Data-driven Mutation Analysis for Cyber-Physical Systems[*]

Enrico Viganò[*], Oscar Cornejo[*], Fabrizio Pastore[*], Lionel Briand[*†]
[*]University of Luxembourg, [†]University of Ottawa
Luxembourg (Luxembourg), Ottawa (Canada)
{enrico.vigano,oscar.cornejo,fabrizio.pastore,lionel.briand}@uni.lu,lbriand@uottawa.ca

ABSTRACT

Cyber-physical systems (CPSs) typically consist of a wide set of integrated, heterogeneous components; consequently, most of their critical failures relate to the interoperability of such components. Unfortunately, most CPS test automation techniques are preliminary and industry still heavily relies on manual testing. With potentially incomplete, manually-generated test suites, it is of paramount importance to assess their quality. Though mutation analysis has demonstrated to be an effective mean to assess test suites' quality in some specific contexts, we lack approaches for CPSs. Indeed, existing approaches do not target interoperability problems and cannot be executed in the presence of black-box or simulated components, a typical situation with CPSs.

In this paper, we introduce *data-driven mutation analysis*, an approach that consists in assessing test suites' quality by verifying if they detect interoperability faults simulated by mutating the data exchanged by software components. Also, we describe a data-driven mutation analysis technique (*DaMAT*) that automatically alters the data exchanged through data buffers. Our technique is driven by fault models in tabular form where engineers specify how to mutate data items by selecting and configuring a set of mutation operators.

We have evaluated *DaMAT* with CPSs in the space domain; specifically, the test suites for the software systems of a microsatellite and nanosatellites launched on orbit last year. Our results show that the approach effectively detects test suite shortcomings, is not affected by equivalent and redundant mutants, and entails acceptable costs.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Mutation analysis, CPS, CPS Interoperability, Integration testing

1 INTRODUCTION

Cyber Physical Systems (CPSs) are heterogeneous systems that integrate computation, networking, and physical processes that are deeply interlaced [34]. In CPSs, conformance with requirements is verified through test cases executed at different development stages, based on available development artifacts [49]. In this paper, we focus on the identification of faults in the executable software to be deployed on the CPS, and thus target software-in-the-loop (SIL) and hardware-in-the-loop (HIL) testing.

When software systems are large and integrate a diverse set of components, it is difficult to ensure that the test suite can detect any latent severe fault. To ensure test suites' quality, standards for safety-critical software provide methodological guidance, for example through structural coverage adequacy; however, those strategies do not directly measure the fault detection capability of a test suite. A more direct solution to evaluate test suites' quality is *mutation analysis* [31, 44]. It consists of automatically generating faulty software versions and computing the mutation score, that is, the percentage of faulty software versions detected (i.e., leading to a test failure). Mutation analysis is a good candidate to assess test suites' quality because there is a strong association between high mutation scores and high fault revelation capability for test suites [45].

Most mutation analysis techniques rely on the generation of faulty software through mutation operators that modify the software implementation (either the source or the executable code). Unfortunately, such techniques suffer from two major limitations which are critical in the CPS context: (1) they cannot identify problems related to the interoperability of integrated components (integration testing) and (2) they can be applied only to components that can be executed in the development environment. In CPSs, major problems may arise because of the lack of *interoperability of integrated components* [22, 33], mainly due to the wide variety and heterogeneity of the technologies and standards adopted. Also, there is limited work on integration testing automation [1], thus forcing companies to largely rely on manual approaches, which are error prone and likely to lead to incomplete test suites. It is thus of fundamental importance to ensure the effectiveness of test suites with respect to detecting interoperability issues, for example by making sure test cases trigger the exchange of all possible data items and report failures when erroneous data is being exchanged by software components. For example, the test suite for the control software of a satellite shall identify failures due to components working with different measurement systems [38]. Unfortunately, well known, code-driven mutation operators (e.g., the sufficient set [13, 15]) simulate algorithmic faults by introducing small changes into the source code and are thus unlikely to simulate interoperability problems resulting in exchanges of erroneous data.

The second limitation of code-driven mutation analysis approaches concerns *the incapability of injecting faults into black-box components* whose implementation is not tested within the development environment (e.g., because it is simulated or executed on the target hardware). For example, in a satellite system, such components include the control software of the Attitude Determination And Control System (ADCS), the GPS, and the Payload Data Handling Unit (PDHU). During SIL testing, the results generated by such components (e.g., the GPS position) are produced by a simulator. As for HIL testing, these components are directly executed on the

[*]Paper submitted for peer review to ICSE 2022 - The 44th International Conference on Software Engineering. Do not distribute.

target hardware and cannot be mutated, either because they are off-the-shelf components or to avoid damages potentially introduced by the mutation.

An alternative to code-driven mutation analysis approaches are model-based ones, which mutate models of the software under test (SUT). Unfortunately, existing approaches do not include strategies to simulate interoperability problems; also, their primary objective is to support test generation not the evaluation of test suites [2, 8, 18, 29]. Furthermore, model-based test generation may not be cost-effective if detailed models of the system under test are not available—which is the case of system models produced in early development stages—and lack key information required for testing (e.g., which telecommands shall trigger a state transition in a satellite system).

To address the above-mentioned limitations, we propose *data-driven mutation analysis*, a new mutation analysis paradigm that alters the data exchanged by software components in a CPS to evaluate the capability of a test suite to detect interoperability faults. Also, we present a technique, *data-driven mutation analysis with tables* (*DaMAT*), to automate data-driven mutation analysis by relying on a fault model that captures, for a specific set of components, both the characteristics of the data to mutate (e.g., the size and structure of the messages generated by the ADCS) and the types of fault that may affect such data (e.g., a value out of the nominal range). The latter is formalized as a set of parameterizable mutation operators. Based on discussions with practitioners, to simplify adoption, we rely on fault models in tabular form where each row specifies, for a given data item, what mutation operator (along with its corresponding parameter values) to apply to which elements of the data item. At runtime, *DaMAT* modifies the data exchanged by components according to the provided fault model (e.g., replaces a nominal voltage value with a value out of the nominal range).

We performed an empirical evaluation of *DaMAT* to determine the effectiveness, feasibility, and applicability of data-driven mutation analysis for evaluating test suites. Our benchmark consists of software for CPSs in the space domain provided by our industry partners, which are an intergovernmental space agency, a world-renowned manufacturer and supplier of nanosatellites, and a European developer of infrastructure products (e.g., microsatellites) and solutions for space. More specifically, the benchmark includes (1) the on-board embedded software system for *ESAIL* [52], a maritime microsatellite recently launched into space, and (2) a configuration library used in constellations of nanosatellites [11]. Our empirical results show that *DaMAT* (1) successfully identifies different types of shortcomings in test suites, (2) prevents the introduction of equivalent and redundant mutants, and (3) is practically applicable in the CPS context.

2 BACKGROUND AND RELATED WORK

Data-driven mutation analysis evaluates the effectiveness of a test suite in detecting **interoperability faults**. The CPS literature reports on four different interoperability types [22]: technical (which concerns communication protocols and infrastructure), syntactic (which concerns data format), semantic (which concerns the exchanged information, that is, errors in the processing of exchanged data), and cross-domain interoperability (which concerns interaction through business process languages such as BPEL [40]). Technical and syntactic interoperability are provided by off-the-shelf hardware and libraries (not tested by CPS developers) while cross-domain interoperability concerns systems integrated in online services (e.g., energy plants) but is out of scope for the type of CPSs we target in this work, which are safety-critical CPSs like flight systems, robots, and automotive systems. In this paper, we thus focus on *semantic interoperability* faults, that is, faults that affect CPS components integration and are triggered (i.e., lead to failures) in the presence of specific subsets of the data that might be exchanged by CPS components. We thus aim to ensure that a test suite fails when the data exchanged by CPS components is not the one specified by test cases (e.g., through simulator configurations). Related work includes mutation analysis [31, 44] and fault injection [39] techniques.

Mutation analysis concerns the automated generation of faulty software versions (i.e., mutants) through automated procedures called mutation operators [31, 44]. The effectiveness of a test suite is measured by computing the mutation score, which is the percentage of mutants leading to failures when exercised by the test suite.

Mutation operators introduce syntactical changes into the code of the SUT. The *sufficient set of operators* is implemented by most mutation analysis toolsets [4, 35, 41, 41, 50]. Unfortunately, these operators simulate faults concerning the implementation of algorithms (e.g., a wrong logical connector), which is usually tested in unit test suites that, by definition, do not exercise the communication among components, our target in this paper. Also, as stated in the Introduction, such operators can't be used to generate faulty data with simulated or off-the-shelf components. *Higher-order* mutation analysis [28], which simply combines multiple operators, has the same limitations.

Components integration is targeted by interface [14], integration [26], contract-based [32], and system-level mutation analysis [36]. The former three assess the quality of integration test suites by introducing changes that concern function invocations (e.g., switch function arguments) and inter-procedural data-flow (e.g., alter assignments to variables returned to other components); they can simulate integration faults in units integrated with API invocations but not interoperability problems concerning larger components communicating through channels (e.g., network). System-level mutation relies on operators for GUI components, which are out of our scope, and configuration files, by applying simple mutations, such as deleting a line of text, and are unlikely to lead to interoperability problems.

Fault injection techniques simulate the effect of faults by altering, at runtime, the data processed by the SUT [39]. Faults are introduced according to a fault model that describes the type of fault to inject, the timing of the injection, and the part of the system targeted by the injection. Different from data-driven mutation analysis, fault injection techniques aim to stress the robustness of the software, not assess the quality of its test suites.

Faults affecting components' communication, CPU, or memory can simulated by performing bit flips [7, 12, 27, 53]. Communication faults are simulated also by duplicating or deleting packets, altering their sequence, or introducing incorrect identifiers, checksums, or counters [19, 20]. Faults affecting signals can be simulated by

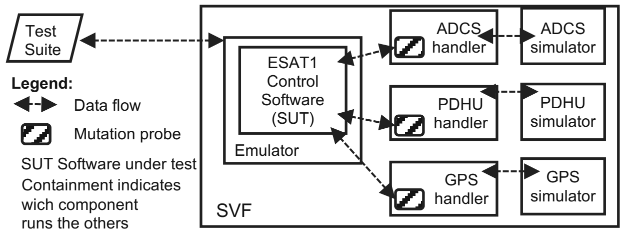


Figure 1: Data mutation probes integrated into ESAIL.

shifting the signal or increasing the number of signal segments [37]. The largest set of faults affecting data exchanged through files or byte streams is simulated by Peach [46], which includes also protocol-specific fault injection procedures such as replacing host names with randomly generated ones. In general, although existing techniques may simulate a large set of faults they do not cover all the CPS interoperability faults (see Section 3.2).

Approaches performing fault injections other than bit flips require a model of the data to modify. The modelling formalisms adopted for this purpose are grammars [9, 21, 23, 24], UML class diagrams [19, 20], or block models [46, 47]. Grammars are used to model textual data (e.g., XML), which is seldom exchanged by CPS components because of parsing cost. Block models enable specifying the representation to be used for consecutive blocks of bytes, which makes them applicable to a large set of systems; however, existing block model formalisms rely on the XML format, which is expensive to process and thus not usable with real-time systems [46, 47]. The UML class diagram is a formalism that enables the specification of complex data structures and data dependencies [19, 20]; however, it requires loading the data as UML class diagram instances, which is too expensive for real-time systems.

To summarize, the modification of the data exchanged by software components enables the simulation of communication and, therefore, semantic interoperability faults. Test suites can thus be assessed by relying on fault injection techniques to mutate data. However, existing fault injection techniques do not target mutation analysis; consequently, we lack methods for the specification of fault models and metrics for the assessment of test suites. Also, a larger set of procedures for the modification of data is needed. Finally, block models can effectively capture the structure of the data to modify but formalisms not relying on XML are needed. Our paper addresses such limitations.

3 APPROACH

3.1 Overview

Data-driven mutation analysis aims to evaluate the effectiveness of a test suite in detecting semantic interoperability faults. It is achieved by modifying (i.e., mutating) the data exchanged by CPS components. It generates *mutated data* that is representative of data that might be observed at runtime in the presence of a component that behaves differently than expected in the test case; also, it mutates data that is not automatically corrected by the software (e.g., through cyclic redundancy check codes) and thus causes software failures (i.e., the mutated data shall have a different semantic than

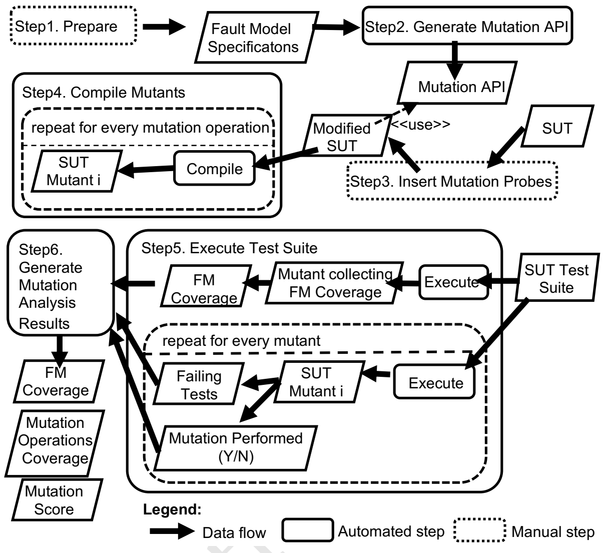


Figure 2: The *DaMAT* process.

the original data). For these reasons, data mutation is driven by a fault model specified by the engineers based on domain knowledge.

Although different types of fault models might be envisioned, in this paper we propose a technique (*data-driven mutation analysis with tables*, *DaMAT*), which automates data-driven mutation analysis by relying on a tabular block model, itself tailored to the SUT through predefined mutation operators. To concretely perform data mutation at runtime, *DaMAT* relies on a set of *mutation probes* that shall be integrated by software engineers into the software layer that handles the communication between components. The runtime behaviour of mutation probes (i.e, what data shall be mutated and how) is driven by the fault model. Thus, *DaMAT* can automatically generate the implementation of mutation probes from the provided fault model. Depending on the CPS, probes might be inserted either into the SUT, into the simulator infrastructure, or both. Figure 1 shows the architecture of the ESAIL satellite system (one of the subjects considered in our empirical evaluation) with mutation probes integrated into the SVF[1] functions that handle communication with external components (PDHU, GPS, and ADCS in this case).

DaMAT works in six steps, which are shown in Figure 2. In Step 1, based on the provided methodology and predefined mutation operators, the engineer prepares a fault model specification tailored to the SUT. In Step 2, *DaMAT* generates a mutation API with the functions that modify the data according to the provided fault model. In Step 3, the engineer modifies the SUT by introducing mutation probes (i.e., invocations to the mutation API) into it. In Step 4, *DaMAT* generates and compiles mutants. Since the *DaMAT* mutation operators may generate mutated data by applying multiple mutation procedures, *DaMAT* may generate several mutants, one for each mutation operation (i.e., a mutation procedure configured for a data item, according to our terminology, see Section 3.5). In

[1]Software Validation Facility [30]; it usually includes one or more simulators, an emulator to run the code compiled for the target hardware, and test harnesses.

Step 5, *DaMAT* executes the test suite with all the mutants including a mutant (i.e., the coverage mutant) which does not modify the data but traces the coverage of the fault model. In Step 6, *DaMAT* generates mutation analysis results.

In the following sections we describe the structure of our fault model and each step of *DaMAT*.

3.2 Fault Model Structure

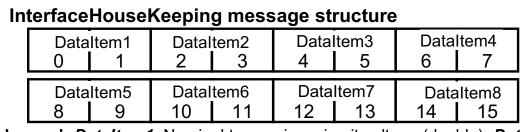
The *DaMAT* fault model enables the specification of the format of the data exchanged between components along with the type of faults that may affect such data. In this paper, we refer to the data exchanged by two components as *message*; also, each CPS component may generate or receive different *message types*. For a single CPS, more than one fault model can be specified. For example, in the case of ESAIL we have defined one fault model for every message type that could be exchanged by the three components under test (i.e., ADCS, PDHU, and GPS). In total, for ESAIL, we have 14 fault models, 10 for the communication concerning ADCS (we have 10 different message types), 3 for PDHU, and 1 for GPS.

The *DaMAT* fault model enables the modelling of data that is exchanged through a specific data structure: the data buffer. This was decided because it is a simple and widely adopted data structure for data exchanges between components in CPS. Also, more complex data structures (e.g., hierarchical ones like trees) are often flattened into data buffers in order to be exchanged by different components (e.g., through the network). When the CPS software is implemented in C or C++ (common CPS development languages) data buffers are implemented as arrays. Figure 3 shows three block diagrams representing (part of) the buffer structure used to exchange messages of type InterfaceHouseKeeping and InterfaceStatus in ESAIL.

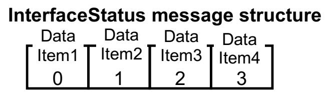
A data buffer is characterized by a *unit size* that specifies the dimension, in bytes, of the single cell of the underlying array and a *buffer size*, which specifies the total number of units belonging to the buffer. Each data buffer can contain one or more *data items*; the size of data items may vary as they may span over multiple units. Also, each data item is interpreted by the CPS software according to a specific *representation* (e.g., integer, double, etc.). In ESAIL, the unit size is one byte and the data items may span over one or two buffer units (see Figure 3).

The *DaMAT* fault model enables engineers to specify (1) the *position* of each data item in the buffer, (2) their *span*, and (3) their *representation type*. Our current implementation supports six data representation types: int, long int, float, double, bin (i.e., data that should be treated in its binary form), hex (i.e., data that should be treated as hexadecimal). Further, for each data item, *DaMAT* enables engineers to specify one or more data faults using the mutation operator identifiers. For each operator, the engineer shall provide values for the required configuration parameters.

Table 1 provides the list of mutation operators included in *DaMAT* along with their description. The *DaMAT* mutation operators generate *mutated data item instances* through one or more *mutation procedures*, which are the functions that generate a mutated data item instance given a correct data item instance observed at runtime. For example, the *VAT* operator includes only one mutation procedure (i.e., setting the current value above the threshold) while the *VOR* operator includes two mutation procedures, which are (1)



Legend: ***DataItem1***: Nominal transceiver circuit voltage (double). ***DataItem2***: Redundant transceiver circuit voltage (double). ***DataItem3***: Internal power supply measured with nominal ADC (double). ***DataItem4***: Internal power supply measured with redundant ADC (double). ***DataItem5***: Main board PCB temperature measured by sensor 1 (double). ***DataItem6***: Main board PCB temperature measured by sensor 2 (double). ***DataItem7***: Sun sensor board PCB temperature from sensor 3 (double). ***DataItem8***: Sun sensor board PCB temperature from sensor 4 (double).



Legend: ***DataItem1***: Bit 0 to 2, information about last reset. Bit 3 indicates if ADCS is ready. Bit 4 indicates an OBC communication error. Bit 5 indicates a communication error with the connected units (binary). ***DataItem2***: Each bit indicates the unit in error (Gyroscope, Reaction Wheel, Magnetorqer, Magnetometer, Sun Sensor) (binary). ***DataItem3***: Watchdog reset counter incremented at every reset (integer). ***DataItem4***: Device reset counter (integer).

Figure 3: Structure of data buffers in ESAIL.

replacing the current value with a value above the specified valid range and (2) replacing the current value with a value below the valid range. The operators VOR, BF, INV, and SS have been inspired by related work [20, 37, 46]; the operators VAT, VBT, FVAT, FVBT, FVOR, IV, ASA, and HV are a contribution of this paper and were derived and conceptualised as a result of discussion with domain experts. Although other data representation types (e.g., null terminated strings) and operators (e.g., replacement of a random char in a string) might be envisioned, in this paper, we focus on operators that are necessary in the CPS context, based on our experience. For example, CPS components are unlikely to exchange strings.

3.3 Fault Modelling Methodology (Step 1)

The fault model shall enable the specification of all possible interoperability problems in the SUT while minimizing equivalent and redundant mutants. Equivalent mutants have the same observable output as the original SUT. Instead, redundant mutants have the same observable output as other mutants. We use the term *observable output* to refer to any output that can be verified by the test suite. The equivalent or redundant nature of a mutant depends on the equivalence relation for observable outputs (i.e., how to determine if two outputs are the same). In a testing context, such equivalence relation depends on the type of testing being performed. For example, system test cases, different than unit test cases, are unlikely to verify the values of all the state variables of the system and thus mutants that are nonequivalent for unit test suites might be considered equivalent for system test suites. For example, in satellite systems, the correctness of the GPS triangulation algorithm output is verified by unit test cases; system test cases, instead, verify if the software takes appropriate actions when the satellite is out of orbit. Consequently, slight changes in the coordinates communicated by the GPS component may not lead to any change in the observable output verified by the test suite.

We provide a set of guidelines for the definition of fault models that are summarized in Table 2. For guidance, we account for the nature of the data (i.e., numerical, categorical, ordinal, or binary) and their representation type. Also, for numerical data, we consider

Table 1: Data-driven mutation operators

Fault Class	Types	Parameters	Description
Value above threshold (VAT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	Replaces the current value with a value above the threshold T for a delta (Δ). It simulates a value that is out of the nominal case and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already above the threshold. *Data mutation procedure:* $v' = (T + \Delta)(if\, v \leq T); v' = v(otherwise);$
Value below threshold (VBT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	Replaces the current value with a value below the threshold T for a delta (Δ). It simulates a value that is out of the nominal case and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already below the threshold. *Data mutation procedure:* $v' = (T - \Delta)(if\, v \geq T); v' = v(otherwise)$
Value out of range (VOR)	I,L,F,D,H	MIN: minimum valid value MAX: maximum valid value Δ: delta, difference with respect to minimum/-maximum valid value	Replaces the current value with a value out of the range $[MIN; MAX]$. It simulates a value that is out of the nominal range and shall trigger a response from the system that shall be verified by the test case (e.g., the system may continue working but an alarm shall be triggered). Not applied if the value is already out of range. This was inspired by the *ARBC* operator [20]; however, *DaMAT* enables engineers to explicitly specify the delta. *Data mutation procedure 1:* $v' = (MIN - \Delta)(if\, MIN \leq v \leq MAX); v' = v(otherwise)$ *Data mutation procedure 2:* $v' = (MAX + \Delta)(if\, MIN \leq v \leq MAX); v' = v(otherwise)$
Bit flip (BF)	B	MIN: lower bit MAX: higher bit STATE: mutate only if the bit is in the given state (i.e., 0 or 1). VALUE: number of bits to mutate	A number of bits randomly chosen in the positions between MIN and MAX (included) are flipped. If STATE is specified, the mutation is applied only if the bit is in the specified state; the value −1 indicates that any state shall be considered for mutation. Parameter VALUE specifies the number of bits to mutate. This was inspired by the *BitFlipperMutator* operator [46]; however, *DaMAT* introduces the STATE parameter, which is not supported by related work. *Data mutation procedure:* the operator flips VALUE randomly selected bit if they are in the specified state.
Invalid numeric value (INV)	I,L,F,D,H	MIN: lower valid value MAX: higher valid value	Replace the current value with a mutated value that is legal (i.e., in the specified range) but different than current value. It simulates the exchange of data that is not consistent with the state of the system. It matches the *ARR* operator [20]. *Data mutation procedure:* Replace the current value with a different value randomly sampled in the specified range.
Illegal Value (IV)	I,L,F,D,H	VALUE: illegal value that is observed	Replace the current value with a value that is equal to the parameter *VALUE*. It matches the *ValidValuesMutator* operator [46]. *Data mutation procedure:* $v' = VALUE(if\, v \neq VALUE); v' = v(otherwise)$
Anomalous Signal Amplitude (ASA)	I,L,F,D,H	T: change point Δ: delta, value to add/remove VALUE: value to multiply	The mutated value is derived by amplifying the observed value by a factor V and by adding/removing a constant value Δ from it. It is used to either amplify or reduce a signal in a constant manner to simulate unusual signals. The parameter T indicates the observed value below which instead of adding we subtract . *Data mutation procedure:* $v' = T + ((v - T) * VALUE) + \Delta(if\ v \geq T); v' = T - ((T - v) * VALUE) - \Delta(if\ v < T);$
Signal Shift (SS)	I,L,F,D,H	Δ: delta, value by which the signal should be shifted	The mutated value is derived by adding a value Δ to the observed value. It simulates an anomalous shift in the signal. This was inspired by work on signal mutation [37]; however, *DaMAT* also enables engineers to rely on SS to increment (or decrement) counters and identifiers. *Data mutation procedure:* $v' = v + \Delta$
Hold Value (HV)	I,L,F,D,H	V: number of times to repeat the same value	This operator keeps repeating an observed value for V times. It emulates a constant signal replacing a signal supposed to vary. *Data mutation procedure:* $v' = previous\ v'(if\ counter \leq V); v' = votherwise$
Fix value above threshold (FVAT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	It is the complement of VAT and implements the same mutation procedure as VBT but we named it differently because it has a different purpose. Indeed, it is used to verify that test cases exercising exceptional cases are verified correctly. In the presence of a value above the threshold, it replaces the current value with a value below the threshold T for a delta Δ. *Data mutation procedure:* $v' = v(if\, v > T); V' = (T - \Delta)(otherwise)$
Fix value below threshold (FVBT)	I,L,F,D,H	T: threshold Δ: delta, difference with respect to threshold	It is the counterpart of FVAT for the operator VBT. *Data mutation procedure:* $v' = v(if\, v < T); v' = (T + \Delta)(otherwise)$
Fix value out of range (FVOR)	I,L,F,D,H	MIN: minimum valid value MAX: maximum valid value	It is the complement of VOR and implements the same mutation procedure as INV but we named it differently because it has a different purpose. Indeed, it is used to verify that test cases exercising exceptional cases are verified correctly. *Data mutation procedure:* $v' = v(if\, MIN \leq v \leq MAX); v' = random(MIN, MAX)(otherwise)$

Legend: I: INT, L: LONG INT, F: FLOAT, D: DOUBLE, B: BIN, H: HEX

Table 2: *DaMAT* fault modelling methodology

Data nature	Representation type	Dependencies	# of input partitions	Operators	Comments
numerical	I, L, F, D	stateless/stateful	2	[VAT,FVAT] or [VBT,FVBT]	Nominal below T Nominal above T
			3 or more	[VOR,FVOR]	
		stateful		INV	For valid range
				[VOR,FVOR]	For out of range
		signal		ASA, SS, HV	
categorical	I, H	N/A	N/A	IV	
	B	N/A	N/A	BF	
ordinal	I, H	N/A	N/A	ASA	
other	B	N/A	N/A	BF	

Legend: N/A not applicable.

the data dependencies, that is how data values depend on the previously observed values; we identified three categories: stateless (i.e., there are no dependencies between consecutive values), stateful (i.e, values depend on previous ones), and signal (i.e., values derive from a function of independent variables like time). Data dependencies determine the granularity of the mutation (i.e., with data dependencies, small differences shall be noticed); for non numerical data, we do not provide mutation operators with different granularities and data dependencies can be ignored.

For *stateless numerical data*, our guidelines are driven by input space partitioning concepts [3]. Indeed, given equivalence relations among outputs, it is unlikely that every change in *stateless numerical data* will result into nonequivalent mutants; however, we can partition the input domain into regions with equivalent values (partitions). Precisely, we rely on the *interface-based input domain modeling* approach [3]: for each data item we identify a number of input partitions (set of values or value ranges) according to the interface specifications of the interacting components. In our methodology, the number and type of mutation operators selected for stateless numerical data depend on the number of input partitions identified. With *two input partitions* (e.g., nominal and exceptional data values), engineers can rely either on the pair [VBT,FVBT] or the pair [VAT,FVAT]. With *three partitions*, engineers must configure one VOR and one FVOR operator. If a different delta (Δ) is considered for the upper and lower bounds, engineers may configure two pairs [VBT,FVBT] and [VAT,FVAT], for the lower and upper bound, respectively. In the presence of *more than three*

Table 3: Portion of the fault model specification for ESAIL

#	Fault Model	Position	Span	Type	Op	MIN	MAX	T	DELTA	STATE	VALUE
1	IfHK	8	2	DOUBLE	VAT	-	-	33.53	0.01	-	-
2	IfHK	8	2	DOUBLE	FVAT	-	-	33.53	0.01	-	-
3	IfHK	8	2	DOUBLE	VBT	-	-	24	1	-	-
4	IfHK	8	2	DOUBLE	FVBT	-	-	24	1	-	-
5	IfHK	10	2	DOUBLE	VAT	-	-	6	1	-	-
6	IfHK	12	2	DOUBLE	VOR	-20	50	-	1	-	-
7	IfHK	14	2	DOUBLE	VOR	-20	50	-	1	-	-
8	IfStatus	0	1	BIN	BF	3	3	-	-	0	1
9	IfStatus	0	1	BIN	BF	3	3	-	-	1	1
10	IfStatus	0	1	BIN	BF	4	4	-	-	0	1
11	IfStatus	0	1	BIN	BF	4	4	-	-	1	1
12	IfStatus	0	1	BIN	BF	5	5	-	-	0	1
13	IfStatus	0	1	BIN	BF	5	5	-	-	1	1
14	IfStatus	1	1	BIN	BF	0	0	-	-	0	1
15	IfStatus	1	1	BIN	BF	0	0	-	-	1	1
16	IfStatus	1	1	BIN	BF	1	1	-	-	0	1
17	IfStatus	1	1	BIN	BF	1	1	-	-	1	1
18	IfStatus	1	1	BIN	BF	2	2	-	-	0	1
19	IfStatus	1	1	BIN	BF	2	2	-	-	1	1
20	IfStatus	1	1	BIN	BF	3	3	-	-	0	1
21	IfStatus	1	1	BIN	BF	3	3	-	-	1	1
22	IfStatus	1	1	BIN	BF	4	4	-	-	0	1
23	IfStatus	1	1	BIN	BF	4	4	-	-	1	1

Note: a "-" is used for parameters not required to configure a mutation operator.

partitions, engineers shall configure one [VOR,FVOR] pair for each extra partition above three (e.g., two pairs in the case of five partitions). The parameter Δ is used to determine the partition to which the mutated data belongs.

In the presence of *stateful data*, replacement with random values in the valid range (i.e., the INV operator) will lead to nonequivalent mutants (e.g., because it leads to data values that are systematically different than the values expected for the current system state). Alternatively, the valid data range might be partitioned as for stateless data. However, to avoid redundant mutants, engineers should rely either on the INV operator or the partitioning of the valid data range. The effect of data outside the valid data range should instead be verified by means of the [VOR, FVOR] pair.

For *signal values*, depending on the shape of the expected signal, engineers should configure one operator among the ASA, SS, and HV. The configuration of more than one of these operators may lead to redundant mutants (e.g., because each of them triggers the same warning in the SUT).

With *categorical data* represented using *integers and hexadecimals*, engineers must configure one IV operator for each possible value; indeed, a change in the observed category shall trigger a different behaviour in the SUT. With categorical data in *binary form*, each bit indicates a specific class (e.g., the unit in error for the DataItem2 in the IFStatus message of Figure 3). To verify that the test suite can detect any possible category change, engineers must configure two BF operators for every bit (both MIN and MAX must coincide with the bit position), one operator must flip a bit when it is set (i.e., $STATE = 1$), and the last one when it is unset (i.e., $STATE = 0$).

For *ordinal data*, which is represented by means of either integers or hexadecimals, we suggest to apply the ASA operator with T being set to the middle point of the ordinal scale and Δ set to the step distance between consecutive data (usually 1). For data in binary form (e.g., pictures), engineers must configure a BF operator to flip a number of bits that is sufficient to alter the semantics of the data (e.g., introduce sufficient noise in images).

Table 3 provides a specification in tabular form (i.e, the format processed by *DaMAT*) of two fault models configured for the IfKH (i.e., Interface House Keeping) and IfStatus (i.e, Interface Status)

```
switch(message_type)
  case IfStatus:
    GetIfStatus(buffer);
    mutate_FM_IfStatus( buffer);
    break;
...
```

```
...
  case IfHouseKeeping:
    GetIfHouseKeeping(buffer);
    mutate_FM_IfHK( buffer );
    break;
```

Figure 4: Example of *DaMAT* mutation probes (in bold).

messages. In the fault models, each row captures the configuration of a mutation operator for a specific data item. For example, row number 5 indicates that *DaMAT* interprets as double the data inside the two buffer units starting at position 10 (units 10 and 11) and applies the VAT operator. Rows 1 and 2 show that, for a same numerical data item (i.e., the one covering units 8 and 9), we can apply both the VAT and VBT operators, using a different delta for each. Rows 2 and 4 show the FVAT and FVBT operators complementing the VAT and VBT operators in rows 1 and 3. They simulate the case in which data for the nominal cases is observed instead of data for exceptional cases, as visible in Table 1. Rows 8 to 23 show that different bits of a same data item can be targeted by different BF operators. Rows 8 to 13 concern binary categorical data with two categories each, thus we configured two BF each. Rows 14 to 23 concern binary categorical data with five categories; consequently, they present ten BF operators configured for the five categories.

3.4 Automated Generation of Mutation API (Step 2) and Probe Insertion (Step 3)

DaMAT automatically generates a *mutation API* to perform mutations at runtime. The API implements a set of functions (called *mutate_FM_<name>*) that mutate a data buffer according to the given fault model. These functions select the data item to mutate and the mutation procedure to apply based on the mutant under test (see Section 3.5).

The *DaMAT* mutation API works with C/C++ code; however it may be extended to deal with other programming languages. Since it is not possible to automatically determine which data buffer to mutate, *DaMAT* requires engineers to modify the source code of the CPS under test by introducing a mutation probe which consists of an invocation of the *DaMAT* function that mutates the data buffer according to a specific fault model. Note that the effort required by the engineer is minimal; indeed, the exchange of data between components is usually managed in a single location (e.g, the function that serializes the data buffer on the network) and thus it is usually sufficient to introduce one function call for each message type to mutate. Figure 4 shows how the implementation of ESAIL has been modified to add the mutation probes. The SVF function was modified to handle the message requests sent to the ADCS by inserting one mutation probe for each message type to mutate, e.g., IfStatus and IfHouseKeeping in Figure 4. Function *mutate_FM_IfStatus* is part of the generated mutation API; it loads the fault model *IfStatus* into memory (our API relies on a tree data structure) and then invokes the function *mutate*. The function *mutate* performs data-driven mutation according to the provided fault model; the implementation of *mutate* is part of the *DaMAT* toolset.

The behavior of function *mutate* depends on the value of a unique identifier (i.e., the *MutantID*) associated at compile time to the mutant; the *Mutant ID* univocally identifies the performed mutation operation (each mutant executes one mutation operation, see Section 3.5). At a high level, *mutate* performs four activities. First, it checks if the mutation should be performed (i.e., if the data buffer is targeted by the mutation operation identified with the *Mutant ID*). Second, it casts the data item instance targeted by the mutant to a support variable of the type specified in the fault model. Third, it mutates the data stored in the support variable; for each mutation operator, we have implemented a distinct set of instructions for each data representation type. Fourth, before terminating, the function *mutate* writes the mutated data back to the data buffer.

3.5 Automated Generation of Mutants (Step 4)

Consistent with code-driven mutation analysis, *DaMAT* generates one mutant for each mutation procedure of the mutation operators configured in the fault model. Each mutant performs exactly one *data mutation operation* (i.e., a data mutation procedure configured for a specific data item). For example, the specification in row 6 of Table 3 makes *DaMAT* generate two mutants: each mutant modifies the value of the data item starting at position 12 but one mutant replaces the current value with the value 51 (i.e., $50 + 1$) while the other replaces the current value with the value -21 (i.e., $-20 - 1$).

The mutant generation is invisible to the end-user who does not need to modify the source code further; indeed, we rely on a C macro to specify, at compile time, which mutation operation must be performed by every mutant. Mutants are generated by compiling the SUT multiple times, once for each mutation operation. At runtime, the mutate function executes only the mutation operation selected for the mutant under test.

3.6 Mutants Execution (Step 5)

As for code-driven mutation analysis, the test suite under analysis is executed iteratively with every data-driven mutant. At runtime, all the data items targeted by a mutant are mutated whenever the mutation preconditions hold (e.g., the STATE of the BF operator); we leave the mutation of a sampled subset of data item instances to future work [25, 54].

To speed up the mutation analysis process, the test suite under analysis is first executed with a special mutant that, instead of mutating data items, keeps trace of the fault models loaded by each test case; in other words, it traces what are the data types covered by each test case. The collected information enables the execution, for every mutant, of the subset of test cases that cover the message type targeted by the mutant, thus speeding up mutation analysis.

3.7 Mutation Analysis Results (Step 6)

Inspired by work on abstract mutation analysis [42], we have defined three metrics to evaluate test suites with data-driven mutation analysis: fault model coverage, mutation operation coverage, and mutation score. These metrics measure the frequency of the following scenarios: (case 1) the message type targeted by a mutant is never exercised, (case 2) the message type is covered by the test suite but it is not possible to perform some of the mutation operations (e.g., because the test suite does not exercise out-of-range cases), (case 3) the mutation is performed but the test suite does not fail. Different from code-driven mutation analysis, these three metrics enable engineers to distinguish between possible test suite shortcomings, including untested message types, uncovered input partitions, poor oracle quality, and lack of test inputs.

Fault model coverage (FMC) is the percentage of fault models covered by the test suite. Since we define a fault model for every message type exchanged by two components, it provides information about the extent to which the message types actually exchanged by the SUT are exercised and verified by the test suites. Since different component functionalities often require different message types, low fault model coverage may indicate that only a small portion of the integrated functionalities have been tested.

Mutation operation coverage (MOC) is the percentage of data items that have been mutated at least once, considering only those that belong to the data buffers covered by the test suite. It provides information about the input partitions covered for each data item; for example, the FVOR operator leads to two mutation operations, which are applied only if the observed value is outside range. Otherwise the two mutation operations will not be covered, thus enabling the engineer to identify such shortcoming in the test suite.

The *mutation score (MS)* is the percentage of mutants killed by the test suite (i.e., leading to at least one test case failure) among the mutants that target a fault model and for which at least one mutation operation was successfully performed. It provides information about the quality of test oracles; indeed, a mutant that performs a mutation operation and is not killed (i.e., is *live*) indicates that the test suite cannot detect the effect of the mutation (e.g., the presence of warnings in logs). Also, a low mutation score may indicate missing test input sequences. Indeed, live mutants may be due to either software faults (e.g., the SUT does not provide the correct output for the mutated data item instance) or the software not being in the required state (e.g., input partitions for data items are covered when the software is paused); in such cases, with appropriate input sequences, the test suite would have discovered the fault or brought the SUT into the required state. Both poor oracles and lack of inputs indicate flaws in the test case definition process (e.g., the stateful nature of the software was ignored).

4 EMPIRICAL EVALUATION

We address the following research questions:

RQ1. What are the types of test suite shortcomings identified by DaMAT*?* We aim to assess the effectiveness of *DaMAT* in identifying various test suite shortcomings, as described in Section 3.7. In other words, we want to know if mutation analysis based on *DaMAT* can provide clear guidance in terms of what to improve in a test suite.

RQ2. What is the impact of equivalent and redundant data-driven mutants on the mutation analysis process? In general, mutation analysis may lead to the generation of equivalent and redundant mutants. In the specific context of *DaMAT*, we analyze the extent of their impact on mutation scores.

RQ3. Is data-driven mutation feasible? To assess its feasibility in practice, we evaluate the cost of setting up data-driven mutation analysis (i.e., defining fault models and instrumenting the CPS with probes), the duration of the mutation analysis process, and the runtime overhead introduced during test case execution.

Table 4: Fault models and mutation operators.

Subject	Fault Models	Configured Operators	Mutation Operations
ESAIL-ADCS	10	142	172
ESAIL-GPS	1	23	23
ESAIL-PDHU	3	29	29
LIBParam	6	80	80

Table 5: Mutation Analysis Results.

Subject	# FMs	FMC	#MOs-CFM	#CMOs	MOC	Killed	Live	MS
ESAIL-ADCS	10	90.00%	135	100	74.00%	45	55	45.00%
ESAIL-GPS	1	100.00%	23	22	95.65%	21	1	95.45%
ESAIL-PDHU	3	100.00%	29	24	82.76%	24	0	100.00%
LIBParam	6	100.00%	80	73	91.25%	28	45	38.36%

CMO=Covered Mutation Operation, MOs-CFM=Mutation Operations in covered FMs.

4.1 Subjects of the study

To assess our research questions, we considered CPS components used in cubesat constellations and in *ESAIL*, which is a microsatellite [11]. More precisely, we consider *LIBParam*, which is a client-server component to manage configuration parameters in cubesats. Also, we examine three *ESAIL* software sub-systems (1) the Attitude Determination And Control System (*ESAIL-ADCS*), the Global Positioning System (*ESAIL-GPS*), and the Payload Data Handling Unit (*ESAIL-PDHU*). These are representative examples of CPS control and utility software, as well as sensor and actuator drivers.

We rely on *DaMAT* to evaluate the *LIBParam* integration test suite by mutating the data exchanged between the client and server components of *LIBParam*. Similarly, *DaMAT* is used to evaluate how well the *ESAIL* test suite covers interoperability problems affecting the integration between the control software of *ESAIL* (hereafter, CSW) and the *ESAIL-ADCS*, *ESAIL-PDHU*, and *ESAIL-GPS* components. We thus mutate the data exchanged between *ESAIL* CSW and these three components. Since each of these sub-systems have a different purpose (i.e., their data is processed by distinct CSW functions and affect distinct *ESAIL* features) we treat them as distinct case study subjects although they are tested using the same test suite. We focus on the *ESAIL* test suite that makes use of an SVF to simulate the *ESAIL-ADCS*, *ESAIL-PDHU*, and *ESAIL-GPS* components. The main reason is that these three components can only be executed on the target hardware and thus most of the scenarios involving them are tested in a simulated environment first. We do not mutate messages or data items that are tested only with HIL.

In the case of *LIBParam*, we inject mutation probes into the *LIBParam* server to mutate both received and generated messages. For *ESAIL*, we insert mutation probes into the SVF that mutate the messages it generates; we avoid mutating the messages received by the SVF because such mutations may lead to input data it does not support. ESAIL features 74 kLoC and its SVF 65 kLoC. The ESAIL test suite includes 384 test cases, takes approximately 10 hours to execute, and relies on three simulated *ESAIL-SVF* sub-systems (i.e., *ESAIL-ADCS*, *ESAIL-GPS*, and *ESAIL-PDHU*). Instead, *LIBParam* contains 3 kLoC and is tested through an integration test suite which is composed by 170 test cases. The *LIBParam* integration test suite takes approximately 1 minute to execute. By considering both a quick integration test suite and an extensive system test suite, we aim to cover the diversity of scenarios in which our approach can be applied.

4.2 Experimental Setup

With the support of our industry partners, we relied on the systems' specification documents to define the fault models for each subject.

Table 4 provides information about the fault models. The fault models (FMs) for the *ESAIL-ADCS* include multiple configurations (*Configured operators*) of eight mutation operators: BF, VAT, VBT, VOR, IV, FVOR, FVBT, and FVAT. The *ESAIL-PDHU* fault models include four operators: BF, IV, VAT and FVAT. Even though the *ESAIL-GPS* fault model concerns only one data type, it makes use of six operators: ASA, HV, IV, SS, VAT, and FVAT. For *LIBParam*, we relied on the operators BF, HV, IV, SS, VAT, and FVAT. All the mutation operators provided by *DaMAT* have been used in at least one fault model, which shows their usefulness. They have led to 172 mutation operations for *ESAIL-ADCS*, 23 for *ESAIL-GPS*, 29 for *ESAIL-PDHU*, and 80 for *LIBParam*; the number of configured operators and mutation operations match except when we rely on VOR and FVOR.

We performed our experiments using an HPC cluster with Intel Xeon E5-2680 v4 (2.4 GHz) nodes.

4.3 RQ1 - Approach effectiveness

We analyzed the extent to which *DaMAT* helps identify limitations in test suites. For each subject, we inspected uncovered fault models, uncovered mutation operations, and live mutants. We then analyzed how they could potentially be explained by the types of shortcomings introduced in Section 3.7: untested message types (UMT), uncovered input partitions (UIP), poor oracle quality (POQ), and lack of test inputs (LTI). To achieve the above, we proceeded as follows. For each uncovered fault model, we discussed with developers if the functionality triggering the exchange of the targeted message was tested by the test suite. For uncovered mutation operations, we discussed with engineers if they match an uncovered input partition. For live mutants, we determined if they could be killed by improving test oracles (see how equivalent mutants are detected for RQ2).

To address RQ1, based on the above analysis, we discuss below how our metrics (i.e., *fault model coverage - FMC*, *mutation operation coverage - MOC*, and *mutation score - MS*) relate to the predefined shortcoming categories (e.g., a low mutation score may indicate missing test oracles). Further, to understand how variations in test effectiveness could be explained, we investigate how our metrics relate to the number of functionalities under test (i.e., the number of fault models - *FM*), the number of mutation operations (*MO*), and the number of covered mutation operations (*CMO*), respectively. To get an idea of observable trends, we compute the Spearman's correlation coefficients between them, hereafter denoted ρ_{FM}, ρ_{MO}, ρ_{CMO}.

Results. Table 5 reports the mutation analysis results according to the metrics introduced in Section 3.7. In Table 6, we report how uncovered fault models, uncovered mutation operations, and live

Table 6: Shortcomings of CPSs test suites.

Short-coming	***ESAIL-ADCS***			***ESAIL-GPS***			***ESAIL-PDHU***			***LIBParam***		
	UF	**UM**	**LM**	**UF**	**UM**	**LM**	**UF**	**UM**	**LM**	**UF**	**UM**	**LM**
UMT	1	-	-	-	-	-	-	-	-	-	-	-
UIP	-	35	-	-	1	-	-	5	-	-	7	-
POQ	-	-	55	-	-	1	-	-	-	-	-	45
LTI	-	-	-	-	-	-	-	-	-	-	-	-
Total	1	35	55	-	1	1	-	5	-	-	7	45

UF=Uncovered Fault model, UM=Uncovered Mutation operation, LM=Live mutant.

mutants are distributed with respect to the different shortcomings we noticed on each subject.

Concerning *fault model coverage*, *ESAIL-ADCS* reached a coverage of 90.00%, while *ESAIL-GPS*, *ESAIL-PDHU*, and *LIBParam* all achieved 100%. As expected, the much higher number of messages to test for *ESAIL-ADCS* leads to incomplete testing.

ESAIL-ADCS reached 74% *mutation operation coverage*. *ESAIL-GPS*, *ESAIL-PDHU*, and *LIBParam* achieved even higher coverage with 95.65%, 82.76%, and 91.25%, respectively. Since ρ_{MO} = -0.8, results suggest that lower mutation operation coverage is more likely when systems are more complex (i.e., there are many mutation operations, whose numbers depend on the number of input partitions).

Regarding *mutation scores*, we report 45.00% for *ESAIL-ADCS*, 95.45% for *ESAIL-GPS*, and 100.00% for *ESAIL-PDHU*. These results indicate a varying performance of the *ESAIL-SVF* test suite across sub-systems. *LIBParam* obtained a mutation score of 38.36%, indicating that only slightly more than a third of mutants are killed by the test suite. Given that ρ_{CMO} = -0.6, we conclude that the mutation score tends to be lower for complex systems with a large number of covered mutation operations.

Table 6 provides the shortcomings identified for all our subjects. Our analysis confirms that (1) uncovered fault models (i.e., low *FMC*) indicate lack of coverage for certain message types (*UMT*) and, in turn, the lack of coverage of a specific functionality (i.e., setting the pulse-width modulation in *ESAIL-ADCS*); (2) uncovered mutation operations (i.e., low *MOC*) highlight the lack of testing of input partitions (*UIP*); (3) live mutants (i.e., low *MS*) suggest poor oracle quality (*POQ*). In our case study systems the presence of live mutants was not explained by the lack of test inputs in the original test suite. Moreover, we have not uncovered latent faults, which is unsurprising given that all these systems went through all testing stages, including HIL, and are on orbit.

4.4 RQ2 - Equivalent and redundant mutants

As they potentially have significant impact on the applicability of any mutation analysis approach, we assess the impact of equivalent and redundant mutants generated by *DaMAT*.

We determined if a live mutant is nonequivalent by verifying, with the support of our industry partners, if there existed a test case that, after performing the mutation operation, would generate one observable output (e.g., log entry, state variable, or data sent in response to test inputs) that differs from the one generated by the original program. Otherwise a mutant was considered equivalent.

According to related work, two mutants should be considered redundant if they produce the same observable output for every possible input [51]. Since, with large CPSs, it is not possible to automatically determine if such condition holds (e.g., differential symbolic execution may not scale and is hardly applicable when components communicate through a network), we rely on manual inspection. To make such such analysis feasible, we first need to select a subset of mutant pairs that are likely to be redundant (e.g., mutants that produce the same output for every executed test case). However, the size of the CPSs under analysis prevents the collection of all the observable outputs produced by the system. We thus select as likely to be redundant all the pairs of killed mutants that (1) are exercised by the same test cases and (2) present the same failing assertions for every test case. We then manually inspect the test cases to determine if an additional assertion or a different test input might lead to different results for the two mutants. Similar to related work, we exclude live mutants from this analysis [43].

Results. All live mutants (i.e., 55 mutants for *ESAIL-ADCS*, 1 mutant for *ESAIL-GPS*, and 45 mutants for *LIBParam*) generate outputs that differ from the original CPS and, therefore, we did not detect any equivalent mutant. Though it needed to be confirmed, such result was expected since our methodology (Section 3.3), if correctly applied, suggests, for every data item, a set of mutation operators that, by construction, should not lead to mutated data that is equivalent to the original data. Live mutants can be killed by introducing oracles that (1) verify additional entries in the log files (39 instances for *ESAIL-ADCS*, 1 instance for *ESAIL-GPS*), (2) verify additional observable state variables (14 instances for *ESAIL-ADCS*, 45 instances for *LIBParam*), and (3) verify not only the presence of error messages but also their content (2 instances for *ESAIL-ADCS*).

We did not find redundant mutants either, which was expected since (1) mutations concerning different data items, by definition, are expected to lead to different outputs, (2) the set of operators applied to a same data item, if selected according to our methodology, cannot lead to mutated data that is redundant. All the pairs of likely redundant mutants were due to five situations: (1) the test case does not distinguish failures across data items (e.g., temperature values collected by different sensors), (2) the test case does not distinguish errors across different messages (e.g., in *ESAIL-ADCS*, the IfHK message reporting a broken sensor or the message sent by a sensor reporting malfunction), (3) the test case does not distinguish between errors in nominal and non-nominal data (e.g., it does not distinguish between VOR and FVOR), (4) the test case does not distinguish between upper and lower bounds (e.g., the mutants for VOR lead to the same assertion failures), and (5) the test case does not distinguish between different error codes (i.e, it simply verifies that an error code is generated). Addressing such shortcomings make test cases more useful for root cause analysis.

4.5 RQ3 - Feasibility

The feasibility of data-driven mutation analysis depends on the required manual effort, which includes defining fault model specifications and injecting probes into the SUT source code. Also, the overhead introduced at runtime by the execution of the mutation operations may introduce delays in real-time systems and consequently cause failures. Finally, feasibility also depends on the overall duration of the mutation analysis process.

To discuss manual effort, we measured, for each subject, (1) the number of rows in the fault model specifications, as they match

Table 7: Manual effort and execution time.

Subject	Configured Operators	Configured Operators / FM	LoC / FM	Execution time Original	*DaMAT*
ESAIL-ADCS	142	14.20	6.10		1,207.32 [h]
ESAIL-GPS	23	23.00	2.72	8.34 [h]	217.45 [h]
ESAIL-PDHU	29	9.66	4.33		69.75 [h]
LIBParam	80	13.33	7.64	0.02 [h]	0.27 [h]

the number of operators manually identified and configured by an engineer, and (2) the number of lines of code (LoC) added to the source code of our subjects. The latter includes invocations to function *mutate* (see Section 3.4) and additional utility code such as exit handlers used to clear the fault models loaded into memory. Since the number of added lines of code depends on the number of fault models per case study, we report the ratio of LoC per fault model.

To address the overhead, we measured the execution time taken by every passing test case when executed with the original software and with any of the mutants generated by the approach. We exclude failing test cases because they may bias the results (e.g., failing assertions may terminate a test case earlier, while test timeout failures are detected when a test case execution takes too long). To account for performance variations due to the varying load of our HPC, we executed every test case three times. For every test case, we then computed the overhead of every mutant as the difference between the average execution time obtained with the mutants and that with the original software. Since different subjects are characterized by different types of messages being exchanged, we discuss the distribution of such overhead among our subjects.

Last, to discuss the overall duration of the mutation analysis process, we report the average time taken to execute the test cases selected by the approach for every mutant, across three runs.

Results. The left part of Table 7 reports the measures related to manual effort. The number of operators configured per subject varies from 23 to 142, with an average between 9.66 and 23 operators per fault model. In our experiments, on average, it took between five and ten minutes to configure an operator. Given that the definition of test cases for safety-critical CPS components, such our case study subjects, takes days to complete, our industry partners found the required effort acceptable. The same considerations hold for the number of LoC per fault model, whose average across subjects varied between 2.72 and 7.64[2]

Excluding outliers (i.e., values above 90^{th} percentile), the maximum execution overhead for *ESAIL-ADCS*, *ESAIL-GPS*, *ESAIL-PDHU*, and *LIBParam* is 1.47%, 3.16%, 1.7%, and 7.59%, respectively. We did not observe any failure due to violated timing constraints. The larger overhead for *LIBParam* is due to the short execution time of its test cases (6 seconds, on average) and is not practically significant. The small overhead introduced by *DaMAT* on the three other subjects is acceptable and does not prevent its application to real-time CPSs.

[2]The exit handler includes one call for each fault model and thus subjects with less fault models show a lower average. For *LIBParam*, the larger number of lines of code is due to the need for recomputing a message checksum after the invocation of function *mutate*.

The right part of Table 7 shows the *DaMAT* analysis time. Although it is much larger than the execution times of test suites for the original SUTs, it is practically feasible. Indeed, in the worst case (i.e., *ESAIL-ADCS*), mutation analysis can be performed in 12 hours with 100 parallel computation nodes; in safety-critical contexts, where development entails large costs, buying computation time on the Cloud is affordable. Code-driven mutation analysis for systems with similar characteristics lasts considerably more [10, 48].

4.6 Threats to validity

Generalizability. We have selected industrial CPSs of diverse size, tested with different types of test suites. They are developed according to space safety standards and are thus representative of CPS software adhering to safety regulations. Also, ESAIL is larger than any other industrial system considered in the mutation analysis literature to date [6, 16, 17, 48]. *Internal.* To minimize implementation errors, we have extensively tested our toolset; we provide both the test cases and the *DaMAT* source code. *Construct.* The indicators selected for cost estimation (configured operators and LoC) are directly linked to the activities of the end-user and are thus appropriate. We leave empirical studies with human subjects to future work. To discuss overhead, we rely on test execution time, which may be affected by other factors than mutation overhead (e.g., the system behaves differently with mutated data). Dedicated benchmarks might be an alternative. *Conclusion.* To ensure reliability, for RQ1 and RQ2, we confirmed our findings with engineers.

5 CONCLUSION

Assessing the quality of test suites is necessary in the case of large, safety-critical systems, such as most CPSs, where software failures may lead to severe consequences including loss of human lives or environmental damages.

In this paper, we have introduced data-driven mutation analysis, a new paradigm to assess the effectiveness of a test suite in detecting interoperability faults. Interoperability faults are a major source of failures in CPSs but they are not targeted by existing mutation analysis approaches. Data-driven mutation analysis works by modifying (i.e., mutating) the data exchanged by software components during test execution. Mutations are performed by a set of mutation operators that are configured according to a fault model provided by software engineers, following a proposed methodology. Test suite assessment is based on three metrics: (1) fault model coverage, (2) mutation operation coverage, (3) and mutation score. These metrics enable engineers to determine specific weaknesses in test suites (e.g., oracles with low mutation score).

We have proposed a technique, *DaMAT*, that is applicable to a vast set of systems since it mutates the data exchanged through data buffers, by relying on a tabular fault model that can be inexpensively loaded into memory. We have defined a set of mutation operators that enable the simulation of data faults causing interoperability problems in CPSs. We have provided a methodology that supports the definition of a fault model based on the nature, representation type, and dependencies of the data to mutate.

We empirically evaluated *DaMAT* by applying it to test suites of commercial CPS components in the space domain that are currently deployed on orbit. Our results show that *DaMAT* can identify a large and diverse set of test suite shortcomings, entails limited modelling and execution costs, and is not affected by redundant and equivalent mutants.

6 DATA AVAILABILITY

The source code of our toolset, usage examples, and the data collected in our empirical evaluation are available [5]. Our case study software cannot be provided since it is proprietary.

REFERENCES

[1] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. 2015. A Survey on Testing for Cyber Physical System. In *Testing Software and Systems*, Khaled El-Fakih, Gerassimos Barlas, and Nina Yevtushenko (Eds.). Springer International Publishing, Cham, 194–207.

[2] Bernhard Aichernig, Harald Brandl, J Elisabeth, Willibald Krenn, Rupert Schlick, and Stefan Tiran. 2015. Model-based Mutation Testing for UML. *Icst 2015* (2015). www.momut.org

[3] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, USA.

[4] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. ACM, 402–411.

[5] Authors of this paper. 2021. Replicability package. TO APPEAR.

[6] Richard Baker and Ibrahim Habli. 2013. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering* 39, 6 (2013), 787–805. https://doi.org/10.1109/TSE.2012.56

[7] James H. Barton, Edward W. Czeck, Zary Z Segall, and Daniel P. Siewiorek. 1990. Fault injection experiments using FIAT. *IEEE Trans. Comput.* 39, 4 (1990), 575–582.

[8] Fevzi Belli, Christof J. Budnik, Axel Hollmann, Tugkan Tuglular, and W. Eric Wong. 2016. Model-based mutation testing—Approach and case studies. *Science of Computer Programming* 120 (2016), 25–48. https://doi.org/10.1016/j.scico.2016.01.003

[9] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 122–131.

[10] Oscar Eduardo Cornejo Olivares, Fabrizio Pastore, and Lionel Briand. 2021. Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3107680

[11] Franco Davoli, Charilaos Kourogiorgas, Mario Marchese, Athanasios Panagopoulos, and Fabio Patrone. 2019. Small satellites and CubeSats: Survey of structures, architectures, and protocols. *International Journal of Satellite Communications and Networking* 37, 4 (2019), 343–359. https://doi.org/10.1002/sat.1277 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1277

[12] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. 1996. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of Annual Symposium on Fault Tolerant Computing*. IEEE, 404–414.

[13] Marcio Eduardo Delamaro, Lin Deng, Vinicius Humberto Serapilha Durelli, Nan Li, and Jeff Offutt. 2014. Experimental evaluation of SDL and one-op mutation for C. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 203–212.

[14] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. 2001. Interface mutation: An approach for integration testing. *IEEE transactions on software engineering* 27, 3 (2001), 228–247.

[15] Marcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. 2014. Designing deletion mutation operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 11–20.

[16] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. 2018. Evaluation of Mutation Testing in a Nuclear Industry Case Study. *IEEE Transactions on Reliability* 67, 4 (2018), 1406–1419.

[17] Alex Denisov and Stanislav Pankevich. 2018. Mull it over: mutation testing based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 25–31.

[18] Xavier Devroey, Mike Papadakis, Axel Legay, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. ii (2016).

[19] Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, and Lionel Briand. 2015. Evolutionary Robustness Testing of Data Processing Systems Using Models and Data Mutation (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 126–137.

[20] Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. 2015. Generating complex and faulty test data through model-based mutation analysis. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[21] Anup K Ghosh, Matthew Schmid, and Viren Shah. 1998. Testing the robustness of Windows NT software. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No. 98TB100257)*. IEEE, 231–235.

[22] Omid Givehchi, Klaus Landsdorf, Pieter Simoens, and Armando Walter Colombo. 2017. Interoperability for Industrial Cyber-Physical Systems: An Approach for Legacy Systems. *IEEE Transactions on Industrial Informatics* 13, 6 (2017), 3370–3378. https://doi.org/10.1109/TII.2017.2740434

[23] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI ?08)*. Association for Computing Machinery, New York, NY, USA, 206?215. https://doi.org/10.1145/1375581.1375607

[24] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

[25] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227.

[26] Mark Grechanik and Gurudev Devanla. 2016. Mutation Integration Testing. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 353–364. https://doi.org/10.1109/QRS.2016.47

[27] Seungjae Han, Kang G Shin, and Harold A Rosenberg. 1995. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE, 204–213.

[28] Mark Harman, Yue Jia, and William B Langdon. 2010. A manifesto for higher order mutation testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 80–89.

[29] Nannan He, Philipp Rümmer, and Daniel Kroening. 2011. Test-case generation for embedded simulink via formal concept analysis. *Proceedings - Design Automation Conference* (2011), 224–229. https://doi.org/10.1145/2024724.2024777

[30] Y. Isasi, A Pinardell, A. Marquez, C. Molon-Noblot, A. Wagner, M. Gales, and M. Brada. 2019. The ESAIL Multipurpose Simulator. In *Onlin Proceedings of Simulation and EGSE for Space Programmes (SESP2019)*. https://atpi.eventsair.com/QuickEventWebsitePortal/sesp-2019/website/Agenda.

[31] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[32] Ying Jiang, Shan-Shan Hou, Jin-Hui Shan, Lu Zhang, and Bing Xie. 2005. Contract-based mutation for testing components. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 483–492. https://doi.org/10.1109/ICSM.2005.36

[33] Václav Jirkovský, Marek Obitko, and Vladimír Mařík. 2017. Understanding Data Heterogeneity in the Context of Cyber-Physical Systems Integration. *IEEE Transactions on Industrial Informatics* 13, 2 (2017), 660–667. https://doi.org/10.1109/TII.2016.2596101

[34] Siddhartha Kumar Khaitan and James D. McCalley. 2015. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal* 9, 2 (2015), 350–365. https://doi.org/10.1109/JSYST.2014.2322503

[35] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2017), 308–333.

[36] Pedro Reales Mateo, Macario Polo Usaola, and Jeff Offutt. 2010. Mutation at system and functional levels. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 110–119.

[37] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. 2019. Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior. *IEEE Transactions on Software Engineering* 45, 9 (Sep. 2019), 919–944. https://doi.org/10.1109/TSE.2018.2811489

[38] NASA. 1998. Mars Climate Orbiter, Spacecraft lost. https://solarsystem.nasa.gov/missions/mars-climate-orbiter/in-depth/.

[39] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. 2016. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 44.

[40] OASIS. 2007. OASIS Web Services Business Process Execution Language (WSBPEL) TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[41] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[42] Jeff Offutt, Paul Ammann, and Lisa Liu. 2006. Mutation testing implements grammar-based testing. *2nd Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), MUTATION'06* (2006), 12. https://doi.org/10.1109/MUTATION.2006.11

[43] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings*

of the 25th International Symposium on Software Testing and Analysis. ACM, 354–365.

[44] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[45] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 537–548.

[46] Peach Tech. [n.d.]. Peach Fuzzer. https://www.peach.tech

[47] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 543–553.

[48] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. 2017. An empirical study on the application of mutation testing for a safety-critical industrial software system. *Proceedings of the ACM Symposium on Applied Computing* Part F128005, Section 4 (2017), 1401–1408. https://doi.org/10.1145/3019612.3019830

[49] Hendrik Roehm, Jens Oehlerking, Matthias Woehrle, and Matthias Althoff. 2019. Model Conformance for Cyber-Physical Systems: A Survey. *ACM Trans. Cyber-Phys. Syst.* 3, 3, Article 30 (Aug. 2019), 26 pages. https://doi.org/10.1145/3306157

[50] Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators A. Jefferson Offutt Ammei Lee George Mason University. *ACM Transactions on software engineering methodology* 5, 2 (1996), 99–118.

[51] D. Shin, S. Yoo, and D. Bae. 2018. A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 914–931. https://doi.org/10.1109/TSE.2017.2732347

[52] Space company. 2020. ESAT1 - Hidden for double-blind review.

[53] Timothy K Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 1999. Stress-based and path-based fault injection. *IEEE Trans. Comput.* 48, 11 (1999), 1183–1201.

[54] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 92–102.