the cost in terms of method calls or drivers's LOC, see the gray columns in Table 10) for state machine models when they are not precise enough to fully capture the class cluster behavior. This is the case for `Cruise Control` and `Elevator`, where real-time behavior and exceptions are not captured by the state machine models.

7 Conclusions

This paper investigated, through a series of controlled experiments involving human participants (senior, carefully trained undergraduate students and experienced graduate students), the fault detection effectiveness and cost of state testing of class clusters with state-driven behavior. This is of practical importance as state-driven testing has often been recommended for complex class clusters in the literature [13], [15], [20], [23], [54], [56]. To provide a baseline of comparison, we compared state testing with structural testing based on code coverage analysis, which can be considered a common, widely adopted practice for testers. Furthermore, we investigated whether the two strategies are complementary in detecting faults and could be combined. Finally, we investigated factors that may affect the effectiveness of these testing strategies.

Results show—in a context where testers have limited time and where state machines closely (but realistically) model the functionalities of the cluster—that testing driven by code coverage analysis is not less effective at detecting faults than a well-known strategy for state testing, i.e., the W-method [24] or round-trip path testing for UML state machines [12]. However, the two test strategies seem to be complementary in terms of the faults they are able to detect. This suggests that they should probably be used together, as opposed to being considered as alternatives. Since state machines are often produced before code and since testing based on code coverage analysis is notoriously tedious and time-consuming, it is probably wise to first test class clusters based on state machines and then complement test suites based on coverage analysis.

The obtained results also suggest that the effectiveness of state testing strongly depends on the nature of the software under test, the state machine model itself, but also the choice of test oracles. Our investigation shows that state testing alone is not sufficient to test class clusters with real-time and concurrent properties. Other model-based testing techniques would be required for this purpose. For example, in some cases it may be advisable to complement the state machine with activity, timing, or sequence diagrams, describing properties not fully modeled by state machines (e.g., modeling the way time-dependent class attributes are updated), and then trying to cover such diagrams to complement state testing. Our results also show the benefits of using precise oracles based on contract assertions and class invariants, as well as the usefulness of testing illegal and implicit transitions in state machines. When considering the generation and execution cost of the test techniques, state testing is found to be more costly than structural, code coverage testing, mainly because of the higher cost of its oracles.

The cost-effectiveness of augmenting state drivers with structural test cases is found to be related to the extent to which the state machine of the cluster under test closely captures the cluster's behavior. A high cost-effectiveness is observed when the state machine does not fully model the cluster's behavior, and in particular, aspects related to concurrency and time.

This paper is the first to report a thorough and detailed assessment, comparison, and combination of structural and state testing, two commonly recommended and used approaches to systematic testing. Though the above experiments involved trained and competent students as subjects, to confirm and generalize our results it is necessary to replicate the experiment on other populations, including professionals with different levels of experience. Future work will investigate how state testing can be improved—even augmented with other models—to detect faults that have been shown to be detected only by structural drivers and how its cost can be further decreased.

REFERENCES

[1] Eclipse, http://www.eclipse.org, 2009.
[2] Eclipse Metrics Plugin, http://sourceforge.net/projects/metrics, 2009.
[3] Eclipse Test and Performance Tools Platform Project (TPTP), http://www.eclipse.org/tptp, 2009.
[4] Software-Artifact Infrastructure Repository (SIR), http://sir.unl.edu/portal/index.html, 2009.
[5] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. 27th Int'l Conf. Software Eng.*, pp. 402-411, 2005.
[6] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608-624, Aug. 2006.
[7] E. Arisholm and D.I. Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Trans. Software Eng.*, vol. 30, no. 8, pp. 521-534, Aug. 2004.
[8] D. Baldwin and F. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Dept. of Computer Science 276, Yale Univ., 1979.
[9] V.R. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 456-473, July/Aug. 1999.
[10] B. Beizer, *Software Testing Techniques*, second ed. Van Nostrand Reinhold Co., 1990.
[11] J.M. Bieman and J.L. Schultz, "An Empirical Evaluation (and Specification) of the All-du-Paths Testing Criterion," *ACM Software Eng. J.*, vol. 7, no. 1, pp. 43-51, 1992.
[12] R.V. Binder, *Testing Object-Oriented Systems—Models, Patterns, and Tools.* Addison-Wesley, 1999.
[13] K. Bogdanov and M. Holcombe, "Statechart Testing Method for Aircraft Control Systems," *Software Testing, Verification and Reliability*, vol. 11, no. 1, pp. 39-54, 2001.
[14] L.C. Briand, "A Critical Analysis of Empirical Research in Software Testing," *Proc. Int'l Symp. Empirical Software Eng. and Measurement*, Sept. 2007.
[15] L.C. Briand, M. Di Penta, and Y. Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Trans. Software Eng.*, vol. 30, no. 11, pp. 770-783, Nov. 2004.
[16] L.C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and Systems Modeling*, vol. 1, no. 1, pp. 10-42, 2002.
[17] L.C. Briand, Y. Labiche, and J. Cui, "Automated Support for Deriving Test Requirements from UML Statecharts," *J. Software and Systems Modeling*, special issue, vol. 4, no. 4, pp. 399-423, 2005.
[18] L.C. Briand, Y. Labiche, and Q. Lin, "Improving State-Based Coverage Criteria Using Data Flow Information," *Proc. IEEE Int'l Conf. Software Reliability Eng.*, pp. 95-104, 2005.
[19] L.C. Briand, Y. Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software—Practice and Experience*, vol. 33, no. 7, pp. 637-672, 2003.
[20] L.C. Briand, Y. Labiche, and Y. Wang, "Using Simulation to Empirically Investigate State Coverage Criteria Based on Statecharts," *Proc. ACM Int'l Conf. Software Eng.*, pp. 86-95, May 2004.
[21] B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java.* Prentice Hall, 2004.
[22] P. Chevalley and P. Thevenod-Fosse, "Automated Generation of Statistical Test Cases from UML State Diagrams," *Proc. 25th Ann. Int'l Computer Software and Applications Conf.*, pp. 205-214, 2001.
[23] P. Chevalley and P. Thevenod-Fosse, "An Empirical Evaluation of Statistical Testing Designed from UML State Diagrams: The Flight Guidance System Case Study," *Proc. 12th Int'l Symp. Software Reliability Eng.*, pp. 254-263, 2001.
[24] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Software Eng.*, vol. 4, no. 3, pp. 178-187, May 1978.
[25] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, second ed. Lawrence Erlbaum Assoc. Inc., 1988.
[26] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Verification of Results in Software Maintenance through External Replication," *Proc. Int'l Conf. Software Maintenance*, pp. 50-57, 1994.
[27] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34-41, Apr. 1978.
[28] J.L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*, 1999.
[29] T. Dybå, V.B. Kampenes, and D.I.K. Sjøberg, "A Systematic Review of Statistical Power in Software Engineering Experiments," *Information and Software Technology*, vol. 48, no. 8, pp. 745-755, 2006.
[30] G.A. Ferguson and Y. Takane, *Statistical Analysis in Psychology and Education*, sixth ed. McGraw-Hill Ryerson Limited, 2005.
[31] P.G. Frankl and S.N. Weiss, "An Experimental Comparison of the Effectiveness of the All-Uses and All-Edges Adequacy Criteria," *Proc. Fourth Symp. Software Testing, Analysis and Verification*, pp. 154-164, Oct. 1991.
[32] P.G. Frankl, S.N. Weiss, and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *Systems and Software*, vol. 38, no. 3, pp. 235-253, 1997.
[33] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley Professional, 2000.
[34] R.J. Harris, *A Primer of Multivariate Statistics.* Lawrence Erlbaum Assoc., 2001.

[35] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian J. Statistics,* vol. 6, pp. 65-70, 1979.

[36] N. Holt, B. Anda, K. Asskildt, L.C. Briand, J. Endresen, and S. Frøystein, "Experiences with Precise State Modeling in an Industrial Safety Critical System," *Proc. Critical Systems Development Using Modeling Languages, Workshop in Conjunction with UML '06,* pp. 68-77, 2006.

[37] R. Holt, W.D. Boehm-Davis, and A.C. Shultz, "Mental Representations of Programs for Student and Professional Programmers," *Empirical Studies of Programmers: Second Workshop,* pp. 33-46, Ablex Publishing Corp., 1987.

[38] H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software Testing, Verification and Reliability,* vol. 10, no. 4, pp. 203-227, 2000.

[39] M. Höst, B. Regnell, and C. Wohlin, "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," *Empirical Software Eng.,* vol. 5, no. 3, pp. 201-214, 2000.

[40] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* 1994.

[41] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, 2003.

[42] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 2004.

[43] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines—A Survey," *Proc. IEEE,* vol. 84, no. 8, pp. 1090-1123, Aug. 1996.

[44] P.S. Levy and S. Lemeshow, *Sampling of Populations: Methods and Applications,* third ed. Wiley, 1999.

[45] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-Class Mutation Operators for Java," *Proc. 13th Int'l Symp. Software Reliability Eng.,* Nov. 2002.

[46] Y.-S. Ma and J. Offutt, "Description of Method-Level Mutation Operators for Java," http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf, 2010.

[47] Y.-S. Ma, J. Offutt, and Y.R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification and Reliability,* vol. 15, no. 2, pp. 97-133, 2005.

[48] B. Marick, *Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing.* Prentice-Hall, 1985.

[49] P. McMinn and M. Holcombe, "Evolutionary Testing Using an Extended Chaining Approach," *Evolutionary Computation,* vol. 14, no. 1, pp. 41-64, 2006.

[50] B. Meyer, "Applying Design by Contract," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.

[51] S. Mouchawrab, L.C. Briand, Y. Labiche, and M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," Carleton Univ. TR SCE-08-09, 2009.

[52] K.R. Murphy and B. Myors, *Statistical Power Analysis: A Simple and General Model for Traditional and Modern Hypothesis Tests.* Lawrence Erlbaum, 1998.

[53] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Trans. Software Eng.,* vol. 32, no. 3, pp. 140-155, Mar. 2006.

[54] A.J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," *Proc. Second Int'l Conf. Unified Modeling Language,* pp. 416-429, 1999.

[55] A.J. Offutt and W.M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *J. Software Testing, Verification and Reliability,* vol. 4, no. 3, pp. 131-154, 1994.

[56] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating Test Data from State-Based Specifications," *J. Software Testing, Verification and Reliability,* vol. 13, no. 1, pp. 25-53, 2003.

[57] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability,* vol. 7, no. 3, pp. 165-192, 1997.

[58] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Fuctional Tests," *Comm. ACM,* vol. 31, no. 6, pp. 676-686, 1988.

[59] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage Measurement Experience during Function Test," *Proc. 15th Int'l Conf. Software Eng.,* 1998.

[60] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One Evaluation of Model-Based Testing and Its Automation," *Proc. Int'l Conf. Software Eng.,* pp. 392-401, 2005.

[61] F. Shull, J. Singer, and D.I.K. Sjøberg, *Guide to Advanced Empirical Software Engineering.* Springer, 2008.

[62] S. Siegel, *Non-Parametric Statistics for the Behavioral Sciences.* McGraw-Hill, 1956.

[63] S. Sinha and M.J. Harrold, "Analysis and Testing of Programs with Exception Handling Constructs," *IEEE Trans. Software Eng.,* vol. 26, no. 9, pp. 849-871, Sept. 2000.

[64] L.E. Toothaker, *Introductory Statistics for the Behavioral Sciences,* second ed. McGraw-Hill College, 1986.

[65] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2006.

[66] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Professional, 2004.

[67] E. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 121-128, Feb. 1990.

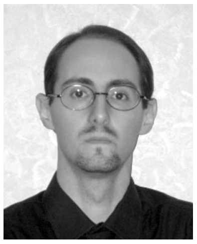
[68] E. Weyuker, T. Goradia, and A. Singh, "Automatically Generating Test Data from a Boolean Specification," *IEEE Trans. Software Eng.,* vol. 20, no. 5, pp. 353-363, May 1994.

[69] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction.* Kluwer, 2000.

Samar Mouchawrab received the BEng degree in software engineering from the Lebanese University in July 1997 and the MSc degree in software for telecommunications from the Institut National de la Recherche Scientifique, University of Quebec, in October 1999. She is currently working toward the PhD degree at Carleton University, Canada, and is a member of the Software Quality Laboratory team there. Her research interests include software testing empirical studies and software testing processes. She had worked for Nortel Networks as a software designer and for Logica as a team leader of a software verification team.

Lionel C. Briand is a group leader at the Simula Research Laboratory and a professor at the University of Oslo, Norway, leading projects on software verification and validation in collaboration with industry. Before that, he was on the Faculty of the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was a full professor and held the Canada Research Chair in Software Quality Engineering. He has also been the Software Quality Engineering Department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE, and ACM conferences. He is the coeditor-in-chief of *Empirical Software Engineering* (Springer) and is a member of the editorial boards of *Systems and Software Modeling* (Springer) and *Software Testing, Verification, and Reliability* (Wiley). He was on the board of the *IEEE Transactions on Software Engineering* from 2000 to 2004. His research interests include model-driven development, testing and quality assurance, and empirical software engineering. He is a fellow of the IEEE and a registered Canadian engineer (Ontario).

Yvan Labiche received the BSc degree in computer system engineering from the Graduate School of Engineering, Centre Universitaire des Science et Techniques (CUST), Clermont-Ferrand, France, the master's degree in fundamental computer science and production systems from the Université Blaise Pascal, Clermont-Ferrand, France, in 1995, and the PhD degree in software engineering from LAAS/CNRS in Toulouse, France, in 2000. He worked with Aerospatiale Matra Airbus (now EADS Airbus) on the definition of testing strategies for safety-critical, onboard software, developed using object-oriented technologies. In January 2001, he joined the Department of Systems and Computer Engineering at Carleton University, Canada, as an assistant professor. His research interests include object-oriented analysis and design, software verification, validation, and testing, and empirical software engineering. He is a member of the IEEE.



Massimiliano Di Penta received the laurea and PhD degrees in computer engineering in 1999 and 2003, respectively. He is an assistant professor in the Department of Engineering at the University of Sannio, Italy. His research interests include software maintenance and evolution, reverse engineering, empirical software engineering, search-based software engineering, and service-centric software engineering. He is the author of more than 120 papers published in journals, conferences, and workshops. He serves and has served on the organizing and program committees of several conferences such as ICSE, ICSM, ICPC, CSMR, GECCO, ICEIS, MSR, SCAM, SEKE, STEP, WCRE, and many other workshops. He was the general chair of WSE 2008, general cochair of WCRE 2008, and program cochair of SSBSE 2008, WCRE 2006 and 2007, IWPSE 2007, WSE 2007, SCAM 2006, STEP 2005, and of other workshops. He is a steering committee member of ICPC, SCAM, CSMR, and WCRE. He is on the editorial board of *Empirical Software Engineering* (Springer). He is a member of the IEEE, the IEEE Computer Society, and the ACM.