Evolutionary Robustness Testing of Data Processing Systems using Models and Data Mutation

Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, Lionel Briand
Interdisciplinary Centre for Security, Reliability and Trust (SnT Centre)
University of Luxembourg, Luxembourg
Email: {daniel.dinardo,fabrizio.pastore,andrea.arcuri,lionel.briand}@uni.lu

***Abstract*—System level testing of industrial data processing software poses several challenges. Input data can be very large, even in the order of gigabytes, and with complex constraints that define when an input is valid. Generating the right input data to stress the system for robustness properties (e.g. to test how faulty data is handled) is hence very complex, tedious and error prone when done manually. Unfortunately, this is the current practice in industry. In previous work, we defined a methodology to model the structure and the constraints of input data by using UML class diagrams and OCL constraints. Tests were automatically derived to cover predefined fault types in a fault model. In this paper, to obtain more effective system level test cases, we developed a novel search-based test generation tool. Experiments on a real-world, large industrial data processing system show that our automated approach can not only achieve better code coverage, but also accomplishes this using significantly smaller test suites.**

I. Introduction

Data processing software is an essential component of systems that aggregate and analyse real-world data, thereby enabling automated interaction between such systems and the real world. Examples are search engines that return stock quotes [1] or personal information collected from the web [2], web applications that show real-time airplane positions [3], devices such as specialised eyeglasses that capture images and provide contextual suggestions [4], and phones able to translate in real time the words of a road sign [5].

Robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [6]. Robustness testing of data processing software in the presence of invalid inputs is of particular importance because of the impact unexpected failures may have. For example, data faults in the transmission of an airplane position may crash airplane tracking applications, while malformed HTML pages may lead a web crawler to report stock market collapses and cause panic to end users [7].

Robustness testing of data processing software is often complicated by the complex structure of the input data. A well known example is an HTML page that contains many blocks, some of which are kept hidden or contain dynamic information. Similar complexity characterises other kinds of processing systems; for example, the data acquisition (DAQ) systems developed by our industrial partner SES to process satellite transmissions [8]. When performing robustness testing, software engineers need to handcraft complex data structures where valid and faulty values need to be inserted while taking care to preserve all the relationships among the data fields. Handcrafting huge amounts of complex data is particularly time consuming and error prone.

In [8] we introduced a technique to test the capability of data processing software to identify invalid test inputs that match a given fault model. The technique uses a set of generic mutation operators to generate test inputs by sampling and mutating field data. The technique ensures that each possible fault instance characterised by a fault model is covered by the generated test cases.

In this paper, we tackle the more general problem of generating minimal robustness test suites, with high fault revealing power, for data processing systems. When dealing with robustness testing, a single test generation criterion, for example the coverage of the given fault model implemented in [8], is not enough. Multiple factors must be considered; for example the presence of multiple data faults in the same input, the coverage of functional specifications in addition to fault models, and the generation of a minimal number of test cases. Satisfying multiple criteria when generating robustness test suites may easily lead to combinatorial explosion and specific techniques able to deal with scalability issues are required.

When addressing complex problems where there is a large space of candidate solutions, and one wants to choose a solution that maximises some chosen criteria, metaheuristics are a plausible solution [9]. Metaheuristic algorithms, such as evolutionary algorithms, identify optimal solutions for a problem by iteratively building candidate solutions and by testing them to identify the one that best achieves the objectives. At each iteration, new candidate solutions are built by means of a tweak operation that is applied on a copy of a candidate solution. The tweak operation allows the algorithm to explore the search space looking for an optimal solution.

Testing techniques based on metaheuristic search focus mostly on unit testing [10], while techniques that tackle testing at the system level either address the problem of testing non-functional properties such as execution time [11], or deal with the problem of testing systems where the costs of testing do not depend on complex input data structures, such as in the case of embedded systems working with input signals [12].

In this paper, we propose a model-based evolutionary algorithm that relies upon a data model and a set of data mutation operators to build system test suites for data processing systems that optimises multiple objectives. The evolutionary algorithm uses data sampling and data mutation operators to generate new test inputs, and relies upon four different model-based and code-based fitness functions to evaluate how well each test input contributes to a proper robustness test suite. Model-based fitness functions exploit the data model to generate test cases that cover important aspects of the behaviour of the system by ensuring the coverage of all the

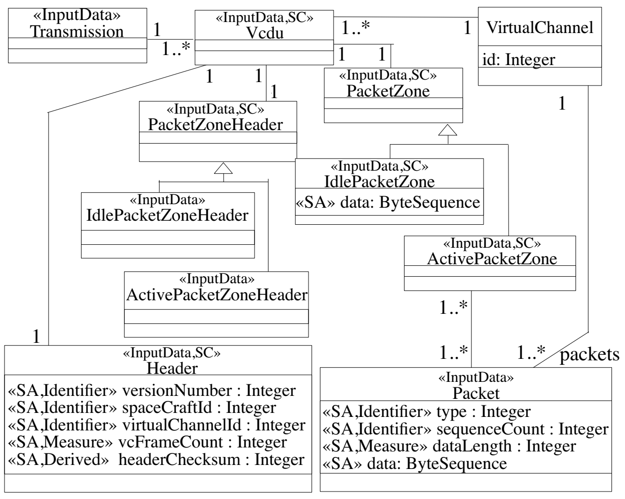

Fig. 1. Simplified input data model for the SES running example. SA, StreamAttributes; SC, StreamClass.

different types of test inputs processed by the system, the presence of different types of data faults, and the possible violations of the constraints among test inputs. The code-based fitness function has the goal of achieving the maximum structural coverage of the system under test.

This paper contributes to the state of the art by:

- proposing an evolutionary algorithm to automate robustness testing of data processing systems;
- defining four fitness functions (model-based and code-based) that enable the effective generation of robustness test cases by means of evolutionary algorithms;
- performing an extensive study of the effect of fitness functions and configuration parameters on the effectiveness of the approach using an industrial data processing system as case study.

The paper proceeds as follows. Section II provides background information on the data modelling and mutation operators used by the algorithm. Section III presents a list of challenges that need to be addressed when building a search algorithm for robustness testing. Section IV provides an overview of the evolutionary algorithm that we designed to generate robustness tests. Section V describes how data mutation is adopted to generate new test inputs during the search. Section VI presents the heuristic functions used to evaluate candidate solutions. Section VII describes the seeding strategy integrated into the algorithm. Section VIII details how we automate the execution and validation of the generated test suites. Section IX presents the empirical results obtained, whereas Section X discusses related work. Finally, Section XI concludes the paper.

II. Background on Data Modelling

The evolutionary algorithm presented in this paper uses data models to automatically generate test inputs. Such data models are designed according to the data modelling methodology introduced in previous work [8], [13]. This section provides an overview of the methodology.

The methodology presented in [8] uses UML class diagrams to capture the structure of inputs and outputs, relies upon Object Constraint Language (OCL) [14] expressions to define relationships between the inputs and outputs, and uses UML

```
context Vcdu inv:
let
    frameCount : Integer = self.header.vcFrameCount,
    vdcuIndex : Integer = self.virtualChannel.vcdu→indexOf( self ),
    previous : Vcdu = self.virtualChannel.vcdu→at( vcduIndex - 1 ),
    previousFrameCount : Integer = previous.header.vcFrameCount
in
    if previousFrameCount < 16777215
        then frameCount <> previousFrameCount + 1
    else previousFrameCount = 16777215 and frameCount <> 0 endif
implies
    VcduEvents.allInstances()
            →exists(e | e.eventType = COUNTER_JUMP)
```

Fig. 2. Input/output constraint for the *COUNTER_JUMP* error event.

stereotypes to capture a fault model driving the generation of test cases.

To briefly present the methodology, we show how it can be applied to model the transmission data processed by SES-DAQ, a DAQ system developed by SES (see [15] for the complete specifications of the transmission data). SES-DAQ processes bytestreams of transmitted satellite data. Developing such a model requires time and engineering effort but the cost of modelling was considered acceptable by SES engineers [13].

Figure 1 shows how we model SES-DAQ input data according to our methodology. The model captures the structure of a transmission: we use UML classes to represent elements that contain multiple fields, while we use UML attributes to model elements that cannot be further decomposed. For example, it shows that each transmission consists of a sequence of *Virtual Channel Data Units (VCDUs)*. Each *VCDU* begins with a *Header*, followed by a *PacketZoneHeader* and a *PacketZone* that may contain a sequence of *Packets* (if the packet zone is active). The *VCDUs* in a transmission may belong to different virtual channels.

Associations are used to represent containment relationships. In Figure 1 the classes that model the VCDU and its Header are connected by an association. We use generalisations to indicate when a data field can have multiple different definitions. For example, in the case of SES-DAQ the packet zones can be active (i.e. they transmit data) or idle (i.e. they do not transmit anything). The data model of SES-DAQ reflects this characteristic with generalisations for PacketZoneHeader and PacketZone, both of which have idle and active subtypes.

We use class attributes to represent the transmitted binary information (e.g. checksums, frame counters, or data). For example, attribute *sequenceCount* of class *Packet* is used to store information about the packet order.

Constraints between inputs and outputs are represented using OCL. These constraints are used as an oracle to validate the execution of automatically generated test cases: a violated constraint indicates that the system under test does not produce the expected output. In the case of SES-DAQ, we use OCL constraints to model the error messages expected in the presence of specific faults in the input data. For example, Figure 2 shows an OCL constraint that states that the frame count of a VCDU should be greater by one than the frame count of the previous VCDU on the same virtual channel. Otherwise, an error event *COUNTER_JUMP* should exist in the system output log file.

Stereotypes are used to capture the fault model of the system, and are used to identify the fields that can be mutated to generate new inputs. The stereotype *InputData* is used by the software engineer to tag the classes that model input data, to contrast them with output data classes, which do not need to be mutated. The two stereotypes *Identifier* and *Measure* are used to indicate the mutation operators to apply to class attributes. The stereotype *Derived* is used to tag class attributes that need to be derived from other attributes after every mutation, in order to prevent trivial inconsistencies (e.g. it is used to update the checksum field when other fields are mutated).

The stereotypes *StreamClass* and *StreamAttributes* are used to automate the loading of data from bytestreams but are out of the scope of this paper (see [8] for details).

III. Challenges for the Search-based generation of Robustness System Tests

Search algorithms are useful when addressing complex problems where there is a large space of candidate solutions, and one wants to choose one that optimises some chosen criteria. A typical example in software engineering is test data generation, in which a tester might want to find test data that maximises code coverage or triggers new failures.

There are many different kinds of search algorithms, including genetic algorithms, which have been widely applied in many fields. Not all search algorithms perform well on all types of problems. Each problem can have special characteristics that are better exploited by some search algorithms, whereas others might struggle.

In this paper, we identify four main challenges that need to be addressed when designing an algorithm for the automatic generation of a system test suite: (1) configuring the algorithm to properly find a tradeoff between *exploration* and *exploitation* of the search landscape, (2) building a *tweak operation*, (3) defining effective *fitness functions* for the problem under investigation, and (4) integrating effective *seeding techniques* (i.e. techniques that speed up search by exploiting knowledge about the input domain).

One of the main discerning characteristics among search algorithms is the tradeoff they have between *exploration* and *exploitation* of the search landscape. On one hand, some algorithms (e.g. hill climbing) put more emphasis on the exploitation of the search landscape. This means that, given an evaluated solution, they will only look at "close" solutions to see if any small change could improve the chosen criterion. On the other hand, other algorithms put more emphasis on the exploration of the search space. Typical examples are population-based algorithms, in which a diverse set of solutions is maintained to consider different areas of the search landscape at the same time, in case one area turns out to feature more fitting solutions than the others.

In the context of robustness testing, exploration is very important since it enables the construction of test suites with a high diversity of test cases. In the case of SES-DAQ, for example, one wants to generate test inputs that include *IdlePacketZones* and test inputs that include *ActivePacketZones* at the same time. However, exploitation leads to covering invalid inputs that include a specific set of data faults. For example, SES-DAQ might be able to properly process an invalid input containing a packet that breaks the constraint in Figure 2, but it may crash in the presence of both a broken constraint and a duplicated packet. To satisfy this case, the algorithm must be able to exploit the search space by both breaking the constraint of Figure 2 and by duplicating a packet.

Tweak operations play an important role in guiding the search algorithm towards the exploitation of the search landscape; they modify existing solutions to generate new candidate solutions. When defining tweak operations, the characteristics of the domain should be carefully considered. For example, when a solution is represented in a binary format, a tweak operation could just flip one or more bits. However, flipping bits on a complex data transmission file would likely result in meaningless or trivially wrong input data. Therefore, one would need to exploit the information in the data models to automatically derive better tweak operations, for example by applying model-based data mutation.

In the context of robustness system testing, one still wants to generate faulty input data, but in a nontrivial way (e.g. by including multiple faults in a same test input). However, there is a limit on the number of faults that can be included in a same input. Although having multiple mutations that affect a same test input might help stress the robustness of the system, a number of faults that is too high might lead to inputs that are trivially recognised and discarded by the system under test.

Another very important aspect for the success of a search algorithm is the definition of a *fitness function*, used to evaluate how close a solution is to optimising a chosen criterion. Such a function is problem dependent, and effort must be made to design a proper one. Ideally, one would like to exploit as much domain information as possible, but this might lead to computationally expensive fitness functions. The more time consuming the fitness function, the fewer solutions can be evaluated by a search algorithm in the same amount of time. In the case of model-based testing, this tradeoff can be very critical, as fitness functions calculated on models can be much quicker to compute than ones calculated from test case executions (e.g. using structural coverage).

Finally, in the presence of complex search spaces, the quick identification of a proper solution is often aided by the adoption of techniques that exploit knowledge about the search space to build solutions that improve the effectiveness of the search algorithm; these are known as *seeding techniques*. Different seeding techniques have been adopted in the context of software testing; for instance, a well known approach used by techniques that generate inputs for unit testing is the reuse of constant values taken from the source code of the software under test [16]. However, smart seeding at the system level is not as simple as that, and poses new challenges.

IV. An Evolutionary Algorithm for Robustness Testing

To address the problem of generating test cases to stress the robustness of software at the system level, we propose a novel evolutionary algorithm based on an archive. Figure 3 shows the algorithm.

In the context of this paper, a solution to the search problem consists of a test suite that effectively tests the capability of the software to handle invalid data. A test suite is a collection of test inputs (i.e. data files conforming to a data model) to be processed by the software under test. During the search, test inputs are generated, and those are then aggregated to form a final test suite to give as output to the user. In our context, the solution to the search problem (i.e. the test suite) cannot be represented using a fixed size data structure because the size of the test suite (i.e. the number and size of the test cases it contains) cannot be known a priori.

We did not directly use a traditional search algorithm (e.g. a genetic algorithm or a hill climbing algorithm) due

```
Require: $fD$, the field data used to generate test inputs
Require: $dM$, the data model used to drive tweaking
Require: $budget$, the maximum proportion of the search space that needs to be visited
Require: $p_{field}$, probability of sampling a new individual from the field data
Require: $p_{mutation}$, probability of mutating an individual just after sampling it
Require: $p_{seeding}$, probability of using seeding to sample from the field data
Require: $minSize$, the minimum size of a test input
Require: $maxSize$, the maximum size of a test input
Require: $maxMutations$, the maximum number of mutations for a same test input
Ensure:  $archive$, the archive containing the minimised robustness test suite
 1: $total = 0$
 2: while $total < budget$ do
 3:     if ( $(archive.individualAvailable(maxMutations) = false)$
 4:         OR $(random() < p_{field})$ ) then
 5:         $ind = sampleNew(fD, dM, minSize, maxSize, p_{seeding})$
 6:         if $random() < p_{mutation}$ then
 7:             $mutate(ind)$
 8:         end if
 9:     else
10:         $ind = archive.sampleACopy()$
11:         $mutate(dM, ind)$
12:     end if
13:     if ( $improving(ind, archive)$ then
14:         $archive.add(ind)$
15:         for $prev\ in\ archive$ do
16:             if $subsume(ind, prev)$ then
17:                 $archive.remove(prev)$
18:             end if
19:         end for
20:     end if
21:     $total = total + ind.size$
22: end while
```

Fig. 3. An Evolutionary Algorithm for Robustness Testing

to the special characteristics of the addressed problem. For example, in system level testing, each test case execution can be computationally very expensive. So, a traditional genetic algorithm that works on a population would likely be too computationally expensive to use. Furthermore, special care would be needed to design a crossover operator that generates valid offspring. On the other hand, hill climbing algorithms put emphasis on the exploitation of the search landscape; this is achieved by using a tweak operation that iteratively improves a single solution. In our case, it is hard to envision a single tweak operation that works at the test suite level and allows for the building of a minimised test suite that contains test inputs with high diversity.

Our customised evolutionary algorithm is based on the use of an archive of test cases, initially empty. Archives have been used in prior work related to multi-objective search algorithms (e.g. [17], [18]). The archive plays the important role of guiding the algorithm towards the exploitation of the search landscape like hill climbing, while maintaining at the same time some characteristics of population-based algorithms. Like hill climbing, the algorithm improves only a single test input at each iteration, but uses the archive to keep a collection of the best test inputs found so far (i.e. the test suite). Furthermore, our algorithm keeps solutions in the archive that are different from each other to maximise exploration, like population algorithms. The size of the archive can vary during the search and the test suite is minimised by keeping in the archive only the best individuals that contribute to overall fitness of the whole test suite (i.e. the archive). Finally, the individuals in the archive are tweaked one at a time, thus exploiting the search landscape and creating new individuals that improve the overall fitness of the test suite.

Our novel algorithm addresses all the challenges presented in Section III. The tradeoff between exploration and exploitation is controlled by configuration parameters that regulate: (1) the probability of tweaking an individual from the archive versus generating a completely new individual (parameter p_{field} in Figure 3), (2) the probability of working with correct test inputs versus the use of test inputs that contain at least one fault (parameter $p_{mutation}$ in Figure 3), and (3) the maximum number of data faults a test input may contain (parameter $maxMutations$ in Figure 3). Tweaking operations are implemented by means of mutation operators described in [8], while specific fitness functions have been developed and are described in Section VI. The probability of seeding is controlled by the parameter $p_{seeding}$ (further details are given in Section VII).

At the first iteration, the archive is empty, and a new random individual needs to be sampled. Generating new input data completely at random would result almost certainly in trivially wrong data. An alternative is to sample according to some specific rules, if those can be defined for the addressed problem domain (e.g. a grammar in the testing of parsers). In our case, we used a different approach that relies on the sampling of field data. In the case of industrial data processing systems, we can have access to very large amounts of existing valid field data. If not already available, a large field data pool can be constructed, and then used by the evolutionary algorithm to sample from.

New individuals are sampled from the available field data by means of the function *sampleNew*, which randomly selects and returns a chunk of the available field data (Line 5). The function *sampleNew* receives as input two integer values, *minSize* and *maxSize* that indicate the minimum and maximum size of the data chunk to be sampled. Since in general an input for a data processing system does not have a size that is fixed a priori, we leave it to the software engineers to decide the range of the input size according to their domain knowledge; for example, in the case of SES we choose the values 1 and 500 for the minimum and maximum values, respectively (where these values represent the number of VCDUs).

The evolutionary algorithm incrementally builds a test suite by keeping only those individuals in the archive that contribute to improving the overall combined fitness of all the currently stored test cases, which will form the final test suite. The algorithm generates test inputs by applying the technique presented in [8], that is by sampling chunks of data from the field data and by mutating these chunks to generate possibly faulty inputs. Unlike [8], in this paper we consider the possibility of applying multiple mutations to a same test input. Each test input is thus represented in terms of the offset from the beginning of the original field data file, the length of the sample, and a list of the mutations that have been applied to the sample.

The algorithm keeps exploring the search space until a given stopping condition is reached (Line 2). Since our algorithm focuses on the generation of test inputs for data processing systems, we express the stopping condition in terms of the amount of data processed to generate test cases, that is the sum of the size of all the test inputs generated during the search. At each iteration, the algorithm increments the counter of the data processed (Line 21). In the specific case of SES, we measure the size of a single test input in terms of the number of VCDUs that it contains, but different measurement units (meaningful for a given domain) may be used for different systems.

At each iteration, the algorithm works by tweaking an individual (i.e. a test input). Each individual is created by sampling either the field data or the archive; this choice is driven by a probability value, the parameter p_{field} in Figure 3, which indicates the probability of sampling a new individual from the field data (see Lines 3 and 4).

If no individuals are available in the archive, the algorithm samples the field data (see the condition *archive.individualAvailable() = false* in Line 3). This happens in two situations, when the archive is empty (i.e. on the first search iteration) or when all the individuals in the archive have already been mutated a maximum numbers of times. Software engineers can specify the maximum number of mutations that can be applied to the same test input. This is done to avoid trivially invalid inputs. Although in principle a test input can be mutated an infinite number of times, the presence of too many mutations (i.e. data faults) on the same test input, might transform the test input into a trivially invalid input that is easily detected by the data processing system and does not help to extensively test its robustness.

Lines 6 to 8 show that the parameter $p_{mutation}$ regulates the probability of mutating an individual just after creating it, that is the probability of working with individuals that contain at least one data mutation. Lines 10 and 11 show that every time the algorithm samples a copy of an individual from the archive it applies a mutation to it. This is done to create a copy that differs from the original one. This tweaking operation is further described in Section V.

Lines 13 and 14 show that an individual is added to the archive only if it improves the overall fitness of the archive. Similarly, the algorithm removes any individual already present in the archive that is subsumed by the last one added (see Lines 15 to 17). Individuals that are subsumed by new ones can be safely removed from the archive because they do not contribute to the overall fitness of the archive. Section VI describes the assessment procedure adopted to measure how individuals contribute to the fitness of the archive.

V. Tweaking by means of Data Mutation

The search algorithm tweaks individuals by applying the data mutation operators described in [8]. There are six mutation operators that can be applied to an individual: Class Instance Duplication, Class Instance Removal, Class Instances Swapping, Attribute Replacement with Random, Attribute Replacement using Boundary Condition, and Attribute Bit Flipping. To apply a mutation operator to an individual, the algorithm loads into memory the data chunk corresponding to the test input as an instance of the data model.

According to [8], each mutation operator can be applied only to a specific set of targets. This is done to avoid making mutations that will result in the generation of trivial data faults and to ensure conformance with a domain-specific fault model. Software engineers specify the targets of each mutation operator by using appropriate stereotypes in the UML class diagram (data model). These stereotypes indicate the elements that can be mutated and the operators that can be applied to them. To mutate an individual, the algorithm randomly picks a mutation operator, identifies a possible target for the operator on the current individual, and applies the operator on the target. For example, during the generation of test inputs for SES-DAQ, the algorithm may randomly choose the operator *Attribute Replacement with Random*. It then selects one of the attributes that can be mutated according to that operator, for example the attribute *sequenceCount* of class *Packet*. Finally, it identifies a specific instance to mutate, which means that it changes the value of the attribute *sequenceCount* of one of the *Packet* instances contained in the current test input.

In case a selected operator cannot be applied on a given individual, another operator is randomly selected; this can happen, for example, if the algorithm selects the attribute *sequenceCount* for replacement with random, but the current test input contains only *IdlePacketZones* (which according to Figure 1 does not contain any Packet instances).

VI. Assessment Procedure

We identify four objectives that should be fulfilled to effectively stress the robustness of the software:

- O1: include input data that covers all the classes of the data-model;
- O2: include data faults such that all the possible faults of the fault model have been covered;
- O3: cover all the clauses of the input/output constraints;
- O4: maximise code coverage.

Each of the four objectives captures how well the test inputs cover some specific targets, respectively, the classes of the data-model, the faults of the fault model, the clauses of the input/output constraints and the code instructions. Each objective defines a set of *targets* (e.g. instructions in code coverage) that the algorithm aims to cover. A given objective is fully achieved by a test suite if each of its targets is covered by at least one test input of the test suite. A portion of the targets for SES-DAQ along with their coverage for three test inputs are shown in Table I (covered targets are marked with an X). We describe below each objective, and how fitness improvements and subsumption can be defined in terms of these objectives.

Objective O1 ensures that the test suite includes test inputs that cover all the classes of the data model. Each class of the data model is univocally represented by an objective target. A test input covers a class if it contains at least one instance of the class. Table I shows that input *I3* covers, among others, classes *ActivePacketZone* and *IdlePacketZone*; input *I1* covers only class *ActivePacketZone* (there are no idle packets in *I1*).

Objective O2 ensures that an instance of each class and attribute has been mutated at least once by each mutation operator that can be applied to it (to generate test inputs covering all the faults of the fault model). Our algorithm generates faulty data (i.e. new test inputs) by applying mutation operators on instantiated field data objects. Since the attributes and classes of the data model can be mutated in different ways by applying different mutation operators, for each test input we keep track of which mutation operator has been applied to a specific class/attribute. Table I, for example, shows that input I1 contains at least one instance of a *VCDU* whose *vcFrameCount* has been mutated with the operator *AttributeReplacementWithRandom*, while input *I3* contains both a *VCDU* with a deleted packet (operator *ClassInstanceRemoval* is marked as being applied on an instance of class *Packet*) and a *VCDU* whose *versionNumber* has been replaced with a random value (see operator *AttributeReplacementWithRandom* marked for the attribute *versionNumber*).

Objective O3 ensures that every clause of the input/output constraints has been exercised. Input/output constraints are expressed in the form of implications. The left hand side of the implication captures the characteristics of the input under which a given output is expected. The right hand side captures the characteristics of the expected output (see Figure 2).

To measure how well the test suite stresses the conditions under which a given output is generated, it is enough to focus on the clauses contained on the left side of the implication (i.e. clauses defined over the characteristics of the test input). For each clause, we aim to have at least one test input that causes the clause to be true and another that causes the clause to be false. Each clause is thus associated to two different targets for objective O3 that trace whether the clause is true/false at least once in the input. Table I shows some targets derived for the constraint in Figure 2. Table I shows that input I1 has at least one VCDU whose *frameCount* does not correspond to the *previousFrameCounter* plus one (this is the effect of the mutation operator *AttributeReplacementWithRandom* applied to the attribute *vcFrameCount*). The same clause can be both true and false within the same test input. This is the case of input I1 that, in addition to a VCDU with the invalid *frameCount*, also includes VCDUs with a valid *frameCount* (i.e. a *frameCount* equal to *previousFrameCount* plus one).

It is noteworthy that, by focusing on the input clauses, we can measure objective O3 without the need to execute the system under test (i.e. without generating an output for a given test input). This makes the search algorithm scale even when the execution of the test cases is particularly time consuming.

Objective O4 aims to maximise the structural coverage of the source code. This is one of the means adopted by software engineers to ensure that all the implemented features have been tested at least once. Each instruction in the system is an objective target. In our implementation, we measure coverage using JaCoCo, a toolkit for measuring Java code coverage [19].

The main limitation of measuring the structural coverage of system test cases is that it requires the execution of the system under test. This may slow down the overall search process considerably and prevent the generation of results in practical time. Furthermore, in certain contexts, for example systems deployed on dedicated hardware, structural coverage might not be easily calculated in practice. For this reason, the empirical study presented in Section IX aims also to determine to which extent objective O4 is subsumed by other objectives.

Our algorithm works with objectives that are not conflicting and aims to maximise the coverage of all targets. Therefore, the algorithm does not rely upon the computation of Pareto fronts, a solution adopted by others (e.g. [20]).

Our algorithm adds to the archive only test inputs that improve the overall fitness (i.e. a test input must cover at least one target not covered by the other inputs in the archive). Furthermore, the algorithm removes from the archive any test inputs subsumed by new test inputs. A test input $i^{'}$ subsumes an input $i^{''}$ if, and only if, $i^{'}$ covers all the targets covered by $i^{''}$, and either $i^{'}$ covers at least one target not covered by $i^{''}$ or the size of $i^{'}$ is smaller. For example, given an archive that contains inputs *I1* and *I2*, our algorithm creates input *I3* by tweaking a copy of input *I2* (i.e. by deleting a class instance). *I3* is added to the archive because it covers target *Packet::ClassInstanceRemoval* (not covered by *I1* and *I2*). Given that *I3* subsumes *I2* the algorithm will then remove *I2* from the archive thus minimising its size.

VII. Input Seeding

To further improve search results, we developed a novel model-driven seeding strategy that guides the search towards the identification of a diverse and complex set of test inputs.

To stress diversity in the data, the algorithm aims to generate test inputs that cover all the available data types (see objective O1 described in Section VI). Given that some of these data types may occur rarely in the field data, it might be highly improbable to cover these types by means of random sampling. To guarantee the coverage of all the data types, it is enough to know the locations of the different data types. For example, within a field data sample of SES-DAQ, we may have idle packets only in a very small number of the VCDUs of the bytestream. Having the location information makes it easier for the search algorithm to load multiple data chunks containing idle packets; otherwise, the chance of loading idle packets is low when only resorting to random sampling.

To stress complexity our algorithm looks for test inputs that contain instances of two or more subclasses belonging to the same generalisation. In the case of SES-DAQ, this corresponds to the case of an input containing a transition between two alternate data types (e.g. from idle to active packet zone). These complex inputs are interesting for robustness testing because one can assume that handling heterogeneous data zones might be more prone to processing failures than homogeneous ones.

Our seeding strategy works by first processing the field data to build a *seeding pool* that contains data chunks that are useful to stress both diversity and complexity. To maximise diversity, we identify, for each class *S* that is a subclass of a generalisation: (1) data chunks that contain *at least* one instance of the subclass *S*, (2) data chunks that contain *only* instances of the subclass *S*, i.e. data chunks that contain instances of *S* but that do not contain instances of other classes belonging to the same hierarchy of *S*. To maximise complexity, we identify, for each pair of classes that belong to a generalisation, data chunks that contain at least one instance of each class in the pair. In the case of SES-DAQ this ensures that we test scenarios where two different data types are processed

TABLE I. Assessment of three inputs of SES-DAQ

	Objective Targets	Test Inputs		
		I1	I2	I3
Objective O1	Transmission	X	X	X
	Sync	X	X	X
	Header	X	X	X
	IdlePacketZoneHeader		X	X
	ActivePacketZoneHeader	X	X	X
	IdlePacketZone		X	X
	ActivePacketZone	X	X	X
	Packet	X	X	X
	...			
Objective O2	Header.versionNumber::AttributeReplacementWithRandom		X	X
	Header.vcFrameCount::AttributeReplacementWithRandom	X		
	Packet::ClassInstanceRemoval			X
	Packet::ClassInstanceDuplication			
	Packet::ClassInstancesSwapping			
	...			
Objective O3	True : previousFrameCount < 16777215	X	X	X
	True : frameCount <> previousFrameCount + 1	X		
	True : previousFrameCount = 16777215			
	True : frameCount <> 0	X	X	X
	...			
	False : previousFrameCount < 16777215			
	False : frameCount <> previousFrameCount + 1	X	X	X
	False : previousFrameCount = 16777215	X	X	X
	False : frameCount <> 0			
	...			
O4	SesDaq.java:Line 10	X	X	X
	SesDaq.java:Line 11	X		
	...			

TABLE II. List of the characteristics of the input data used to drive seeding for SES-DAQ

Only IdlePacketZoneHeader instances are included
At least one IdlePacketZoneHeader instance is included
Only ActivePacketZoneHeader instances are included
At least one ActivePacketZoneHeader instance is included
Only IdlePacketZone instances are included
At least one IdlePacketZone instance is included
Only ActivePacketZone instances are included
At least one ActivePacketZone instance is included
Both IdlePacketZone and ActivePacketZone are included
Both IdlePacketZoneHeader and ActivePacketZoneHeader are included

in sequence (e.g. idle and active packets). Table II shows a list with the characteristics of the data chunks we identify for the SES-DAQ data model in Figure 1.

The *seeding pool* can be used when a new individual is sampled. When seeding is enabled, the algorithm selects one of the chunks in the seeding pool. In the case of SES-DAQ this is done by first selecting one on the ten characteristics listed in Table II, and then by loading a data chunk that presents such characteristics. Software engineers can tune the use of seeding by means of a parameter for the search algorithm that indicates the probability of applying seeding when sampling a data chunk from the field data ($p_{seeding}$ in Figure 3).

Higher values of $p_{seeding}$ guarantee that all the characteristics are covered, at the expense of a free exploration of the search space (which may lead to the sampling of complex test inputs not identified by predefined seeding characteristics).

VIII. Testing Automation

The evolutionary algorithm generates a minimised robustness test suite that is kept in an archive. The test suite consists of a set of test inputs that can be executed against the DAQ system under test. The oracle relies on the model-based automated validation technique presented in [13] and [8].

After the execution of a test case, the oracle simply loads the test input and the test output as an instance of the data model, and checks if the OCL constraints of the data model are satisfied[1]. Unsatisfied constraints indicate the presence of a failure (i.e. unexpected or missing output) and are reported to the software engineers. Similarly, crashing executions are reported.

IX. Empirical Evaluation

We performed an empirical evaluation in order to respond to the following research questions:

- **RQ1:** How does the search algorithm compare with random and state-of-the-art approaches?
- **RQ2:** How does fitness based on code coverage affect performance?
- **RQ3:** How does smart seeding affect performance?
- **RQ4:** What are the configuration parameters that affect performance?
- **RQ5:** What configuration should be used in practice?

A. Subject of the Study

As subject of our study we considered the industrial SES-DAQ system. SES-DAQ is a good example of a data processing system dealing with complex input and output data, written in Java (having 32170 bytecode instructions). The data model of SES-DAQ includes 82 classes with 322 attributes and 56 associations. As an input for our approach, we considered a large transmission file containing field data provided by SES, the same adopted for the empirical evaluation in [8]. The size of the transmission file is about 2 gigabytes, containing 1 million VCDUs belonging to four different virtual channels.

[1]This is done by using the MDT/UML2 library, http://eclipse.org/modeling/

B. Experimental Settings

To answer our research questions, we carried out a series of experiments. Since our search algorithm depends on several parameters, we evaluated several possible configurations.

The *minSize* and *maxSize* parameters (i.e. the minimum and the maximum size of a test input, measured in VCDUs) were fixed to 1 and 500, respectively. We used three different values for p_{field}: 0.3, 0.5, and 0.8. For $p_{mutation}$, we considered the values: 0.0, 0.5, and 1.0. For *maxMutations*, we used: 1, 10, and 100. For $p_{seeding}$, we used two values, 0.0 and 0.5, which means that in one case the seeding strategy was not used, while in the other case the seeding was applied with a 50% probability every time a new input was sampled.

In order to give the algorithm some degree of freedom when exploring the search space, we do not consider the case in which inputs are selected exclusively according to the smart seeding strategy. Finally, we considered cases with and without the code coverage fitness function. This led to $3{\times}3{\times}3{\times}2{\times}2 = 108$ different configurations.

The search budget (in the case of SES-DAQ, the number of VCDUs inspected when building new inputs during search) might vary from project to project. For this reason, we evaluated each of the 108 configurations of the algorithm on five different search budgets from $50,000$ to $250,000$ VCDUs (in steps of $50k$). $250,000$ VCDUs correspond to one fourth of the transmission file used for building test inputs. This led to $108 \times 5 = 540$ different configurations of the search algorithm.

Because search algorithms have a random component, to take into account the effects of such randomness on the final results, each of the experiments was repeated five times with a different random seed. This led to a total of $540 \times 5 = 2,700$ runs of the algorithm. Because each run could take between ten and thirty-five hours, we used a large cluster of computers to run these experiments [21].

C. Cost and Effectiveness Metrics

We want to assess and compare cost-effectiveness among automatically generated test suites. Code coverage is used as a measure of test effectiveness as it helps assess how complete the test suite is from a structural and functional standpoint. We measure the code coverage in terms of the number of bytecode instructions covered by the test suite by using JaCoCo. Maximising code coverage within time constraints is often an objective among system testers. Even small improvements in coverage can help exercise important corner cases. For example, in our case study, additionally covered instructions often turned out to be critical blocks of code including exception handling and critical scenarios. We also compare test suites with respect to their size because, given two test suites having identical coverage, one would prefer the smaller one, entailing a lower testing and debugging cost. The size of a test suite is measured in terms of the number of test inputs in the test suite. Smaller test suites are of practical importance as, in many systems, system testing must be performed on actual deployment hardware or a dedicated, realistic testing platform, which requires some degree of tuning

TABLE III. Comparison between the best search algorithm configuration and random search. (Seeding disabled. Code coverage fitness enabled.)

Budget	Configuration	Coverage (Avg/Min/Max)	# Tests (Avg/Min/Max)
50k	Best: r=0.5,m=1,n=100	**23424.4** / 23407 / 23448	**28.4** / 19 / 32
	BO: r=0.5,m=1,n=100	**23424.4** / 23407 / 23448	**28.4** / 19 / 32
	Rand: r=1,m=1,n=1	23386.8 / 23341 / 23424	43.2 / 38 / 46
100k	Best: r=0.5,m=1,n=100	**23487.8** / 23461 / 23577	**31.6** / 25 / 35
	BO: r=0.5,m=1,n=100	**23487.8** / 23461 / 23577	**31.6** / 25 / 35
	Rand: r=1,m=1,n=1	23436.8 / 23428 / 23458	52.0 / 50 / 57
150k	Best: r=0.5,m=1,n=100	**23502.0** / 23471 / 23577	**34.0** / 30 / 38
	BO: r=0.5,m=1,n=100	**23502.0** / 23471 / 23577	**34.0** / 30 / 38
	Rand: r=1,m=1,n=1	23453.4 / 23438 / 23480	57.8 / 55 / 64
200k	Best: r=0.5,m=0.5,n=100	**23519.6** / 23464 / 23618	**34.6** / 28 / 38
	BO: r=0.5,m=1,n=100	23513.4 / 23476 / 23579	36.0 / 31 / 41
	Rand: r=1,m=1,n=1	23465.8 / 23449 / 23490	60.2 / 57 / 66
250k	Best: r=0.5,m=1,n=10	**23538.6** / 23463 / 23631	38.4 / 31 / 43
	BO: r=0.5,m=1,n=100	23515.2 / 23480 / 23579	**36.4** / 33 / 40
	Rand: r=1,m=1,n=1	23482.6 / 23452 / 23499	62.4 / 60 / 69

or simulation to run test cases. Access time to such platforms can also be limited.

D. *RQ1: How does the search algorithm compare with random and state-of-the-art approaches?*

The effectiveness of a search-based algorithm highly depends on the nature of the problem to solve. For example, if the solution space of the problem is flat, that is if the fitness function does not provide any gradient, then a search-based algorithm might perform even worse than a random approach.

RQ1 aims to evaluate the usefulness of the search-based algorithm proposed in this paper by comparing it with a random algorithm and with a simple model-based algorithm.

However, a direct comparison with a trivial random generation approach would not bring any useful result. For example, test inputs containing random data may be trivially invalid and useless for extensively testing the system. For this reason, we use as baseline for the comparison the random algorithm employed in previous work [8]; the algorithm samples a portion of the field transmission file, randomly selects a mutation operator and applies it to one of the elements where it can be applied.

The algorithm employed in [8] does not support the generation of test inputs of variable size; furthermore, it does not minimise the generated test suite. To address these limitations and perform a fairer comparison, we execute an improved version of the random algorithm employed in [8] by using our search algorithm with a specific configuration: *minSize* and *maxSize* are set to the same values as the search algorithm; $p_{field} = 1$, to generate a new test input at each iteration by sampling the field data; and $p_{mutation} = 1$, to always mutate the sampled test input like in [8]. The value chosen for $maxMutations$ is irrelevant because we generate a new test input at each iteration (because of $p_{field} = 1$).

To compare with a simple model-based approach, we consider the results achieved with the *All Possible Targets (APT)* approach proposed in [8]. The APT mutation strategy ensures that each class or attribute of the data model is mutated at least once by each of the mutation operators that can be applied to it.

Table III shows the comparison of the search algorithm with random search. Columns *Budget* and *Configuration* report the search budget and the best configurations found, column *Coverage* reports the average, minimum and maximum coverage achieved by the test suite, and column *# Tests* reports the average, minimum and maximum number of test inputs in each test suite. For each search budget we identified the best configuration out of the 54 configurations of the search algorithm with seeding disabled (*Best* in Table III). Furthermore, we identified the best configuration on average over all the search budgets (*BO* in Table III). The comparison with *BO* is fairer since the configuration indicated as *BO* is not optimised for a specific search budget, but is a stable, good overall configuration. For each configuration we report the probability of random sampling (*r*), the probability of applying mutation when sampling (*m*), and the maximum number of allowed mutations in a test (*n*). The best values for maximum coverage and minimum test suite size appear in bold. Our comparison did not include configurations with seeding because it is an optimisation of the search algorithm. The impact of seeding is addressed in *RQ3*.

The search algorithm presented in this paper provides better results than both the random approach and the APT algorithm presented in [8]. APT achieves an average coverage of 23283 instructions, which is less than the coverage obtained with the search and random approaches (see Table III). This is mostly because APT focuses on the coverage of the model and stops after sampling many fewer VCDUs (at most 20,000 VCDUs in [8] while the lowest search budget is 50,000). In summary, the search algorithm presented in this paper generates significantly less test inputs while achieving better coverage. This mainly results from the adoption of fitness functions that help minimise the test suites by keeping only useful test inputs in the archive.

Results also show that the search algorithm achieves better coverage than the random approach. The difference in coverage ranges between 37.6 and 56 instructions. Though the difference in coverage might not appear large, as discussed earlier, even small increases in coverage might exercise important corner cases. Once again, the search algorithm generates significantly smaller numbers of test inputs (e.g. 57.8 versus 34 on average for a 150k budget).

Our conclusions hold even considering the random variation across runs, which is small, as shown by the Min and Max values appearing in Table III.

E. *RQ2: How does fitness based on code coverage affect performance?*

To answer *RQ2*, Table IV shows, for each search budget, the best configurations (*Best*) with and without code coverage fitness (*Code*), with seeding enabled and disabled (*Seeding*). Furthermore, Table IV also reports the configuration that performs better on average over all the search budgets (*BO* in Table IV).

Enabling code coverage fitness results in higher coverage but it comes, however, at the expense of a significantly larger test suite. All configurations in Table IV with higher coverage enable coverage fitness whereas all the ones with smaller test suites do not. For small search budgets, the difference in coverage when enabling coverage fitness is small, thus suggesting that relying on model information is enough, not requiring test execution for generating test cases.

F. *RQ3: How does smart seeding affect performance?*

Table IV shows that smart seeding has a positive effect on cost-effectiveness when the search budget is above 150k. In

TABLE IV. COMPARISONS BETWEEN BEST SEARCH ALGORITHM CONFIGURATIONS BASED ON WHETHER CODE COVERAGE IS EMPLOYED IN THE FITNESS EVALUATION (COLUMN 'CODE'), AND ON WHETHER SMART SEEDING IS ACTIVATED (COLUMN 'SEEDING').

Budget	Code	Seeding	Configuration	Coverage	#Tests	#Mut.
50k	False	0.0	Best: r=0.5,m=1,n=100	23361.4	**17.0**	4.8
	True	0.0	Best: r=0.5,m=1,n=100	23424.4	28.4	3.6
	False	0.5	Best: r=0.5,m=1,n=10	23417.2	21.0	4.0
	True	0.5	Best: r=0.5,m=1,n=10	**23428.4**	34.2	3.2
	True	0.5	BO: r=0.3,m=0,n=10	23401.8	27.0	4.3
100k	False	0.0	Best: r=0.3,m=1,n=10	23404.4	**16.8**	8.2
	True	0.0	Best: r=0.5,m=1,n=100	**23487.8**	31.6	4.9
	False	0.5	Best: r=0.5,m=1,n=10	23442.2	21.0	6.4
	True	0.5	Best: r=0.3,m=0,n=10	23487.0	33.2	5.6
	True	0.5	BO: r=0.3,m=0,n=10	23487.0	33.2	5.6
150k	False	0.0	Best: r=0.8,m=1,n=100	23418.4	28.2	4.0
	True	0.0	Best: r=0.5,m=1,n=100	23502.0	34.0	6.0
	False	0.5	Best: r=0.5,m=1,n=100	23447.4	**23.4**	7.5
	True	0.5	Best: r=0.3,m=0,n=10	**23528.2**	35.6	6.5
	True	0.5	BO: r=0.3,m=0,n=10	**23528.2**	35.6	6.5
200k	False	0.0	Best: r=0.8,m=1,n=100	23426.0	28.0	4.7
	True	0.0	Best: r=0.5,m=0.5,n=100	23519.6	34.6	6.7
	False	0.5	Best: r=0.5,m=1,n=100	23456.0	**23.2**	9.2
	True	0.5	Best: r=0.3,m=0,n=10	**23551.0**	37.2	7.0
	True	0.5	BO: r=0.3,m=0,n=10	**23551.0**	37.2	7.0
250k	False	0.0	Best: r=0.8,m=1,n=100	23433.2	28.6	5.4
	True	0.0	Best: r=0.5,m=1,n=10	23538.6	38.4	7.1
	False	0.5	Best: r=0.5,m=1,n=100	23461.8	**23.6**	10.3
	True	0.5	Best: r=0.3,m=0,n=10	**23554.4**	37.2	7.4
	True	0.5	BO: r=0.3,m=0,n=10	**23554.4**	37.2	7.4

these cases, smart seeding is always part of the configurations that achieve the highest coverage or the lowest number of test cases.

G. RQ4: What are the configuration parameters that affect performance?

For each of the 108 configurations, we calculated their average coverage over all the search budgets (thus considering 25 test suites for each configuration). We ranked these configurations based on their average coverage (not reported in the paper due to space constraints). A detailed analysis showed that coverage fitness was enabled in the top 15 configurations, and never by the worst 15, thus showing its importance to guide the search. The effect of seeding is however much less visible as it depends on other parameters.

Different parameters have different effects depending on the selected search budget. For example, Table IV shows that, for small search budgets (i.e. search budgets including at most 100,000 VCDUs) search achieves better results when more focused on exploitation (i.e. having a low probability of random sampling, like 0.3 and 0.5). Table IV also shows that with higher search budgets, in the absence of coverage fitness or seeding, putting more emphasis on the exploration of the search landscape (i.e. using a 0.8 probability of random sampling) pays off. This result is expected since, with a higher search budget, random sampling allows for the sampling of much of the field data transmission file, roughly one fourth.

Further, Table IV shows some interesting side effects. For search budgets above or equal to 150,000, using either seeding or code coverage decreases the need to explore the search landscape (probability of random sampling decreasing from 0.8 to 0.5). If both are used, even less exploration is needed (probability of random sampling equal to 0.3). This phenomenon can be easily explained for smart seeding, as it does provide more diverse and useful samples in the search

TABLE V. STATISTICAL COMPARISONS OF BEST OVERALL (BO) CONFIGURATION AGAINST BEST WITH NO SEEDING, BEST WITH NO CODE COVERAGE FITNESS FUNCTION, AND RANDOM WITH CODE COVERAGE.

Budget	Configuration	Coverage	$\hat{A}_{12}$	p-value
50k	BO	23401.8	-	-
	BO no seeding	23424.4	0.32	0.403
	BO no code	23417.2	0.40	0.676
	Rand with code	23386.8	0.64	0.530
100k	BO	23487.0	-	-
	BO no seeding	23487.8	0.56	0.835
	BO no code	23442.2	0.92	0.037
	Rand with code	23436.8	0.96	0.022
150k	BO	23528.2	-	-
	BO no seeding	23502.0	0.68	0.403
	BO no code	23443.4	1.00	0.012
	Rand with code	23453.4	0.94	0.027
200k	BO	23551.0	-	-
	BO no seeding	23513.4	0.80	0.144
	BO no code	23448.6	1.00	0.012
	Rand with code	23465.8	1.00	0.012
250k	BO	23554.4	-	-
	BO no seeding	23515.2	0.80	0.144
	BO no code	23450.4	1.00	0.012
	Rand with code	23482.6	1.00	0.012

landscape. In the case of code coverage, this phenomenon occurs because some rare inputs contribute to code coverage but not to other search objectives. Enabling code coverage fitness prevents the algorithm from discarding rare inputs that contribute to code coverage once they are found. In the absence of code coverage fitness, such rare inputs can be easily missed if there is low variety in the test inputs stored in the archive. In the case of higher values of exploration, there is going to be higher variety in the archive, which increases the probability of having those rare inputs.

For completeness, Table IV reports also the average number of mutations per test input (column *#Mut.*). Although the presence of multiple mutations (i.e. data faults) in a same input may trigger hard-to-detect, complex failures, it could also complicate debugging. Table IV shows that on average the number of mutations is low compared to the maximum allowed (e.g. 5.4 versus 100 for a budget of 250k), thus making eventual debugging operations easier[2].

H. RQ5: What configuration should be used in practice?

The best overall configuration (see Table IV) is using $p_{field} = 0.3$ (small probability of sampling a new test data at random instead of reusing the ones already in the archive), $p_{mutation} = 0$ (do not mutate new inputs immediately when sampled), and $maxMutations = 10$. Furthermore, it does use seeding and the code coverage fitness function.

For each search budget, we ran experiments only five times per configuration, due to the high time cost of running them. On one hand, this is useful to get a general picture of cost-effectiveness trends among different parameter configurations. On the other hand, it makes it harder to compare two specific configurations, as the randomness of the algorithm does introduce some degree of noise. Is the best found configuration really better than the others? To address this issue, one could run more experiments just on a subset of configurations of interest. For example, in our case, we are interested in what is

[2]To further simplify debugging our implementation keeps track of the list of mutations applied on each test input and a reference to the mutated element.

the best overall configuration, how it differs when seeding and code coverage fitness functions are or are not used, and how it compares with random search. However, even with just five runs, we obtained statistically significant results regarding our research questions, as reported in Table V.

Following the guidelines in [22], we used the Wilcoxon-Mann-Whitney U-test to check statistical difference quantified by the Vargha-Delaney standardised effect size. For large search budgets, code fitness has the strongest effect (e.g. for a 250k budget, *BO* is statistically significantly better than *BO no code*, with an effect size of 1.00). Also for large budgets, random yields statistically and practically worse results than search (e.g. for a 250k budget, *BO* is statistically significantly better than *Rand with code*, with an effect size of 1.00). But for low budgets, no statistically significant results are visible, which can be explained by low statistical power resulting from lower effect size (less search) and a small number of observations.

X. Related Work

Related work dealing with the automatic generation of faulty input data for system level testing focuses mostly on model-based and mutation testing approaches, while search-based approaches are still in their infancy.

Most model-based testing techniques target the generation of valid data structures to be used in unit testing [23]–[25] that are typically much simpler than the input files needed for testing data processing systems.

Existing model-based techniques like *threat modelling techniques* and *specification mutation testing* generate test inputs that are less complex than those processed by data processing systems. Threat modelling techniques use models that capture the characteristics of typical invalid inputs that should be properly handled by the software under test. Models like attack trees [26], UML state machines [27], and transition nets [28], are used to generate sequences of illegal actions, which are not relevant for testing data processing systems where the complexity of the testing process lies in the definition of the input data structures. Context free grammars instead can generate input data structures [29], [30] but do not allow for the modelling of relationships among data fields.

Specification-based mutation testing techniques use mutation operators to seed faults into specification models (e.g. a state machine) to generate specification mutants [31]. Approaches that generate mutated statecharts can only generate inputs that are sequences of system operations [32], while approaches that mutate class diagrams have only been used to test model transformation systems in which the state diagram itself is the input [33].

Data mutation approaches use mutation operators to generate new test inputs from existing ones [34]–[36]. Like our previous approach [8], the approaches based on data mutation focus on a single objective (e.g. covering all the possible data faults of a fault model) but do not allow for covering the multiple objectives needed to perform effective robustness testing. The answer to *RQ1* in Section IX shows that the search-based approach presented in this paper performs better than approaches that focus on the coverage of a single objective.

Most of the search-based testing techniques have primarily focused on unit testing [37] or the testing of non-functional properties [11]. Work on search-based robustness testing performed at the system level focuses either on the identification of performance issues [38], which are out of the scope of this paper, or functional faults caused by complex sequences of test inputs [39], [40], or by input signals [41], [42].

Ali et al. [39] exploit the information encoded into UML state machines and aspect-oriented modelling to generate test cases that stress the robustness of a software system by generating complex invocations of function calls. Similarly, Fu and Kone [40] use finite state machines to generate robustness test cases for protocol testing. In contrast with these techniques, we focus on systems for which it is important to generate complex data structures; thus, instead of using behavioural models such as state machines, we rely upon exploit class diagrams.

In the context of embedded systems, metaheuristic search is used for the generation of input signals satisfying some given properties such as requirements on signal shapes [41] or temporal constraints [42]. Our work is complementary to these approaches since it addresses the problem of generating complex data structures by innovatively combining metaheuristic search and data mutation.

XI. Conclusion

Building a minimal robustness test suite for data processing systems, with high fault revealing power, is complicated by multiple factors: the complex structure of the test inputs that present several constraints among their data fields, the need for generating a set of inputs that covers both the functional specifications and the data faults captured by a given fault model, and the possibility to have multiple data faults in a same input.

We designed a novel evolutionary algorithm that addresses these challenges by: generating complex test inputs by means of data mutation, relying upon model-based and code-based fitness functions, and identifying optimal test suites by managing the tradeoff between the exploration and the exploitation of the search landscape thanks to a set of configuration parameters. The fitness functions capture aspects that are relevant for robustness testing, that is how well each input covers the structure of the input data, the fault model, the functional specifications, and the structure of the system.

Empirical results obtained by applying our search-based testing approach to test an industrial data processing system show that it outperforms previous approaches based on fault coverage and random generation: higher code coverage is achieved with smaller test suites.

Furthermore, we show that although a fitness function that includes code coverage is essential to maximise the coverage of the generated test suite, fitness functions based on models alone can achieve good coverage results, while significantly reducing test suite size. This is of practical importance as test generation is much quicker and often more practical when no test execution is required. Finally, we identified a best configuration for our search algorithm that returns better results regardless of the search budget; this configuration facilitates the application of our approach and includes smart seeding, which turns out to be a key feature in improving search results.

Acknowledgment

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR 4082113) and an SES grant. The authors would like to thank Raul Gnaga, Vincent Masquelier, and Tomislav Nakic-Alfirevic for their valuable feedback and assistance.

References

[1] Yahoo!, "Stock market search engine." http://finance.yahoo.com/.

[2] Peoplefinders, "People search service." http://www.peoplefinders.com.

[3] FlightRadar24, "Flight tracking web service." http://www.flightradar24.com.

[4] Google Inc., "Google glasses," http://www.google.com/glass/start/.

[5] http://questvisual.com/, "Wordlens camera translation application," http://mashable.com/2010/12/17/word-lens/.

[6] "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[7] The Register, "Dow Jones average collapses to 0.20." http://www.theregister.co.uk/2001/03/19/dow_jones_average_collapses/.

[8] D. Di Nardo, F. Pastore, and L. Briand, "Generating complex and faulty test data through model-based mutation analysis," in *8th International Conference on Software Testing, Verification and Validation (ICST'15)*. IEEE Computer Society, 2015.

[9] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[10] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 742–762, Nov 2010.

[11] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957 – 976, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908001833

[12] M. Iqbal, A. Arcuri, and L. Briand, "Environment modeling and simulation for automated testing of soft real-time embedded software," *Software and Systems Modeling*, vol. 14, no. 1, pp. 483–524, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0328-6

[13] D. Di Nardo, N. Alshahwan, L. Briand, E. Fourneret, T. Nakić-Alfirević, and V. Masquelier, "Model based test validation and oracles for data acquisition systems," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 540–550.

[14] The Object Management Group, "The object constraint language," http://www.omg.org/spec/OCL/.

[15] Council of the Consultative Committee for Space Data Systems, "CCSDS 732.0-B-2 AOS space data link protocol," http://public.ccsds.org/publications/archive/732x0b2c1.pdf, 2006.

[16] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012, pp. 121–130.

[17] G. T. Parks and I. Miller, "Selective breeding in a multiobjective genetic algorithm," in *Parallel Problem Solving From NaturePPSN V*. Springer Berlin Heidelberg, 1998.

[18] J. Knowles and D. Corne, "The pareto archived evolution strategy: A new baseline algorithm for pareto," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*. IEEE Computer Society, 1999.

[19] Mountainminds, "JaCoCo library," http://www.eclemma.org/jacoco/.

[20] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization." in *Proceedings of EUROGEN 2001 - Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, 2001.

[21] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014, pp. 959–967.

[22] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[23] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 123–133.

[24] V. Senni and F. Fioravanti, "Generation of test data structures using constraint logic programming," in *Proceedings of the 6th International Conference on Tests and Proofs*, ser. TAP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 115–131.

[25] S. Khurshid and D. Marinov, "TestEra: Specification-based testing of Java programs using SAT," in *Automated Software Engineering*, vol. 11, no. 4. Hingham, MA, USA: Kluwer Academic Publishers, 2004, pp. 403–434.

[26] A. Morais, A. Cavalli, and E. Martins, "A model-based attack injection approach for security validation," in *Proceedings of the 4th International Conference on Security of Information and Networks*, ser. SIN '11. New York, NY, USA: ACM, 2011, pp. 103–110.

[27] M. Hussein and M. Zulkernine, "UMLintr: A UML profile for specifying intrusions," in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, ser. ECBS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 279–288. [Online]. Available: http://dx.doi.org/10.1109/ECBS.2006.70

[28] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Transactions of Dependable and Secure Computing*, vol. 9, no. 4, pp. 526–539, Jul. 2012.

[29] D. Hoffman, H.-Y. Wang, M. Chang, and D. Ly-Gagnon, "Grammar based testing of HTML injection vulnerabilities in RSS feeds," in *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, ser. TAIC-PART '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 105–110.

[30] S. Zelenov and S. Zelenova, "Automated generation of positive and negative tests for parsers," vol. 3997, pp. 187–202, 2006.

[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[32] R. Schlick, W. Herzner, and E. Jöbstl, "Fault-based generation of test cases from UML-models: Approach and some experiences," in *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 270–283.

[33] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Mutation analysis testing for model transformations," in *Proceedings of the 2nd European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA'06)*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 376–390.

[34] L. Shan and H. Zhu, "Generating structurally complex test cases by data mutation," *Comput. J.*, vol. 52, no. 5, pp. 571–588, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxm043

[35] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, F. Martinelli, and P. Mori, "Testing of PolPA-based usage control systems," *Software Quality Journal*, vol. 22, no. 2, pp. 241–271, 2014.

[36] M. De Jonge and E. Visser, "Automated evaluation of syntax error recovery," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 322–325.

[37] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: http://dx.doi.org/10.1002/stvr.294

[38] L. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10710-006-9003-9

[39] S. Ali, L. C. Briand, and H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems," *Software and Systems Modeling*, vol. 11, no. 4, pp. 633–670, 2012.

[40] Y. Fu and O. Kone, "Security and robustness by protocol testing," *Systems Journal*, vol. 8, no. 3, pp. 699–707, 2014.

[41] A. Baresel, H. Pohlheim, and S. Sadeghipour, "Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms," in *Genetic and Evolutionary Computation ? GECCO 2003*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2724, pp. 2428–2441. [Online]. Available: http://dx.doi.org/10.1007/3-540-45110-2_147

[42] B. Wilmes and A. Windisch, "Considering signal constraints in search-based testing of continuous systems," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, April 2010, pp. 202–211.