Synthetic Data Generation for Statistical Testing

Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand
SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
Email: {ghanem.soltana, mehrdad.sabetzadeh, lionel.briand}@uni.lu

***Abstract*—Usage-based statistical testing employs knowledge about the actual or anticipated usage profile of the system under test for estimating system reliability. For many systems, usage-based statistical testing involves generating synthetic test data. Such data must possess the same statistical characteristics as the actual data that the system will process during operation. Synthetic test data must further satisfy any logical validity constraints that the actual data is subject to. Targeting data-intensive systems, we propose an approach for generating synthetic test data that is both statistically representative and logically valid. The approach works by first generating a data sample that meets the desired statistical characteristics, without taking into account the logical constraints. Subsequently, the approach tweaks the generated sample to fix any logical constraint violations. The tweaking process is iterative and continuously guided toward achieving the desired statistical characteristics. We report on a realistic evaluation of the approach, where we generate a synthetic population of citizens' records for testing a public administration IT system. Results suggest that our approach is scalable and capable of simultaneously fulfilling the statistical representativeness and logical validity requirements.**

***Index Terms*—Test Data Generation, Usage-based Statistical Testing, Model-Driven Engineering, UML, OCL.**

I. Introduction

Usage-based statistical testing, or statistical testing for short, is concerned with detecting faults that cause the most frequent failures (thus affecting reliability the most), and with estimating reliability via statistical models [1]. In contrast to testing techniques that focus on system verification (fault detection), e.g., testing driven by code coverage, statistical testing focuses on system validation from the perspective of users. Statistical testing typically requires a *usage profile* of the system under test. This profile characterizes, often through a probabilistic formalism, the population of the system's usage scenarios [2].

Existing work on usage profiles has focused on state- and event-based systems, with the majority of the work being based on Markov chains [3], [4], [5], [6], [7], [8]. For many systems, which we refer to as *data-centric* and concentrate on in this paper, system behaviors are driven primarily by data, rather than being triggered by stimuli. For example, consider a public administration system that calculates social benefits for citizens. How such a system behaves is determined mainly by complex and interdependent data such as citizens' employment and household makeup. The system's scenarios of use are thus intimately related to the data that is processed by the system. Consequently, the usage profile of such a system is governed by the statistical characteristics of the system's input data, or stated otherwise, by the system's *data profile*. Given our focus on data-centric systems and the explanation above, we equate, for the purposes of this paper, "usage profile" and "data profile", and use the latter term hereafter.

When actual data, e.g., real citizens' records in the aforementioned example, is available, one may be able to perform statistical testing without a data profile. In most cases, however, gaps exist in actual data, since new and retrofit systems may require data beyond what has been recorded in the past. These gaps need to be filled with *synthetic data*. To generate synthetic data that is representative and thus suitable for statistical testing, a profile of the missing data will be required.

Further, and perhaps more importantly, synthetic data (and hence a data profile) are indispensable when access to actual data is restricted. Notably, under most privacy regulations, e.g., EU's General Data Protection Regulation [9], "repurposing" of personal data is prohibited without explicit consent. This complicates sharing of any actual personal data with third-parties who are responsible for software development and testing. Anonymization offers a partial solution to this problem; however, doing so often comes at the cost of reduced data quality [10] and proneness to deanonymization attacks [11].

Data profiles have received little attention in the literature on software testing. This is despite the fact that many data-centric systems, e.g., public administration and financial systems, are subject to reliability requirements and thus statistical testing. Recently, we proposed a statistical data profile and a heuristic algorithm for generating representative synthetic data [12]. Although motivated by microeconomic simulation [13] rather than software testing, our previous approach provides a useful basis for generating data that can be used for statistical testing. However, the approach suffers from an important limitation: while the approach generates synthetic data that is aligned with a desired set of statistical distributions and has shown to be good enough for running financial simulations [14], the approach cannot guarantee the satisfaction of *logical constraints* that need to be enforced over the generated data.

To illustrate, we note three among several other logical anomalies that we observed when using our previous approach [12] for generating test cases based on a data profile of citizens' records: children who were older than their parents, individuals who were married before being born, and individuals who were classified as widower without ever having been married. Without the ability to enforce logical constraints to avoid such anomalies, the generated data is unsuitable for statistical testing and estimating reliability. This is because such anomalies may result in exceptions or system behaviors that are not meaningful. In either case, targeted system behaviors will not be exercised.

The question that we investigate in this paper is as follows: ***Can we generate synthetic test data that is both statistically representative and logically valid?*** The key challenge we need to address when tackling this question is *scalability*. Specifically, to obtain statistical representativeness, we need to construct a *large* data sample (test suite), potentially with *hundreds* or *thousands* of members. At the same time, this large sample has to satisfy logical constraints, meaning that we need to apply computationally-expensive constraint solving.

Contributions. The contributions of this paper are as follows:

1) We develop a model-based test data generator that can simultaneously satisfy statistical representativeness and logical validity requirements over complex, interdependent data. The desired statistical characteristics are expressed using our previously-developed probabilistic UML annotations [12]. Validity constraints are expressed using UML's constraint language, OCL [15]. Our data generator incorporates two collaborating components: (a) a search-based OCL constraint solver which enhances previous work by Ali et al. [16], and (b) a mechanism that guides the solver toward satisfying the statistical characteristics that the generated data (test suite) must exhibit.

2) We evaluate our data generator through a realistic case study, where synthetic data is required for statistical testing of a public administration IT system in Luxembourg. Our results suggest that our data generator can create, in practical time, test data that is sound, i.e., satisfies the necessary validity constraints, and at the same time, is closely aligned with the desired statistical characteristics.

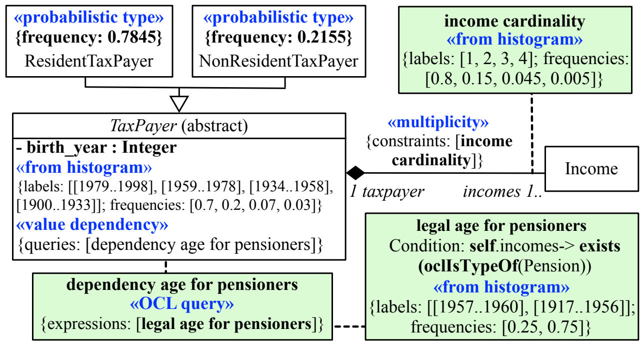
II. Background

In this section, we briefly describe our previous data generation approach [12]. We leverage this approach for (1) expressing the desired statistical characteristics of data, and (2) generating initial data samples which we process further to achieve not only representativeness but logical validity as well.

For specifying statistical characteristics, we use a set of annotations (stereotypes) which can be attached to a data schema expressed as a UML Class Diagram. Fig. 1 illustrates some of these annotations on a small excerpt of a data schema for a tax administration system.

The *«probabilistic type»* stereotypes applied to the specializations of the *TaxPayer* class state that ≈78% of the taxpayers should be resident and the remainder should be non-resident. The *«from histogram»* stereotype attached to the *birth_year* attribute provides, via a histogram, the birth year distribution for taxpayers. The attribute *birth_year* is further annotated with a conditional probability specified via the *«value dependency»* stereotype. The details of this conditional probability are provided by the *legal age for pensioners* box. The information in the box reads as follows: 25% of pensioners have their birth year between 1957 and 1960, i.e., are between 57 and 60 years old; the remaining 75% are older than 60.

The *«multiplicity»* stereotype attached to the association between *TaxPayer* and *Income* classes describes, via the *income cardinality* histogram, the distribution of the number of incomes per taxpayer. As shown in Fig. 1, 80% of the taxpayers have one income, 15% have two, and so on.

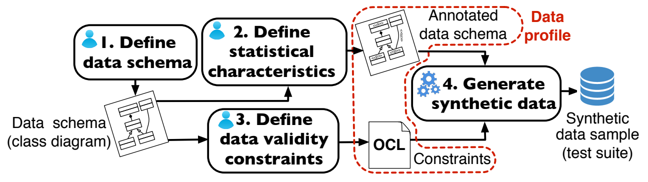

«probabilistic type»
{frequency: 0.7845}
ResidentTaxPayer
«probabilistic type»
{frequency: 0.2155}
NonResidentTaxPayer
income cardinality
«from histogram»
{labels: [1, 2, 3, 4]; frequencies: [0.8, 0.15, 0.045, 0.005]}
TaxPayer (abstract)
- birth_year : Integer
«from histogram»
{labels: [[1979..1998], [1959..1978], [1934..1958], [1900..1933]]; frequencies: [0.7, 0.2, 0.07, 0.03]}
«value dependency»
{queries: [dependency age for pensioners]}
«multiplicity»
{constraints: [income cardinality]}
Income
1 taxpayer
incomes 1..
legal age for pensioners
Condition: self.incomes-> exists (oclIsTypeOf(Pension))
«from histogram»
{labels: [[1957..1960], [1917..1956]]; frequencies: [0.25, 0.75]}
dependency age for pensioners
«OCL query»
{expressions: [legal age for pensioners]}
Fig. 1. Data Schema (Excerpt) Annotated with Statistical Characteristics

For generating a data sample, we previously proposed a heuristic technique that is aimed exclusively at representativeness [12]. This technique traverses the elements of the data schema and instantiates them according to the prescribed probabilities. The technique attempts to satisfy multiplicity constraints but satisfaction is not guaranteed. More complex logical constraints are not supported.

In this paper, we use as a starting point the data generated by our previous approach, and alter this data to make it valid without compromising representativeness. Indeed, as we show in Section V, our new approach not only results in logically valid data but also outperforms our previous approach in terms of representativeness.

III. Approach Overview

Fig. 2 presents an overview of our approach for generating representative and valid test data. Steps 1–3 are manual and Step 4 is automatic. In Step 1, *Define data schema*, we define using a UML Class Diagram (CD) [17] the schema of the data to generate. This diagram, illustrated earlier in Fig. 1, is the basis for: (a) capturing the desired statistical characteristics of data (Step 2), and (b) generating synthetic data (Step 4).


1. Define data schema
2. Define statistical characteristics
Annotated data schema
Data profile
4. Generate synthetic data
Synthetic data sample (test suite)
Data schema (class diagram)
3. Define data validity constraints
OCL
Constraints
Fig. 2. Approach for Generating Valid and Representative Synthetic Data

In Step 2, *Define statistical characteristics*, the CD from Step 1 is enriched with probabilistic annotations (see Section II) to express the representativeness requirements that should be met during data generation in Step 4. In Step 3, *Define data validity constraints*, users express via the Object Constraint Language (OCL) [15] the logical constraints that the generated data must satisfy. For example, the following OCL constraint states that children must be born at least 16 years after their parents: **self**.children->**forAll**(c| c.birth_year > self.birth_year + 16). Here, **self** refers to a person.

Steps 2 and 3 of our approach can in principle be done in parallel. Nevertheless, it is advantageous to perform Step 3 *after* Step 2. This is because the probabilistic annotations of Step 2 may convey some implicit logical constraints. For example, the annotations of Step 2 may specify a uniform distribution over the month of birth for physical persons. It would therefore be redundant to define the following OCL constraint: **self**.birth_month >= 1 and **self**.birth_month <= 12. Such redundancies can be avoided by doing Steps 2 and 3 sequentially.

Step 4, *Generate synthetic data*, generates a data sample (test suite) based on a data profile. In our approach, a data profile is materialized by the combination of the probabilistic annotations from Step 2 and the OCL constraints for Step 3. As stated earlier, the synthetic data generated in Step 4 must meet both the statistical representativeness and logical validity requirements, respectively specified in Steps 2 and 3. The output from Step 4 is a collection of instance models, i.e., instantiations of the underlying data schema. Each instance model characterizes *one* test case for statistical testing.

In the next section, we elaborate Step 4, which is the main technical contribution of this paper.

IV. Generating Synthetic Data

In this section, we describe our synthetic data generator. Fig. 3 shows the strategy employed by the data generator. Initially, a potentially invalid collection of instance models is created using our previous data generation approach (see Section II). We refer to this initial collection as the *seed sample*. Our data generator then transforms the seed sample into a collection of valid instance models. This is achieved using a customized OCL constraint solver, presented in Section IV-A.

The solver attempts to repair the invalid instance models in the seed sample. To do so, the solver considers the constraint specified in Step 3 of our overall approach (Fig. 2) alongside the multiplicity constraints of the underlying data schema and the constraints implied by the probabilistic annotations from Step 2 of the approach. The rationale for feeding the solver with instance models from the seed sample, rather than having the solver build instance models from scratch, is based on the following intuitions: (1) By starting from the seed sample, the solver is more likely to be able to reach valid instance models, and (2) The valid sample built by the solver will not end up too far away from being representative, in turn making it easier to fix deviations from representativeness, as we discuss later.

The OCL solver that we use is based on metaheuristic search. If the solver cannot fix a given instance model within a predefined maximum number of iterations, the instance model is discarded. To compensate, the seed sample is extended with a new instance model byre-invoking our previous data generator. This process continues until we obtain the desired number of valid instance models (test cases). The number of instance models to generate is an input parameter that is set by users.

Once we have a valid data sample that has the requested number of instance models in it, our data generator attempts to realign the sample back with the desired statistical characteristics. This is done through an iterative process, delineated in Fig. 3 with a dashed boundary. We elaborate the details of this iterative process in Section IV-B.

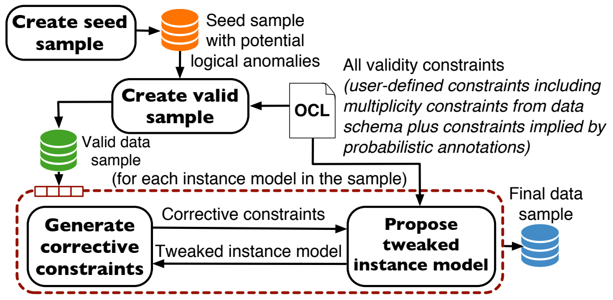

Create seed sample
Seed sample with potential logical anomalies
Create valid sample
OCL
All validity constraints (user-defined constraints including multiplicity constraints from data schema plus constraints implied by probabilistic annotations)
Valid data sample
(for each instance model in the sample)
Final data sample
Generate corrective constraints
Corrective constraints
Tweaked instance model
Propose tweaked instance model

Fig. 3. Overview of our Data Generation Strategy

Briefly, the process goes in a sequential manner through the instance models within the valid sample, and subjects these instance models to additional constraints that are generated on-the-fly. These additional constraints, which we refer to as *corrective constraints*, provide cues to the solver as to how it should tweak an instance model so that the statistical representativeness of the whole data sample is improved.

For example, let us assume that instance models represent households in a tax administration system. Now, suppose that the proportion of households with no children is over-represented in the sample. If, under such circumstances, the iterative process is working on a household with no children, a corrective constraint will be generated stating that the number of children should be non-zero (in that particular household). The solver will then attempt to satisfy this constraint without violating any of the validity constraints discussed earlier.

If the solver fails to come up with a tweaked household that satisfies both the corrective constraint and all the validity constraints at the same time, the original household (which is valid but has no children) is retained in the sample. Otherwise, that is, when a tweaked and valid household is found, we need to decide whether it is advantageous to replace the original household by the tweaked one. Let I be the original household and I' the tweaked one. Further, let $\mathcal{S}$ denote the current sample containing I (but not I') and let $\mathcal{S}' = (\mathcal{S} \setminus \{I\}) \cup \{I'\}$. The decision is made as follows: If $\mathcal{S}$ is better aligned than $\mathcal{S}'$ with the desired statistical characteristics then I' is discarded; otherwise, I' will replace I in the sample. The reason why this decision is required is because tweaking may have side effects. Therefore, I' may not necessarily improve overall representativeness, although it does reduce the proportion of households with no children. For example, it could be that the solver adds some children to the household in question, but in doing so, it also changes the household allowances. These allowances too may be subject to representativeness requirements. Without the comparison above, one cannot tell whether the tweaked household is a better fit for representativeness.

In the above scenario, we illustrated the iterative process using a single corrective constraint. In practice, the process may generate multiple such constraints, since the data sample at hand may be deviating from multiple representativeness requirements. We treat corrective constraints as being *soft*. This means that if after the maximum number of iterations, the solver manages to solve some of the corrective constraints but not all, the process will give the tweaked instance model a chance to replace the original one *as long as* the tweaked instance model still satisfies all the validity constraints.

The rest of this section presents the technical machinery behind the (customized) OCL solver and our data generator.

A. Solving OCL Constraints

A number of techniques exist for solving OCL constraints, notably using Alloy [18], [19], constraint programming [20], and (metaheuristic) search [21], [16]. Alloy often fails to solve constraints that involve large numbers [22]. We observed via experience that this limitation could be detrimental in our context. For example, our case study in Section V has several constrained quantities, e.g., incomes and allowances, that are large numbers. As for constraint programming, to our knowledge, the only publicly-available tool is UML2CSP [20]. We observed that this tool did not scale for our purposes. In particular, given a time budget of 2 hours, UML2CSP did not produce any valid instance model in our case study. This is not practical for statistical testing where we need a representative sample with many (*hundreds* or more) valid instance models.

Search, as we demonstrate in Section V, is more promising in our context. Although search-based techniques cannot prove (un)satisfiability, they are efficient at exploring large search spaces. In our work, we adopt with two customizations the search-based OCL solver of Ali et al.'s [16], hereafter referred to as the *baseline solver*. The customizations are: (1) a feature for setting a specific instance model as the starting point for search, and (2) a strategy to avoid premature narrowing of the search space. The former customization, which is straightforward and not discussed here, is necessary for realizing the process in Fig. 3. The latter customization is discussed next.

The baseline solver has a fixed heuristic for selecting what OCL clause to solve next: it favors clauses that are closer to being satisfied based on a fitness function. For example, assume that we want to satisfy constraint *C1* defined as follows: **if** (x=2) **then** y=5 **else if** (x=3) **then** y=4 **else** y=0 **endif endif**, where x and y are attributes. For the sake of argument, suppose the solver is processing a candidate solution where $x = 3$ (satisfying the condition of the second nested if statement) and $y = 7$ (not satisfying any clause). This makes the second nested if statement in *C1* the closest to being satisfied. At this point, the heuristic employed by the solver narrows the search space by locking the value of x and starting to exclusively tweak y in order to satisfy y=4. Now, if we happen to have another constraint *C2* stating y>4, the solver will fail since x can no longer be tweaked.

The above heuristic in the baseline solver poses no problem as long as the goal is to find *some* valid solution. If search fails from one starting point, the solver (pseudo-)randomly picks another and starts over. In our context however, starting over from an arbitrary point is not an option. For the final data sample to have a chance of being aligned with the desired statistical characteristics, the solver needs to use as starting point instance models from a statistically representative seed sample (see Fig. 2). If the solver fails at making valid an instance model from the seed sample, that instance model has to be discarded. This negatively affects performance, since the solver will need to start all over on a replacement instance model supplied by the seed data generator, as noted earlier.

In a similar vein, if the solver fails at enforcing corrective constraints over a (valid) instance model, it cannot help with improving representativeness. To illustrate, suppose that constraint *C2* mentioned earlier is a corrective constraint and that the valid solution (instance model for *C1*) is $x = 3,\ y = 4$. In such a case, the baseline solver will deterministically fail as long as the starting point is this particular valid solution. In other words, *C2* will have no effect.

To address the above problem, we customize the baseline solver as follows: Rather than working directly on the original constraints, the customized solver works on the constraints' *prime implicants (PI)*. An implicant is prime (minimal) if violating any of its literals results in the violation of the underlying constraint. To derive all the PIs for a given OCL constraint, we first transform the constraint into a logical expression with only ANDs and ORs, negation, and OCL operations. We next transform this expression into Disjunctive Normal Form (DNF) by applying De Morgan's law [23]. Each clause of the DNF expression is a PI. For instance, constraint *C1* yields three PIs: (x=2 **and** y=5), (x<>2 **and** x=3 **and** y=4), and (x<>2 **and** x<>3 **and** y=0). Note that we use the term PI slightly differently than what is standard in logic. Our literals are not necessarily independent logically. For example, in the second PI above, x<>2 is redundant because x=3 implies x<>2. Such redundancies pose no problem and are ignored.

For each constraint C to be solved, the customized solver *randomly* picks one of C's PIs. For instance, if we want to solve constraints *C1* and *C2* together, we would randomly pick one of *C1*'s three PIs alongside *C2* (whose only PI is y>4). This way, we give a chance to all PIs to be considered, thus avoiding the undesirable situation discussed earlier, where the baseline solver would (deterministically) lead itself into dead-ends. For example, from the PIs of *C1*, we may pick (x=2 **and** y=5). Now, if we start the search at $x = 3,\ y = 7$, the solver will have a feasible path toward a valid solution, $x = 2,\ y = 5$, which satisfies both *C1* and *C2*. If a certain combination of randomly-selected PIs (one PI per constraint) fails, other combinations are tried until either a solution is found or the maximum number of iterations allowed is reached.

Due to space, we cannot present all the details of this customization. We only make two remarks. First, all OCL operations are treated as opaque literals when building PIs. For example, the operation **self**.*navigation*−>**forAll**(x=3 **or** y=2) is a single literal, just like, say, x=3. Solving OCL operations is a recursive process and similar to solving operation-free expressions. In particular, to solve OCL operations, we employ the same DNF transformation discussed earlier. For example, to solve **self**.*navigation*−>**forAll**(x=3 **or** y=2), we derive two PIs, x=3 and y=2, and use them to constrain the objects at the association end that has *navigation* as role name.

Second, the DNF transformation can result in exponentially large DNF representations when there are many literals [24]. Such exponential explosion is unlikely to arise in our context:

Manually-written logical constraints for data models typically include only a handful of literals. For the corrective constraints that are generated automatically (through Alg. 2 described later), the number of literals is at most as many as the number of ranges (or categories) in the bar graphs that capture the desired statistical distributions. Again, these numbers are seldom very large. In our case study of Section V, the DNF transformations took negligible time (milliseconds).

To summarize, using PIs instead of the original constraints helps avoid dead-ends when solution search has to start from a specific point in the search space. In Section V (RQ1), we examine how customizing the baseline solver in the manner described in this section influences performance.

B. Generating Valid and Representative Data

This section presents the technical details of our data generation strategy, depicted in Fig. 3 and outlined earlier on. We already discussed the creation of the seed sample (from our previous work [12]) and how we make this sample valid using a customized OCL solver (Section IV-A). Below, we focus on the iterative process in Fig. 3, i.e., the region delineated by dashed lines, and present the algorithms behind this process.

We start with some remarks about how we represent statistical distributions. The instruments we use to this end are *barcharts* (for categorical quantities) and *histograms* (for ordinal and interval quantities). Without loss of generality, and while we support both notions, we talk exclusively about histograms in the text. A histogram is a set of *bins*. Each bin is defined by a label (value or range), and a *relative frequency* denoting the proportional abundance of the bin's label. We do not directly handle continuous distributions, e.g., the normal distribution. Continuous distributions are discretized into histograms. Doing so is routine [25] and not explained here. We note however that the discretization should not be too fine-grained, e.g., resulting in more than 100 bins. This is because the corrective constraints in our approach will get complex, in turn posing scalability issues for the OCL solver, e.g., with respect to the DNF transformation discussed in Section IV-A.

The PIM algorithm. Alg. 1, *Process Instance Model (PIM)*, presents the procedure for one iteration of the iterative process (region within the dashed boundary) in Fig. 3. PIM takes the following as input: (1) a valid data sample, (2) a specific instance model from the sample to process, (3) a set of validity constraints, (4) the desired statistical characteristics defined as histograms, (5) a parameter specifying how many attempts the algorithm should make to generate tweaked instance models, and (6) a parameter specifying how sensitive the algorithm is to differences in relative frequencies. Essentially, if the difference between two relative frequencies is below the sensitivity parameter, the two frequencies are considered equal.

The algorithm works in three stages as we describe next.

1) Generate corrective constraints (L. 1-3 of Alg. 1): In this stage, PIM calls another algorithm GCC (Alg. 2, described later). GCC generates corrective constraints for the instance model being processed (L. 1). For example, assume that the instance model is a pensioner, and that pensioners are currently over-represented in the data sample. In response, GCC will generate the following corrective constraint, named *CC1*: **self**.incomes->**forAll**(**not oclIsTypeOf**(Pension)). If GCC does not yield any corrective constraints, then the original instance model will be retained in the sample (L. 2-3).

2) Build tweaked instance models (L. 4-10 of Alg. 1): In this stage, PIM attempts to produce a set of tweaked instance models based on the corrective constraints generated previously. These constraints are fed to the solver alongside the validity constraints (L. 7). To illustrate, consider the example corrective constraint *CC1* generated at the first stage. This constraint instructs the solver to tweak the instance model at hand so that the income type will no longer be pension. PIM tries building tweaked instance models multiple times (L. 6). This is intended at coming up with multiple candidates (ideally more than one) for replacing the original instance model in the sample. As noted earlier, we treat corrective constraints as soft and try to satisfy them on a best-effort basis. Therefore, any tweaked instance model returned by the solver will be included in the set of candidate replacements as long as the validity constraints hold (L. 9-10).

3) Select best replacement (L. 11-18 of Alg. 1): In this stage, PIM chooses to either retain the original instance model or replace it with one of the tweaked instance models computed in the second stage. The criterion applied for the decision is which instance model, once incorporated into the sample, will result in the most statistically representative sample.

The metric we use for measuring statistical representativeness is *Euclidean distance* [26]. This metric measures

Alg. 1: Process Instance Model (**PIM**)

```
Inputs    : (1) a set $\mathcal{S}$ of valid instance models; (2) an instance model
            inst $\in \mathcal{S}$ to process; (3) a set $\mathcal{V}$ of validity constraints;
            (4) a set $\mathcal{H}_{\text{desired}}$ of desired statistical characteristics
            (expressed as histograms); (5) a parameter nb_attempts
            denoting the number of times that the solver will be
            invoked over inst to create tweaked instance models; (6) a
            parameter freq_sensitivity $\in [0..1]$ denoting the margin
            beyond which two relative frequencies are deemed far apart.
            /* freq_sensitivity is used only for invoking GCC (Alg. 2). */
Output    : Either inst or a tweaked instance model, whichever leads to
            a more representative data sample.
Fun. calls: GCC: generates corrective constraints (Alg. 2);
            solve: invokes the customized solver (see Section IV-A).
1  $\mathcal{CC} \leftarrow$ GCC($\mathcal{S}$, inst, $\mathcal{H}_{\text{desired}}$, freq_sensitivity)
   /* CC is the set of corrective constraints returned by Alg. 2. */
2  if ($\mathcal{CC} = \emptyset$) then
3      return inst
4  $\mathcal{T} \leftarrow \emptyset$ /* $\mathcal{T}$ will store potential replacements for inst. */
5  i $\leftarrow 0$ /* i is the number of times the solver has been invoked so far. */
6  while (i < nb_attempts) do
7      inst_tweaked $\leftarrow$ solve(inst, $\mathcal{V} \cup \mathcal{CC}$)
8      i $\leftarrow$ i + 1
9      if (inst_tweaked satisfies the constraints in $\mathcal{V}$) then
10         $\mathcal{T} \leftarrow \mathcal{T} \cup$ {inst_tweaked}
11 inst_best $\leftarrow$ inst
12 $\mathcal{S}_{\text{best}} \leftarrow \mathcal{S}$
13 foreach candidate $\in \mathcal{T}$ do
14     $\mathcal{S}' \leftarrow (\mathcal{S} \setminus$ {inst}) $\cup$ {candidate}
15     if ($\mathcal{S}'$ is better aligned with $\mathcal{H}_{desired}$ than $\mathcal{S}_{best}$) then
16         inst_best $\leftarrow$ candidate
17         $\mathcal{S}_{\text{best}} \leftarrow \mathcal{S}'$
18 return inst_best
```

how far two histograms are from one another. The closer the distance between two histograms is to zero, the better aligned the histograms are. For example, suppose that the data sample is composed of 40% resident versus 60% non-resident taxpayers. As showed in the data schema excerpt in Fig. 1, the desired distribution is ≈78% resident versus ≈22% non-resident. The Euclidean distance between the data sample and the desired distribution is ≈0.55, indicating that the sample is not representative. Since PIM needs to take into account several distributions simultaneously, it uses the average of the Euclidean distances computed for all the histograms.

We next describe the GCC algorithm that PIM calls (on L. 1 of Alg. 1) for generating corrective constraints.

The GCC algorithm. Given an instance model inst within a data sample $\mathcal{S}$, Alg 2., titled *Generate Corrective Constraints (GCC)*, provides suggestions (in the form of constraints) as to how inst can be tweaked so that $\mathcal{S}$ will become a more representative sample. The input to GCC was described previously as part of PIM's input. GCC works in three stages as explained below. Throughout the explanation, we will be referring to Table I, Table II and Fig. 4 for illustration.

1) Generate OCL literals (L. 1-18 of Alg. 2): In this stage, GCC groups histograms that annotate the same data schema element, i.e., class, attribute or association, as illustrated in Fig. 1 (L. 4-11). For each group, sets $\mathcal{O}$ and $\mathcal{U}$ will be built (L. 12-18). These two sets contain OCL literals for *O*ver-represented and *U*nder-represented histogram bins, respectively. These literals will later be assembled into intermediate constraints (see second stage below).

The literals in $\mathcal{O}$ and $\mathcal{U}$ are derived as follows: We compare in a pairwise manner the relative frequencies of the actual characteristics of $\mathcal{S}$ against the desired characteristics in $\mathcal{H}_{\text{desired}}$ (L. 13-15). To illustrate, consider rows 2 and 3 of Table I. The relative frequencies to compare are $F1$ against $D1$, $F2$ against $D2$, and so on. If for an index i, $|Fi - Di| >$ freq_sensitivity (L. 15), the algorithm will generate a literal. Whether an exclusion or inclusion literal is generated depends on whether the underlying bin is over- or under-represented (L. 16-18). For example, in Table I, the difference between $F1$ and $D1$ is $|0.9 - 0.7| = 0.2$, which is larger than the (user-provided) freq_sensitivity value on row 4 of Table I. Since $F1$ is over-represented, the following literal is added to $\mathcal{O}$ in order to exclude $L1$: TaxPayer.**allInstances**()->**select**(id = 1)->**forAll**(**not** (birth_year >= 1979 **and** birth_year <= 1998)). Note that the generated literal targets the specific instance model being processed, since ultimately, the literal is intended at tweaking that particular instance model. The case for under-representation is dual and not illustrated.

2) Combine literals (L. 19-21 of Alg. 2): In the second stage, the algorithm combines the literals in $\mathcal{O}$ and $\mathcal{U}$ into what we call an *intermediate* constraint. We use the term "intermediate" to distinguish the output of this stage from the final corrective constraint built in the next (third) stage of the algorithm, described later. In particular, in the final corrective constraint, we have to account for the fact that some histograms apply only under certain conditions. For example, histogram $H2$ on row 2 of Table I applies to pensioners only. This detail is not captured by the literals in $\mathcal{O}$ and $\mathcal{U}$.

Alg. 2: Generate Corrective Constraints (**GCC**)

Inputs : (1) a set $\mathcal{S}$ of valid instance models; (2) an instance model inst $\in \mathcal{S}$ for which corrective constraints should be generated; (3) a set $\mathcal{H}_{\text{desired}}$ of desired statistical characteristics (expressed as histograms); (4) a parameter freq_sensitivity $\in [0..1]$ denoting the margin beyond which two relative frequencies are deemed far apart.

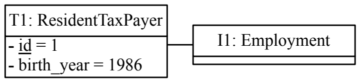
Output : A set $\mathcal{CC}$ of corrective constraints for inst.

Fun. calls: **includeBin** (resp. **excludeBin**): generates an OCL literal prescribing the inclusion (resp. exclusion) of a specific histogram bin.

```
1  $\mathcal{CC} \leftarrow \emptyset$
2  $\mathcal{H}_{\text{current}} \leftarrow$ Statistical characteristics of $\mathcal{S}$
3  $\mathcal{M} \leftarrow \{H \in \mathcal{H}_{\text{current}} \mid H \mapsto ""\}$ /* $\mathcal{M}$ maps each histogram in $\mathcal{H}_{current}$ onto an "intermediate" constraint (explained in the text). All histograms are initially mapped onto an empty expression.
4  $\mathcal{P} \leftarrow \emptyset$ /* $\mathcal{P}$ will store histograms (from $\mathcal{H}_{current}$) which have been already processed. */
5  foreach $H \in \mathcal{H}_{current}$ do
6      if ($H \in \mathcal{P}$) then
7          continue /* We have already processed H and thus skip the loop. */
8      Let $e$ be the data schema element to which $H$ has been attached
9      Let $\mathcal{L}$ be the set of all histograms in $\mathcal{H}_{\text{current}}$ that annotate $e$
10     foreach $L \in \mathcal{L}$ do
11         $\mathcal{P} \leftarrow \mathcal{P} \cup \{L\}$ /* Histogram L is marked as processed. */
12         Let $\mathcal{O}$ and $\mathcal{U}$ be initially empty sets of OCL literals
           /* $\mathcal{U}$ stores literals generated for Under-represented bins;
           $\mathcal{O}$ stores literals generated for Over-represented bins. */
13         foreach relative frequency $F \in L$ do
14             Let $D$ be the relative frequency in $\mathcal{H}_{\text{desired}}$ corresponding to $F$
15             if ($|F - D| >$ freq_sensitivity) then
16                 if ($F > D$) then
17                     $\mathcal{O} \leftarrow \mathcal{O} \cup \{$excludeBin(inst, $F$)$\}$;
18                 else $\mathcal{U} \leftarrow \mathcal{U} \cup \{$includeBin(inst, $F$)$\}$;
19         if ($\mathcal{O} \neq \emptyset$ or $\mathcal{U} \neq \emptyset$) then
20             $\text{OCL}_{\text{intermediate}} \leftarrow \left(\bigwedge_{j=1}^{j=|\mathcal{O}|} \mathcal{O}_j \wedge \bigvee_{j=1}^{j=|\mathcal{U}|} \mathcal{U}_j\right)$ /* See Fig. 4. */
21             $\mathcal{M} \leftarrow \mathcal{M} \cup \{L \mapsto \text{OCL}_{\text{intermediate}}\}$
22     $\mathcal{A} \leftarrow \{A \in \mathcal{M} \mid \mathcal{M}(A) \neq ""\}$ /* $\mathcal{A}$ is the set of all histograms in $\mathcal{M}$ with a non-empty intermediate constraint */
23     if ($\mathcal{A} \neq \emptyset$) then
24         if ($|\mathcal{A}| = 1$ and $\mathcal{M}$(single histogram in $\mathcal{A}$) is unconditional) then
25             $\text{OCL}_{\text{final}} \leftarrow \mathcal{M}$(single histogram in $\mathcal{A}$) /* Row 1 of Table II */
26         else
27             $\text{OCL}_{\text{else}} \leftarrow$ "true" /* $OCL_{else}$ will store the "catch all" else rule when all of $\mathcal{A}$'s histograms are conditional (Row 3 of Table II) */
28             foreach $A \in \mathcal{A}$ do
29                 $\text{condition}_A \leftarrow$ "true" /* $\text{condition}_A$ will store the OCL condition for histogram A's intermediate constraint. */
30                 if (A is conditional) then
31                     $\text{condition}_A \leftarrow$ condition of $A$
32                     $\text{OCL}_{\text{else}} \leftarrow \text{OCL}_{\text{else}} \wedge (\neg\ \text{condition}_A)$
33                     foreach $B \in (\mathcal{A} \setminus \{A\})$ do
34                         /* Now, complete A's condition based on other histograms in $\mathcal{A}$. */
35                         if (B is conditional) then
36                             $\text{condition}_A \leftarrow \text{condition}_A \wedge (\neg\ \text{condition}_B)$
37                 $\text{OCL}_{\text{final}} \leftarrow \text{OCL}_{\text{final}} \vee (\text{condition}_A \wedge \mathcal{M}(A))$
38             if (all histograms in $\mathcal{A}$ are conditional) then
39                 $\text{OCL}_{\text{final}} \leftarrow \text{OCL}_{\text{final}} \vee \text{OCL}_{\text{else}}$
40         $\mathcal{CC} \leftarrow \mathcal{CC} \cup \{\text{OCL}_{\text{final}}\}$ /* Store $OCL_{final}$ in CC. */
41     $\mathcal{M} \leftarrow \{H \in \mathcal{H}_{\text{current}} \mid H \mapsto ""\}$ /* Reset $\mathcal{M}$. */
42 return $\mathcal{CC}$
```

The construction of the intermediate constraint is straightforward, noting that we take the *conjunction* of the literals in $\mathcal{O}$ which prescribe *exclusions*, and the *disjunction* of the literals in $\mathcal{U}$ which prescribe *inclusions* (L. 20). In Fig. 4, we provide

TABLE I
ILLUSTRATIVE EXAMPLE FOR ALG. 2

	Construct	Value
1	Excerpt of the instance model to process.	 T1: ResidentTaxPayer - id = 1 - birth_year = 1986 I1: Employment
2	Desired statistical characteristics ($\mathcal{H}_{\text{desired}}$): For simplicity, we limit our illustration to the histograms attached to the *birth_year* attribute of *TaxPayer* in Fig. 1.	The first histogram, $H1$, attached to *birth_year*: - Bin labels: $\{L1$=[1979..1998], $L2$=[1959..1978], $L3$=[1934..1958], $L4$=[1900..1933]$\}$ - Relative Frequencies: $\{D1$=0.7, $D2$=0.2, $D3$=0.07, $D4$=0.03$\}$ - Condition: true (none)
		The second histogram, $H2$, attached to *birth_year*: - Bin labels: $\{L5$=[1957..1960], $L6$=[1917..1956]$\}$ - Relative Frequencies: $\{D5$=0.25, $D6$=0.75$\}$ - Condition: **self**.incomes->**exists** (**oclIsTypeOf**(Pension))
3	Statistical characteristics of the current sample ($\mathcal{H}_{\text{current}}$ computed on L. 2 of Alg. 2). $\mathcal{H}_{\text{current}}$ differs from $\mathcal{H}_{\text{desired}}$ only in the relative frequencies.	Histogram $H1'$ for the sample (differs from $H1$ on row 2 above only in relative frequencies): - Relative Frequencies for $H1'$: $\{F1$=0.9, $F2$=0.05, $F3$=0.05, $F4$=0$\}$
		Histogram $H2'$ for the sample (differs from $H2$ on row 2 only in relative frequencies): - Relative Frequencies for $H2'$: $\{F5$=0.5, $F6$=0.5$\}$
4	freq_sensitivity.	0.03

```
         ((TaxPayer.allInstances()->select(id = 1)->
         forAll(not(birth_year >= 1979 and birth_year <= 1998)))
From 𝒪 { and
         (TaxPayer.allInstances()->select(id = 1)->
         forAll(not(birth_year >= 1959 and birth_year <= 1978)))
                and
From 𝒰 { (TaxPayer.allInstances()->select(id = 1)->
         forAll(birth_year >= 1900 and birth_year <= 1933))
```

Fig. 4. Intermediate OCL Constraint for Distribution $H1'$ in Table I

an example of an intermediate constraint for histogram $H1'$, shown on row 3 of Table I.

3) Generate final constraints (L. 22-41 of Alg. 2): In the third (and final) stage, the algorithm (1) adds to the intermediate constraints conditions that describe under what circumstances these constraints apply (L. 27-36), and (2) combines the intermediate constraints, now complemented with conditions, into corrective constraints (L. 25, 37 and 39). Due to space, we do not show the final corrective constraint for the example of Table I. Detailed exemplification of corrective constraints, including the corrective constraint generated for the example of Table I, can be found in our supplementary material [27].

Instead, in Table II, we show all possible scenarios for composing a corrective constraint from the set of histograms that annotate a given data schema element. In the first scenario (row 1 of Table II), there is no condition involved. The corrective constraint is thus the same as the intermediate constraint built for the unconditional histogram (L. 25 of Alg. 2). In the second scenario (row 2 of Table II), the algorithm first complements with conditions the intermediate constraints of the conditional histograms. The condition of one (conditional) histogram is naturally exclusive of the conditions of others (L. 33-36). This has been illustrated in the second column of Table II. The third scenario (row 3 of Table II) is similar to the second scenario. The only difference is that, since there is no unconditional histogram, we need an extra clause to deal with the situation where none of the conditional histograms

TABLE II
SCENARIOS FOR COMPOSING CORRECTIVE CONSTRAINTS

	Possible annotation scenarios for a data schema element	Shape of the final corrective constraint
1	The element is annotated only by one *unconditional* histogram, U.	$U_{\text{intermediate}}$
2	The element is annotated by one *unconditional* histogram, U, plus one or more *conditional* histograms, Ci. The shape shown is for when there are two conditional histograms.	$U_{\text{intermediate}}$ **or** ($C1_{\text{condition}}$ **and not** $C2_{\text{condition}}$ **and** $C1_{\text{intermediate}}$) **or** (**not** $C1_{\text{condition}}$ **and** $C2_{\text{condition}}$ **and** $C2_{\text{intermediate}}$)
3	The element is annotated only by *conditional* histograms, Ci. The shape shown is for when there are two conditional histograms.	($C1_{\text{condition}}$ **and not** $C2_{\text{condition}}$ **and** $C1_{\text{intermediate}}$) **or** (**not** $C1_{\text{condition}}$ **and** $C2_{\text{condition}}$ **and** $C2_{\text{intermediate}}$) **or** (**not** $C1_{\text{condition}}$ **and not** $C2_{\text{condition}}$)

apply (L. 38-39). This "catch all" clause ensures that the final corrective constraint will not impact an instance model to which none of the conditional histograms should apply.

V. EVALUATION

In this section, we empirically evaluate our synthetic data generator through a realistic case study.

A. Research Questions (RQs)

Our evaluation aims to answer the following RQs:

RQ1: How does the customized OCL solver fare against the baseline OCL solver? As discussed in Section IV-A, we customize a baseline OCL solver [16]. RQ1 compares the customized solver against the baseline across two dimensions: (a) execution time, and (b) success rate, i.e., how often each solver succeeds in constructing a logically valid instance model.

RQ2: Does our synthetic data generator run in practical time? Statistical testing requires representative test data. Achieving representativeness often necessitates a large number of instance models to be built. RQ2 investigates whether our approach can construct a sufficiently large number of instance models within practical time.

RQ3: Can our approach generate data samples that are both valid and statistically representative? RQ3 investigates whether our approach yields data samples suitable for statistical testing. Since the approach enforces the validity constraints of interest over all instance models, data samples generated by the approach always meet the validity requirement. Answering RQ3 therefore boils down to determining how well our data generator meets the representativeness requirement.

The experimental setup for answering these RQs is elaborated in Section V-D alongside our results and discussion.

B. Implementation

Our data generator (http://people.svv.lu/tools/SDG/) has been implemented in Java using the Eclipse Modeling Framework [28]. Excluding comments and third-party libraries, our data generator is approximately 39K lines of code.

C. Case Study Description

Our case study is motivated by an anticipated difficulty that acceptance testing of a public administration IT system in Luxembourg will pose, once the development of the system is completed. For this system, many of the software development

TABLE III
COMPARISON AGAINST THE BASELINE SOLVER (RQ1)

	Baseline Solver	**Customized Solver**
Execution time (per instance model)	Avg = 58.3 sec. Std dev = 17.66	Avg = 17.5 sec. Std dev = 11.33
Success rate (calculated based on 100 attempts)	21%	92%

and testing activities have been commissioned to third-parties. Since the actual data that the system will manipulate is sensitive and of a personal nature, sharing the data with third-parties poses complications. Further, there are gaps in the actual data as well as structural mismatches between the data schema used by the system under development and the data schema in which the historical records have been archived. Due to these issues, our collaborating partners have concluded that the most practical way to ascertain reliability is through testing the system using synthetic test data.

The schema for the core data items manipulated by our case study system was developed with participation from subject-matter experts at our collaborating partners. The resulting schema, expressed as a UML class diagram, has 64 classes, 17 enumerations, 53 associations, 43 generalizations, and 344 attributes. The statistical characteristics of the data items were captured using 15 histograms (e.g., for age and income type), 7 conditional distributions (e.g., age distribution upon the condition that the individuals are pensioners), and 13 distributions of other types (e.g., uniform distribution for the day of the year on which individuals are born).

The validity constraints over the data are expressed using 68 OCL invariants available in our supplementary material [27]. Of these, 26 target avoiding logical anomalies (e.g., children being older than their parents). Of the remaining 42 constraints, 30 are implied by the ranges (upper and lower bounds) of the probabilistic annotations, and the final 12 are multiplicity constraints from the data schema. The constraints include 10 nested if-then-else expressions, 7 occurrences of OCL quantifiers, 23 variable declarations, 107 references to predefined OCL operations, and 212 logical operators.

D. Results and Discussion

In this section, we present our case study results and discuss the RQs. The experiments in this section were conducted on a laptop with a 3GHz dual-core processor and 16GB of memory.

RQ1: To answer RQ1, we attempted to generate 100 valid instance models with both the customized and the baseline solver. In this experiment, we considered only the validity constraints of our case study, without taking representativeness into account. We recall that in contrast to the baseline solver which starts from a randomly-generated instance model, the customized solver is seeded with the output of the data generator presented in Section II. Further, the two solvers differ in their strategy for exploring the search space as discussed in Section IV-A. In Table III, we report the execution time and success rate of the two solvers in the 100 attempts made. We note that different runs of the customized solver were seeded with different and randomly-selected initial instance models. None of these initial instance models were valid.

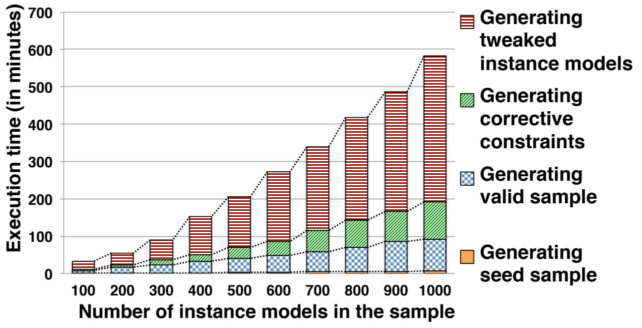


Fig. 5. Execution Times for Generating Valid Data Samples of Different Sizes

As shown in Table III, the customized solver is on average ≈3 times faster than the baseline solver. More importantly, the customized solver is on average ≈4 times more likely to succeed in reaching a valid instance model. Stated otherwise, the customized solver produces a valid instance model in much fewer runs, thus significantly decreasing wasted time and CPU usage when compared to the baseline. The observed improvements are explained mainly by two factors: First, the customized solver has a better starting point (initial instance model) which is easier to make valid. And second, the customized solver has a strategy (explained in Section IV-A) for avoiding entrapment in regions of the search space that do not contain any valid solutions.

The answer to RQ1 is that the customized solver outperforms the baseline by a factor of ≈3 in terms of execution time and by a factor of ≈4 in terms of success rate.

RQ2: To answer RQ2, we measured the average execution time of our data generator for building data samples of different sizes, ranging from 100 to 1000. In the context of our case study, each element in the sample is an instance model that represents a household for the purposes of taxation. For a given data sample size, the data generation process was *repeated five times* to account for random variation.

For this experiment, we configured our data generator as follows: (a) The number of times the solver is invoked over a given instance model in order to create tweaked instance models (parameter nb_attempts of Alg. 1) is set to two, and (b) the margin for comparing relative frequencies (parameter freq_sensitivity of Alg. 1) is set to 0.01. This means that a difference of 1% between a relative frequency in the data sample and the corresponding frequency in the desired characteristics will prompt our data generator to take corrective action.

Average execution times for different sample sizes are shown in Fig. 5. For example, the average execution time (across five runs) for producing a data sample with 500 (valid) instance models is ≈200 minutes. Overall, we generated $(100 + 200 + ... + 1000) \times 5 = 27500$ (valid) instance models. On average, an instance model from this cumulative population has 40 objects, 276 attribute values, and 37 object links. Average instance model size depends on the specific data profile of the system under test.

Fig. 5 further provides a breakdown of the execution times over the different steps of our data generator. The breakdown indicates: First, the time required for creating an initial seed sample is negligible. Second, the generation of corrective constraints (by Alg. 2) is highly scalable with its execution time showing a linear growth trend. Finally, the most computationally-intensive steps are those involving constraint solving (i.e., generating valid sample and generating tweaked instance models in Fig. 5). Constraint solving accounts on average for 85% of the execution time. Despite its complexity, our data generator could produce in less than ten hours a data sample with 1000 instance models (i.e., 1000 test cases). This execution time is practical in our context because, in the worst case, the data can be generated overnight. Indeed, since data profiles are often stable, one can imagine that the test data can be generated early on and well before the testing phase. For systems with more complex data schemas, parallelization can be considered, noting that the solver technology underlying our approach is search-based and easily parallelizable [16].

The answer to RQ2 is that our data generator could produce samples with up to 1000 instance models in less than ten hours. This execution time is practical in our context, since data generation can be performed overnight. For more complex systems, parallelization of search during constraint solving can be considered. Further, test data generation can be initiated well in advance of the testing phase, and as soon as the data profile for the system under test has stabilized.

RQ3: To answer RQ3, we use the same experimental setup and instance models as in RQ2. The basis for our answer is the average distance between the statistical distributions in a given sample and the corresponding distributions specified by the data profile. Note that for a given sample size, we compute average distances based on five runs, as explained in RQ2.

As noted in Section IV-B, we use the Euclidean distance metric for guiding data generation. Euclidean distance is nevertheless not the only metric that one can use for quantifying representativeness. To gain more thorough insights about the representativeness of the data samples generated by our approach, we employ two additional distance metrics, namely Manhattan and Canberra [26]. These additional metrics were selected on the basis of the following criteria: (1) they, alongside Euclidean distance, are among the most commonly-used distance metrics for comparing distributions [26], and (2) robust implementations of the metrics were readily available [29]. These two new distance metrics are interpreted in the same way as Euclidean distance: the closer the distance is to zero, the better aligned a given pair of distributions are. Using these additional metrics in our evaluation helps ensure that our results are not strongly biased toward the specific notion of representativeness induced by Euclidean distance.

Figs. 6(a) – (c) respectively show the representativeness results computed by the Euclidean, Manhattan, and Canberra distance metrics. For each sample size, distances are computed for: (1) the seed (potentially invalid) data sample (2) the initial valid data sample built based on the seed sample, and (3) the final sample returned by our data generator. These distances

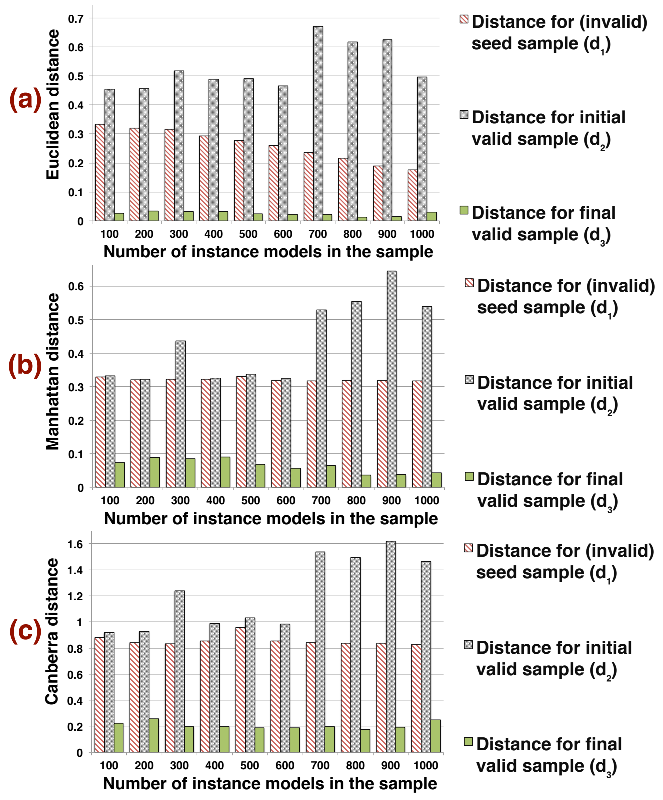


Fig. 6. Distance between Generated Sample and Desired Distributions: (a) Euclidean Distance, (b) Manhattan Distance, and (c) Canberra Distance

are denoted d_1, d_2 and d_3 as shown in Fig. 6. The difference between d_2 and d_1 results from fixing the logical anomalies in the seed sample. The difference between d_3 and d_2 is the improvement induced by the corrective constraints. The difference between d_3 and d_1 indicates the improvement in representativeness brought about by our data generator when compared to the representativeness of the seed sample.

The same trends are observed across the results irrespective of the distance metric used: First, we see that $d_2 > d_1$. This is natural, since we initially attempt to make the seed sample valid without accounting for representativeness. Second, $d_2 \gg d_3$, which means that the generated corrective constraints have been effective at guiding constraint solving toward representativeness. Thirdly, and remarkably, $d_1 > d_3$. The average of $(d_1 - d_3)$ across all data sample sizes is ≈0.2, ≈0.25 and ≈0.6 for the Euclidean, Manhattan and Canberra distance metrics, respectively. This means that our data generator, in addition to producing logically valid samples, has *surpassed* in terms of representativeness the seed sample, which was built exclusively to be representative.

When considering the final data samples, the largest standard deviation observed in distances across the five runs made for each sample size was ≈0.004 (not shown in Fig. 6). This provides confidence that random variation has little influence over the representativeness of the final data samples.

*The answer to RQ3 is that the (final) data samples created by our data generator are **valid**, and at the same time, **surpassing** the state-of-the-art in terms of representativeness.*

E. Threats to Validity

Conclusion and external validity are the most relevant aspects of validity to our case study.

Conclusion validity. As stated in Section V-C, our case study was prompted by a foreseen difficulty in the acceptance testing of a system that is still under development. The unavailability of the final system prevented us from using the data generated by our approach for system-level testing. This leaves the possibility that the system may require test data beyond what was generated. To mitigate this threat, we ensured that the data schema was validated by domain experts. Further, since the system is an operationalization of procedures described in taxation and social security laws, we were able to check our data schema against legal provisions and make sure that no important concepts were overlooked. We thus believe that the likelihood of major omissions in our data schema is low.

External validity. Generalizability is always a concern in case study research, particularly when the results are drawn from a single case. Our evaluation results need to be interpreted with respect to the complexity of our data schema and the associated OCL constraints. Further studies remain essential to determine how our approach will perform on more complex data profiles. This said, our case study system is by any standard a complex data-intensive system. In addition, and as noted earlier, our approach provides two alternatives for further enhancing scalability: (1) to start test data generation long before the testing starts, and (2) to parallelize constraint solving.

VI. Related Work

Usage profiles. In the introduction, we already compared our work with the existing literature on usage profiles. Without repeating what was already said, we make some additional remarks. Existing usage profiles mainly target embedded and web-based systems. The behaviors of these systems typically lend themselves to being modeled as states and transitions (for web-based systems, web pages represent states, and clicks on links and buttons represent transitions [6]). State-machine-like notations such as Markov chains therefore provide a convenient way to build usage profiles for these systems. Our work in contrast focuses on systems whose behavior is driven by data that is interdependent and subject to complex logical constraints. A data schema enhanced with probabilistic information and constraints is a more natural choice for encoding usage profiles in our application context.

Synthetic data generation. The ability to create synthetic data is an integral part of automated test case generation. Since, in practice, it is often infeasible to cover all possible test scenarios, test case generation (and thus the underlying data generation strategy) is typically targeted at optimizing some notion of coverage, e.g., state or path coverage [30]. Metaheuristic search is widely used for generating data to support coverage-based testing [30]. Our data generation strategy relies on search, but rather than attempting to maximize some coverage criterion, we try to achieve statistical representativeness.

In the context of model-based development, data generation has been considered from many angles, including model verification and model-based testing. The most notable tool to mention here is Alloy [31], which provides a specification language based on first-order logic and a SAT-based model finder. Another interesting work strand is UML2CSP [20], where constraint programming is employed for generating instance models that satisfy a given set of OCL constraints. In theory, we could have employed in our approach either Alloy or UML2CSP for constraint solving. Nevertheless, due to the technical limitations already discussed in Section IV-A, most importantly scalability, we opted for a search-based solution.

Aside from the above work, a number of heuristic techniques exist for generating large synthetic data. Notably, Hartmann et al. [32] propose a rule-based technique for generating realistic smart grid instances according to the grid's known topological characteristics. And, Mougenot et al. [33] adopt random sampling for generating large models in linear time. These techniques cannot enforce complex validity constraints over data. The techniques, on their own, are therefore not sufficient for achieving our goals in this paper.

Whole test suite generation. Whole test suite generation builds an entire test suite by simultaneously optimizing multiple fitness functions (e.g., for multiple coverage criteria) [34], [35]. In principle and with appropriate fitness functions defined for validity and representativeness, the problem addressed in this paper can be formulated as whole test suite generation. The realization is however impractical: Whole test suite generation has been applied mainly to unit testing, where the test cases are small. In our context, test cases are much larger and are composed of complex and interdependent data elements. A whole test suite would therefore be prohibitively large for being manipulated by search. Further, our goal is not to optimize validity, but rather to guarantee it while optimizing representativeness. Our approach therefore takes a different route than whole test suite generation. We achieve validity and representativeness separately. Specifically, we start with a representative but invalid test suite. We make this test suite valid, but in the process, reduce its representativeness. At the end, we optimize representativeness without affecting validity.

VII. Conclusion

Focusing on data-intensive systems, we proposed an approach for building synthetic test data. We evaluated the approach over an industrial case study. Our empirical results suggest that our approach can generate within practical time test data that is both statistically representative and logically valid. Meeting these criteria is key for meaningful reliability estimation via statistical testing. For future work, we would like to use the generated data for actual system testing. We further plan to conduct additional case studies to better assess the usefulness and scalability of our data generation approach.

Acknowledgment. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

References

[1] P. Runeson and C. Wohlin, "Statistical usage testing for software reliability control," *Informatica*, vol. 19, no. 2, pp. 195–207, 1995.

[2] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, no. 2, pp. 14–32, 1993.

[3] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 1, pp. 93–106, 1993.

[4] J. H. Poore and C. J. Trammell, "Application of statistical science to testing and evaluating software intensive systems," in *Statistics, Testing, and Defense Acquisition*, M. L. Cohen, D. L. Steffey, and J. E. Rolph, Eds. National Academies Press, 1999, ch. 3.

[5] C. Kallepalli and J. Tian, "Measuring and modeling usage and reliability for statistical web testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 11, pp. 1023–1036, 2001.

[6] P. Tonella and F. Ricca, "Statistical testing of web applications," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 103–127, 2004.

[7] H. L. Guen, R. Marie, and T. Thelin, "Reliability estimation for statistical usage testing using markov chains," in *Proceedings of 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*. IEEE, 2004, pp. 54–65.

[8] S. Herbold, P. Harms, and J. Grabowski, "Combining usage-based and model-based testing for service-oriented architectures in the industrial practice," *International Journal on Software Tools for Technology Transfer (STTT)*, 2016, (in press).

[9] "General Data Protection Regulation (Regulation (EU) 2016/679)," 2016. [Online]. Available: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC

[10] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Fragments and loose associations: Respecting privacy in data publishing," *Proceedings of Very Large Data Bases Endowment (VLDB)*, vol. 3, no. 1, pp. 1370–1381, 2010.

[11] D. Al-Azizy, D. Millard, I. Symeonidis, K. O'Hara, and N. Shadbolt, "A literature survey and classifications on data deanonymisation," in *Proceedings of 10th International Conference on Risks and Security of Internet and Systems (CRiSIS'10)*. Springer, 2015, pp. 36–51.

[12] G. Soltana, N. Sannier, M. Sabetzadeh, and L. Briand, "Model-based simulation of legal policies: Framework, tool support, and validation," *Software & Systems Modeling (SoSyM)*, 2016, (in press).

[13] F. Figari, A. Paulus, and H. Sutherland, "Microsimulation and policy analysis," *Handbook of Income Distribution*, vol. 2, 2014.

[14] G. Soltana, M. Sabetzadeh, and L. Briand, "Model-based simulation of legal requirements: Experience from tax policy simulation," in *Proceedings of 24th IEEE International Requirements Engineering Conference (RE'16)*. IEEE, 2016.

[15] Object Management Group, "Object Constraint Language 2.4 Specification," 2004, http://www.omg.org/spec/OCL/2.4/, last accessed: May 2017.

[16] S. Ali, M. Z. Iqbal, M. Khalid, and A. Arcuri, "Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach," *Empirical Software Engineering (ESE)*, vol. 21, no. 6, pp. 2459–2502, 2016.

[17] Object Management Group, "OMG Unified Modeling Language (UML)," 2015, http://www.omg.org/spec/UML/2.5, last accessed: March 2017.

[18] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software & Systems Modeling (SoSyM)*, vol. 9, no. 1, pp. 69–86, 2010.

[19] A. Cunha, A. Garis, and D. Riesco, "Translating between Alloy specifications and UML class diagrams annotated with OCL," *Software & Systems Modeling (SoSyM)*, vol. 14, no. 1, pp. 5–25, 2015.

[20] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming," *Journal of Systems and Software (JSS)*, vol. 93, pp. 1–23, 2014.

[21] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from OCL constraints with search techniques," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 10, pp. 1376–1402, 2013.

[22] M. P. Krieger and A. Knapp, "Executing underspecified OCL operation contracts with a SAT solver," *Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, vol. 15, pp. 1–16, 2008.

[23] P. Hurley, *A concise introduction to logic*. Nelson Education, 2014.

[24] P. B. Miltersen, J. Radhakrishnan, and I. Wegener, "On converting CNF to DNF," *Theoretical Computer Science*, vol. 347, no. 1-2, pp. 325–335, 2005.

[25] R. K. Hammond and J. E. Bickel, "Discretization methods for continuous probability distributions," in *Wiley Encyclopedia of Operations Research and Management Science*. Wiley, 2015.

[26] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *Mathematical Models and Methods in Applied Sciences*, vol. 1, no. 2, pp. 300–307, 2007.

[27] G. Soltana, M. Sabetzadeh, and L. Briand, "Synthetic data generation for statistical testing: Supplementary material," SnT Centre for Security, Reliability and Trust, University of Luxembourg, Supplementary Material, May 2017, http://people.svv.lu/soltana/ASE17_supp.pdf.

[28] Eclipse Foundation, "EMF: Eclipse Modeling Framework," http://www.eclipse.org/emf, last accessed: May 2017.

[29] Apache Foundation, "Apache commons mathematics library," http://commons.apache.org/proper/commons-math/, last accessed: May 2017.

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

[31] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[32] T. Hartmann, F. Fouquet, J. Klein, Y. Le Traon, A. Pelov, L. Toutain, and T. Ropitault, "Generating realistic smart grid communication topologies based on real-data," in *Proceedings of 5th IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*, 2014, pp. 428–433.

[33] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria, "Uniform random generation of huge metamodel instances," in *Proceedings of 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, 2009, pp. 130–145.

[34] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 2, pp. 276–291, 2013.

[35] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering (ESE)*, vol. 22, no. 2, pp. 852–893, 2017.