A Model-Based Framework for Probabilistic Simulation of Legal Policies

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

***Abstract*—Legal policy simulation is an important decision-support tool in domains such as taxation. The primary goal of legal policy simulation is predicting how changes in the law affect measures of interest, e.g., revenue. Currently, legal policies are simulated via a combination of spreadsheets and software code. This poses a validation challenge both due to complexity reasons and due to legal experts lacking the expertise to understand software code. A further challenge is that representative data for simulation may be unavailable, thus necessitating a data generator. We develop a framework for legal policy simulation that is aimed at addressing these challenges. The framework uses models for specifying both legal policies and the probabilistic characteristics of the underlying population. We devise an automated algorithm for simulation data generation. We evaluate our framework through a case study on Luxembourg's Tax Law.**

***Index Terms*—Legal Policies, Simulation, UML Profiles, Model-Driven Code Generation, Probabilistic Data Generation**

I. Introduction

In legal domains such as taxation and social security, governments need to formulate and implement complex policies to meet a range of objectives, including a balanced budget and equitable distribution of wealth. These policies are reviewed and revised on an ongoing basis to keep them aligned with fiscal, monetary, and social targets at any given time.

Legal policy simulation is a key decision-support tool to predict the impact of proposed legal reforms, and to develop confidence that the reforms will bring about the intended consequences without causing undesirable side effects. In applied economics, this type of simulation falls within the scope of *microsimulation*. Microsimulation encompasses a variety of techniques that apply a set of rules over individual units (e.g., households, physical persons, or firms) to simulate changes [1]. The rules may be deterministic or stochastic, with the simulation results being an estimation of how these rules would work in the real world. For example, in the taxation domain, one may use a sample, say 1000 households from the entire population, to simulate how a set of proposed modifications to the tax law will impact quantities such as due taxes for individual households or at an aggregate level.

Existing legal policy simulation frameworks, e.g., EUROMOD [1] and ASSERT [2], use a combination of spreadsheets and software code written in languages such as C++ for implementing legal policies. Directly using spreadsheets and software code nevertheless complicates the validation of the implemented policies. Particularly, spreadsheets tend to get too complex, making it difficult to check whether the policy implementations match their specifications [3]. The difficulty to validate legal policies is only exacerbated when software code is added to the mix, as legal experts often lack the expertise necessary to understand software code. This validation problem also has implications for software systems, as many legal policies need to be implemented into public administration and eGovernment applications.

A second challenge in legal policy simulation is posed by the absence of complete and accurate simulation data. This could be due to various reasons. For example, in regulated domains such as healthcare and taxation, access to real data is highly restricted; to use real data for simulation, the data may first need to undergo a de-identification process which may in turn reduce the quality and resolution of the data. Another reason is that the data needed for simulation may not have been collected. For example, tax simulation often requires a detailed breakdown of the declared tax deductions at the household level. Such fine-grained data may not have been recorded due to the high associated costs. Finally, when new policies are being introduced, no real data may be available for simulation. Due to these reasons, a simulation data generator is often needed in order to produce artificial (but realistic) data, based on historical aggregate distributions and expert estimates. A manual, hard-coded implementation of such a data generator is costly, and provides little transparency about the data generation process.

Contributions. Motivated by the challenges above, we develop in this paper a model-based framework for the simulation of legal policies. Our work focuses on *procedural* policies. These policies, which are often the primary targets for simulation, provide an explicit process to be followed for compliance. Procedural policies are common in many legal domains such as taxation and social security where the laws and regulations are prescriptive. In this work, we do not address declarative policies, e.g., those concerning privacy, which are typically defined using deontic notions such as permissions and obligations [4].

Our simulation framework leverages our previous work [5], where we developed a UML-based modeling methodology for specifying procedural policies (rules) and evaluated its feasibility and usefulness. We adapt this methodology for use in policy simulation. Building on this adaptation, we develop a model-based technique for automatic generation of simulation data, using an explicit specification of the probabilistic characteristics of the underlying population.

Our work addresses a need observed during our collaboration with the Government of Luxembourg. In particular, the Government needs to manage the risks associated with legal reforms. Policy simulation is one of the key risk assessment tools used in this context. Our proposed framework fully automates, based on models, the generation of the simulation infrastructure. In this sense, the framework can be seen as a specialized form of model-driven code and data generation for policy simulators. While the framework is motivated by policy simulation, we believe that it can be generalized and used for other types of simulation, e.g., system simulation.

Specifically, the contributions of this paper are as follows, with 2) and 3) being the main ones:

1) We augment our previously-developed methodology for policy modeling [5] so as to enable policy simulation.
2) We develop a UML profile to capture the probabilistic characteristics of a population. Our profile supports a variety of probabilistic notions, including probabilistic attributes, multiplicities and specializations, as well as conditional probabilities.
3) We automatically derive a simulation data generator from the population characteristics captured by the above profile. To ensure scalability, the data generator provides a built-in mechanism to narrow data generation to what is relevant for a given set of policy models.

We evaluate our simulation framework using six policies from Luxembourg's Income Tax Law and automatically-generated simulation data with up to 10,000 tax cases. The results suggest that our framework is scalable and that the data produced by our data generator is consistent with known distributions about Luxembourg's population.

Structure. Section II provides an overview of our framework. Sections III through V describe the technical components of the framework. Section VI discusses evaluation. Section VII compares with related work. Section VIII concludes the paper.

II. Simulation Framework Overview

Fig. 1 presents an overview of our framework. In Step 1, *Model legal policies*, we express the policies of interest by interpreting the legal texts describing the policies. This step yields two outputs: First, a *domain model* of the underlying legal context expressed as a UML class diagram, and second, for each policy, a *policy model* describing the realization of the policy using a specialized and restricted form of UML activity diagrams. This step has been already addressed in our previous work [5]. We briefly explain our background work in Section III-A and elaborate, in Section III-B, the extensions we have made to the work in order to support policy simulation.

In Step 2, *Annotate domain model with probabilities*, we enrich the domain model (from Step 1) with probabilistic information to guide simulation data generation. This information may originate from various sources, including expert estimates, and business and census data. The conceptual basis for this step is a UML profile that provides the required expressive power for capturing the probabilistic characteristics of a population. We

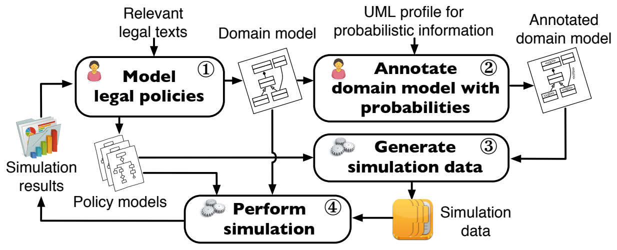

Fig. 1. Simulation Framework Overview

present this UML profile and illustrate it over a real example in Section IV.

In Step 3, *Generate simulation data*, we automatically generate an instance of the domain model based on the probabilistic annotations from Step 2. Our data generation process is discussed in Section V. Finally, in Step 4, *Perform simulation*, we execute the policy models (from Step 1) over the simulation data (from Step 3) to compute the simulation outcomes. Noteworthy details about this step are presented alongside our modeling extensions in Section III-B.

The simulation results are subsequently presented to the user so that they can be checked against expectations. If the results do not meet the expectations, the policy models may be revised and the simulation process repeated. Our framework additionally supports result differencing, meaning that the user can provide an original and a modified set of policies, subject both sets to the same simulation data, and compare the simulation results to quantify the impact. This type of analysis does not add new conceptual elements to our framework and is thus not further discussed in the paper.

III. Legal Policy Models

To enable automated analysis, including simulation, legal policies need to be interpreted and captured in a precise manner. To this end, we developed in our previous work [5] a modeling methodology for specifying (procedural) legal policies. The modeling methodology produces two main outputs, as already described and illustrated in Fig. 1: (1) a domain model, and (2) a set of policy models.

Our main contributions in this paper are adding probabilistic annotations to the domain model and using these annotations for automated simulation data generation (Sections IV and V). Nevertheless, policy models are also a critical component of our framework as these models need to be executed over the simulation data for producing the simulation results. In this section, we first briefly review and illustrate our tailored UML activity diagram notation for policy models. We then present the adaptations we made to support simulation.

A. Legal Policy Modeling Notation

Fig. 2 shows a *simplified* policy model that calculates the tax deduction granted to a taxpayer for disability (invalidity). The stereotypes used in the model are from a previously-developed UML profile [5]. This earlier profile extends activity diagrams with additional semantics for expressing

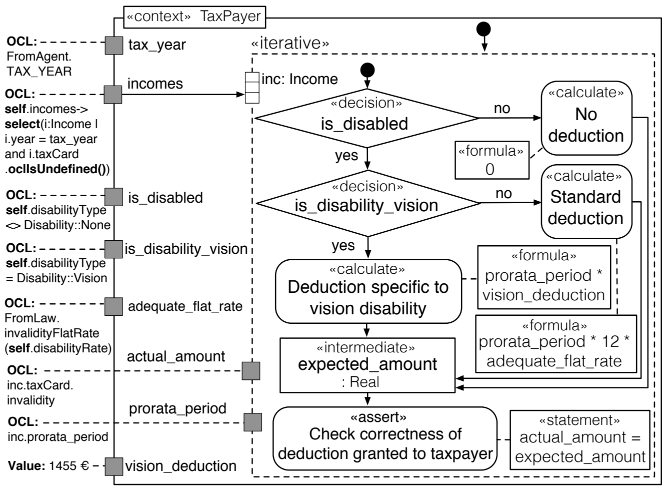
«context» TaxPayer
OCL: FromAgent. TAX_YEAR
tax_year
«iterative»
OCL: self.incomes-> select(i:Income | i.year = tax_year and i.taxCard .oclIsUndefined())
incomes
inc: Income
«decision» is_disabled
no
«calculate» No deduction
yes
«formula» 0
OCL: self.disabilityType <> Disability::None
is_disabled
«decision» is_disability_vision
no
«calculate» Standard deduction
OCL: self.disabilityType = Disability::Vision
is_disability_vision
yes
«calculate» Deduction specific to vision disability
«formula» prorata_period * vision_deduction
OCL: FromLaw. invalidityFlatRate (self.disabilityRate)
adequate_flat_rate
«formula» prorata_period * 12 * adequate_flat_rate
actual_amount
«intermediate» expected_amount : Real
OCL: inc.taxCard. invalidity
prorata_period
«assert» Check correctness of deduction granted to taxpayer
«statement» actual_amount = expected_amount
OCL: inc.prorata_period
Value: 1455 €
vision_deduction

Fig. 2. Policy Model for Calculating Invalidity Tax Deduction (Simplified)

policy models. The model in Fig. 2 envisages three alternative deduction calculations, denoted by the actions with the *«calculate»* stereotype. Each calculation is defined by a corresponding formula (*«formula»*). Based on the taxpayer's eligibility, assessed through decisions (denoted by the *«decision»* stereotype), the appropriate calculation is selected. For instance, if a given taxpayer is not disabled, this policy yields a value of zero; otherwise, another alternative is selected based on disability type (e.g., *Standard deduction*).

The gray boxes in the model of Fig. 2 represent input parameters. For succinctness, we have omitted several details from the model, e.g., the input types and the stereotypes denoting the input origins. Each input is either a value or an OCL query. In the simplest case, an OCL query could point to an attribute from the domain model, e.g., *actual_amount*. An example of a more complex query is for *incomes*, where for a given taxpayer and a tax year, all incomes admitting a tax card are retrieved.

The action annotated with an *«assert»* defines a statement that must hold for policy compliance. This stereotype is not used for simulation purposes, as the main motivation for the stereotype is to define a test oracle and verify whether the output from a system under test complies with a given policy.

Our policy models are automatically transformable to OCL [5], [6]. However, OCL is inadequate for simulation purposes as OCL operations cannot have side effects and are thus unable to make updates. In the next subsection, we describe how our current work adapts policy models for simulation and changes the transformation target language to Java to support operations with side effects.

B. Extending Policy Models to Support Simulation

A simple but important requirement for simulation is to be able to record the simulation results. This requirement cannot be met in a straightforward manner through OCL, due to the language being side-effect-free. To accommodate this requirement, we extend our profile for UML activity diagrams with an additional stereotype, discussed below, and change the target language for model transformation from OCL to Java.

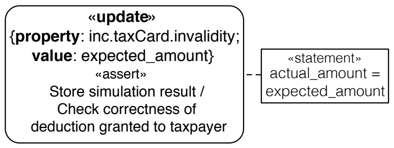

Fig. 3. Modeling Extension to Enable Operations with Side Effects

The new stereotype, *«update»*, makes it possible to update any object (in an instantiation of the domain model), including input parameters. To use the model of Fig. 2 for simulation, we need to record the amount of the disability deduction once it has been calculated. To do so, we attach an *«update»* to the final action in the model of Fig. 2. The modified action is shown in Fig. 3.

With regards to model transformation, we have revised our original transformation rules [6] so that, instead of OCL expressions, the rules will generate Java code with calls to an OCL evaluator. This allows us to handle updates through Java, while still using OCL for querying the domain model. Fig. 4 shows a fragment of the Java code generated for the policy model of Fig. 2, after applying the modifications of Fig. 3. As shown by the code fragment, Java handles loops (L. 8), condition checking (e.g., L. 12), and operations with side effects (e.g., L. 20); whereas OCL handles queries (e.g., L. 5-7). For succinctness and due to the similarity of our revised transformation rules to the original ones, we do not elaborate the transformation of policy models in this paper.

The resulting Java code will be executed over the simulation data produced by the process described in Section V.

```
 1 public static void invalidity(EObject input, String ADName){
 2 OCLInJava.setContext(input);
 3 String OCL = "FromAgent.TAX_YEAR";
 4 int tax_year = OCLInJava.evalInt(input,OCL);
 5 OCL = "self.incomes->select(i:Income | i.year=tax_year and
 6           i.taxCard.oclIsUndefined())";
 7 Collection<EObject> incomes = OCLInJava.evalCollection(input,OCL);
 8 for(EObject inc: incomes){
 9   OCLInJava.newIteration("inc",inc,"incomes",incomes);
10   OCL = "self.disability_type <> Disability_Types::OTHER";
11   boolean is_disabled = OCLInJava.evalBoolean(input,OCL);
12   if(is_disabled == true){
13     OCL = "self.disabilityType = Disability::Vision";
14     boolean is_disability_vision = OCLInJava.evalBoolean(input,OCL);
15     if(is_disability_vision == true){
16       OCL = "inc.prorata_period";
17       double prorata_period = OCLInJava.evalDouble(input,OCL);
18       double vision_deduction = 1455;
19       double expected_amount = prorata_period * vision_deduction;
20       OCLInJava.update(input,"inc.taxCard.invalidity",expected_amount);
```

Fig. 4. Fragment of Generated Java Code for the Policy Model of Fig. 2

IV. Expressing Population Characteristics

In this section, we present our UML profile for capturing the probabilistic characteristics of a population. The profile, which extends UML class diagrams, is shown in Fig. 5. The shaded elements in the figure represent UML metaclasses and the non-shaded elements – the stereotypes of the profile. Below, we explain the stereotypes and illustrate them over a (partial) domain model of Luxembourg's Income Tax Law, shown in Fig. 6. The rectangles with thicker borders in Fig. 6 are constraints (not to be confused with classes). References to Fig. 5 for the stereotypes and Fig. 6 for the examples are not repeated throughout the section.

• ***«probabilistic type»*** extends the Class and EnumerationLiteral metaclasses with relative frequencies. For example, *«probabilistic type»* is applied to the specializations of *Income*, stating that 60% of income types are *Employment*, 20% are *Pension*, and the remaining 20% are *Other*. In this example, the relative frequencies for the specializations of *Income* add up to 1. This means that no residual frequency is left for instantiating *Income* (the parent class). Here, instantiating an *Income* is not possible as *Income* is an abstract class. One could nevertheless have situations where the parent class is also instantiable. In such situations, the relative frequency of a parent class is the residual frequency from its (immediate) subclasses. An example of *«probabilistic type»* applied to enumeration literals can be found in the (truncated) *Disability* enumeration class. Here, we are stating that 90% of the population does not have any disability, while 7.5% has vision problems.

• ***«probabilistic value»*** extends the Property and Constraint metaclasses. Extending the Property metaclass is aimed at augmenting class attributes with probabilistic information. As for the Constraint metaclass, the extension is aimed at providing a container for expressing probabilistic information used by two other stereotypes, *«multiplicity»* and *«dependency»* (discussed later). The *«probabilistic value»* stereotype has an attribute, *precision*, to specify decimal-point precision, and an attribute, *usesOCL*, to state whether any of the attributes of the stereotype's subtypes uses OCL to retrieve a value from an instance of the domain model. A *«probabilistic value»* can be: (1) a *«fixed value»*, (2) *«from chart»*, which could in turn be a bar or a histogram, or (3) *«from distribution»* of a particular *type*, e.g., normal or triangular. The names and values of distribution parameters are specified using the *parameterNames* and *parameterValues* attributes, respectively. The index positions of *parameterNames* match those of the corresponding *parameterValues*. The same goes with the index positions of *items/bins* and *frequencies* in *«from chart»*.

To illustrate, consider the *disabilityRate* attribute of *Taxpayer*. The attribute is drawn from a histogram, stating that 40% of disability rates are between 0 and 0.2, 30% are between

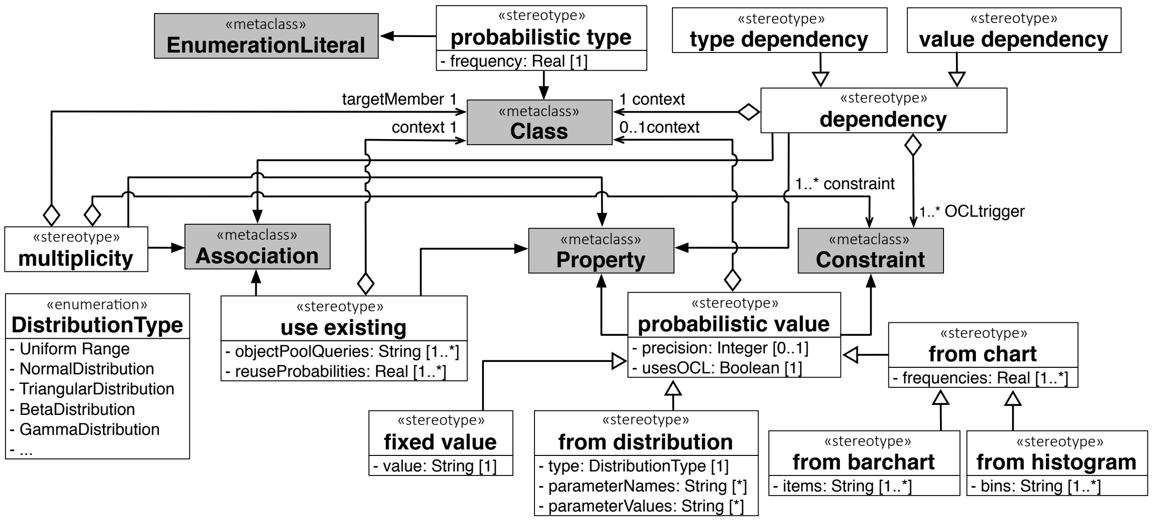
«metaclass» EnumerationLiteral
«stereotype» probabilistic type
- frequency: Real [1]
«stereotype» type dependency
«stereotype» value dependency
targetMember 1
context 1
«metaclass» Class
1 context
0..1context
«stereotype» dependency
1..* constraint
1..* OCLtrigger
«stereotype» multiplicity
«metaclass» Association
«metaclass» Property
«metaclass» Constraint
«enumeration» DistributionType
- Uniform Range
- NormalDistribution
- TriangularDistribution
- BetaDistribution
- GammaDistribution
- ...
«stereotype» use existing
- objectPoolQueries: String [1..*]
- reuseProbabilities: Real [1..*]
«stereotype» probabilistic value
- precision: Integer [0..1]
- usesOCL: Boolean [1]
«stereotype» from chart
- frequencies: Real [1..*]
«stereotype» fixed value
- value: String [1]
«stereotype» from distribution
- type: DistributionType [1]
- parameterNames: String [*]
- parameterValues: String [*]
«stereotype» from barchart
- items: String [1..*]
«stereotype» from histogram
- bins: String [1..*]

Fig. 5. Profile for Expressing Probabilistic Characteristics of a Population

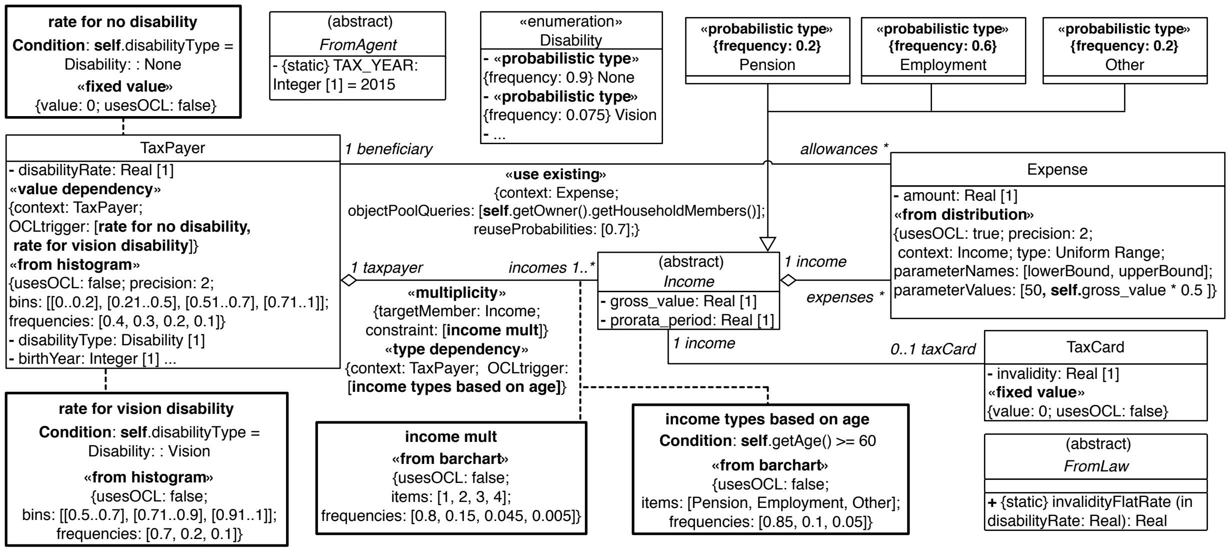
rate for no disability
Condition: self.disabilityType = Disability: : None
«fixed value»
{value: 0; usesOCL: false}
(abstract) FromAgent
- {static} TAX_YEAR: Integer [1] = 2015
«enumeration» Disability
- «probabilistic type» {frequency: 0.9} None
- «probabilistic type» {frequency: 0.075} Vision
- ...
«probabilistic type» {frequency: 0.2} Pension
«probabilistic type» {frequency: 0.6} Employment
«probabilistic type» {frequency: 0.2} Other
TaxPayer
- disabilityRate: Real [1]
«value dependency»
{context: TaxPayer;
OCLtrigger: [rate for no disability, rate for vision disability]}
«from histogram»
{usesOCL: false; precision: 2;
bins: [[0..0.2], [0.21..0.5], [0.51..0.7], [0.71..1]];
frequencies: [0.4, 0.3, 0.2, 0.1]}
- disabilityType: Disability [1]
- birthYear: Integer [1] ...
1 beneficiary
allowances *
«use existing»
{context: Expense;
objectPoolQueries: [self.getOwner().getHouseholdMembers()];
reuseProbabilities: [0.7];}
Expense
- amount: Real [1]
«from distribution»
{usesOCL: true; precision: 2;
context: Income; type: Uniform Range;
parameterNames: [lowerBound, upperBound];
parameterValues: [50, self.gross_value * 0.5]}
1 taxpayer
incomes 1..*
«multiplicity»
{targetMember: Income;
constraint: [income mult]}
«type dependency»
{context: TaxPayer; OCLtrigger: [income types based on age]}
(abstract) Income
- gross_value: Real [1]
- prorata_period: Real [1]
1 income
expenses *
1 income
0..1 taxCard
TaxCard
- invalidity: Real [1]
«fixed value»
{value: 0; usesOCL: false}
(abstract) FromLaw
+ {static} invalidityFlatRate (in disabilityRate: Real): Real
rate for vision disability
Condition: self.disabilityType = Disability: : Vision
«from histogram»
{usesOCL: false;
bins: [[0.5..0.7], [0.71..0.9], [0.91..1]];
frequencies: [0.7, 0.2, 0.1]}
income mult
«from barchart»
{usesOCL: false;
items: [1, 2, 3, 4];
frequencies: [0.8, 0.15, 0.045, 0.005]}
income types based on age
Condition: self.getAge() >= 60
«from barchart»
{usesOCL: false;
items: [Pension, Employment, Other];
frequencies: [0.85, 0.1, 0.05]}

Fig. 6. Partial Domain Model of Luxembourg's Income Tax Law Annotated with Probabilistic Information

0.21 and 0.5, and so on. An example of *«probabilistic value»* specified using an OCL query is the *amount* attribute of *Expense*. This attribute is modeled as a uniform distribution ranging from 50 € up to a maximum of half of the income's gross value for which the expense has been declared.

• ***«multiplicity»*** extends the Association and Property metaclasses. This stereotype is used for attaching probabilistic cardinalities to: (1) association ends (specified as *targetMember*) and (2) attributes defined as collections. To illustrate, consider the association between *TaxPayer* and *Income*. The multiplicity on the *Income* end is expressed as a constraint named *income mult*, which states that the multiplicity is a random variable drawn from a certain bar chart.

• ***«use existing»*** extends the Property and Association metaclasses to enable reusing an object from an existing object pool, as opposed to creating a new one. The object to be reused or created will be assigned to an attribute or to an association end. An application of *«use existing»* involves defining two collections: (1) a collection $q_1, \cdots, q_n$ of OCL queries; (2) a collection $p_1, \cdots, p_n$ of probabilities. Each p_i specifies the probability that an object will be picked from the result-set of q_i. Within the result-set of the q_i picked, all objects have an equal chance of being selected. The residual probability, i.e., $1 - \sum_1^n p_i$, is that of creating a new object.

To illustrate, consider the *beneficiary* end of the association between *TaxPayer* and *Expense*. The *«use existing»* stereotype applied here states that in 70% of situations, the beneficiary is an existing household member; for the remaining 30%, a new *TaxPayer* needs to be created. *«use existing»* envisages collections of queries and probabilities, instead of an individual query and an individual probability, as in UML, one can apply a particular stereotype only once to a model element. In the case of *«use existing»*, one may want to define multiple object pools with their probabilities. For example, the 70% of household members above could have been organized into smaller pools based on the family relationship to the taxpayer (e.g., parent or children), each pool having its own probability.

• ***«dependency»*** is aimed at supporting conditional probabilities. This stereotype is refined into two specialized stereotypes: *«value dependency»* and *«type dependency»*. The former applies to properties only; whereas the latter applies to both properties and associations. In either case, the conditional probabilities are specified by a constraint annotated with *«probabilistic value»*. This constraint is connected to the dependency in question via the *OCLTrigger* aggregation.

To illustrate *«value dependency»*, consider the *disabilityType* and *disabilityRate* attributes of *TaxPayer*. The value of *disabilityRate* is influenced by *disabilityType*. Specifically, if the taxpayer has no disability, then *disabilityRate* is zero. If *disabilityType* is vision, then the distribution of *disabilityRate* follows the histogram given in the constraint named, *rate for vision disability*. Note that disability types other than vision are handled by the generic histogram attached to the *disabilityRate* attribute of *TaxPayer*. The condition under which a particular *«dependency»* applies is provided as part of the constraint that defines the conditional probability. For example, the condition associated with *rate for vision disability* is the following OCL expression: **self**.disabilityType = Disability::Vision.

As for *«type dependency»*, the same principles as above apply. The distinction is that this stereotype influences the choice of the object that fills an association end, rather than the choice of the value for an attribute. To illustrate, consider the association between *TaxPayer* and *Income*. The *«type dependency»* stereotype attached to this association conditions the type of income upon the taxpayer's age. Specifically, for a taxpayer older than 60, *Income* is more likely to be a *Pension* (85%) than an *Employment* (10%) or *Other* (5%).

• ***Consistency constraints:*** Certain consistency constraints must be met for a sound application of the profile. Notably, these constraints include: (1) Mutually-exclusive application of certain stereotypes, e.g., *«fixed value»* and *«from histogram»*; (2) Well-formedness of the the probabilistic information, e.g., sum of probabilities not exceeding one, and correct naming of distribution parameters; and (3) Information completeness, e.g., ensuring that a context is provided when OCL is used in stereotype attributes. These constraints are specified at the level of the profile using OCL, providing instant feedback to the modeler when a constraint is violated.

V. Simulation Data Generation

In this section, we describe the process for automated generation of simulation data (Step 3 of the framework in Fig. 1). An overview of this process is shown in Fig. 7. The inputs to the process are: a domain model annotated with the profile of Section IV and the set of policy models to simulate. The process has four steps, detailed in Sections V-A through V-D. We discuss the practical considerations and limitations of the process in Section V-E.

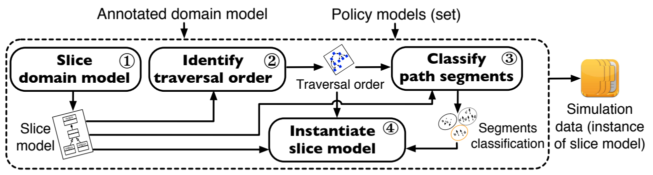

Fig. 7. Overview of Simulation Data Generation

A. Domain Model Slicing

In Step 1 of the process in Fig. 7, *Slice domain model*, we extract a *slice model* containing the domain model elements relevant to the input policy models. This step is aimed at narrowing data generation to what is necessary for simulating the input policy models, and thus improving scalability.

The slice model is built as follows. First, all the OCL expressions in the input policy model(s) are extracted. These expressions are parsed with each element (class, attribute, association) referenced in the expressions added to the slice model. Next, all the elements in the (current) slice model are inspected and the stereotypes applied to them retrieved. The OCL expressions in the retrieved stereotypes are recursively parsed, with each recursion adding to the slice any newly-encountered element. The recursion stops when no new elements are found.

In Fig. 8(a), we show an example slice model, obtained from the domain model of Fig. 6 specifically for simulating the policy model of Fig. 2. Among other elements, the *Expense* class has been excluded from the slice because, to simulate the policy model of Fig. 2, we do not require instances of *Expense*. To avoid clutter in Fig. 8(a), we have not shown the constraints. For the policy model in Fig. 2, all the constraints in the domain model of Fig. 6 are part of the slice. The slice model of Fig. 8(a) also includes three abstract classes, namely *Income*, *FromLaw*, and *FromAgent*. Obviously, these abstract classes will not be instantiated during data generation (Step 4). Nevertheless, these classes are necessary for OCL evaluation and may further play a role in determining the order of object instantiations. We describe how we determine this order next.

B. Identifying a Traversal Order

In Step 2 of the process in Fig. 7, *Identify traversal order*, we compute a total ordering of the classes in the slice model, and for each such class, a total ordering of its attributes. These orderings are used later (in Step 4) to ensure that any model element m is instantiated after all model elements upon which m depends. An element m depends on an element m' if some OCL expression in a stereotype attached to m directly or indirectly references m'.

The orderings are computed via *topological sorting* [7] of a class-level Dependency Graph (DG), and for each class, of an attribute-level DG. The class-level DG is a directed graph whose nodes are the classes of the slice model and whose edges are the *inverted* dependencies, which we call *precedences*, between these classes. More precisely, there is a precedence edge from class C_i to class C_j if C_j depends on C_i, thus requiring that the instantiation of C_i should precede that of C_j. Further, there will be edges from C_i to *all descendants* of C_j as per the generalization hierarchy of the slice model. An attribute-level DG is a graph where the nodes are attributes and the edges are inverted attribute dependencies. Note that the above consideration about descendants is only for classes and does not apply to attributes.

In Fig. 8(b), we illustrate DGs and topological sorting over the slice model of Fig. 8(a). The upper part of Fig. 8(b) is the class-level DG, and the lower part – the attribute-level DG for the *TaxPayer* class. Each of the other classes in the slice has its own attribute-level DG (not shown). All the edges in the class-level DG are induced by the *«type dependency»* stereotype that is attached to the association between *TaxPayer* and *Income* (Fig. 6), specifically by the OCL constraint named *income types based on age*. Since the instantiation of *TaxPayer* should precede that of *Income*, there are precedence edges from *TaxPayer* to all *Income* subclasses as well. The numbers in the DGs of Fig. 8(b) denote one possible total ordering for the respective DGs. Computing these orderings is linear in the size of the DGs [7] and thus inexpensive.

If the class-level or any of the attribute-level DGs are cyclic, topological sorting will fail, indicating that the stereotypes of the slice model are causing cyclic dependencies. In such situations, the cyclic dependencies are reported to the analyst and need to be resolved before data generation can proceed.

The orderings computed in this step ensure that the data generation process will not encounter an uninstantiated object or an unassigned value at the time the object or value is needed. Nevertheless, these orderings do not guarantee that the data generation process will not fall into an infinite loop caused by cyclic association paths in the slice model. In the next step, we describe our strategy to avoid such infinite loops.

C. Classifying Path Segments

To instantiate the slice model, we need to traverse its associations. Traversal is directional, thus necessitating that we keep track of the direction in which each association is traversed. We use the term *segment* to refer to an association being traversed in a certain direction. For example, the association between *TaxPayer* and *Income* has two segments: one from *TaxPayer* to *Income*, and the other from *Income* to *TaxPayer*.

In Step 3 of the process in Fig. 7, *Classify path segments*, the segments of the slice model are classified as *Safe*, *PotentiallyUnsafe*, or *Excluded*. The resulting classification will be used in Step 4 to guide the instantiation. The classification is done via a depth-first search of the segments in the slice model. The search starts from a root class. When there is only one policy model to simulate, this root is the (OCL) context class of that policy. For example, for the slice model of Fig. 2, the root would be *TaxPayer*. When simulation involves multiple policy models, we pick as root the context class from which all other context classes can be reached via aggregations. For example, if the model of Fig. 2 is to be simulated alongside another policy model whose context is *Expense*, the root would still be *TaxPayer* as *Expense* is reachable from *TaxPayer* through aggregations. If no such root class can be found, a unifying interface class has to be defined and realized by the context classes. This interface class will then be designated as the root.

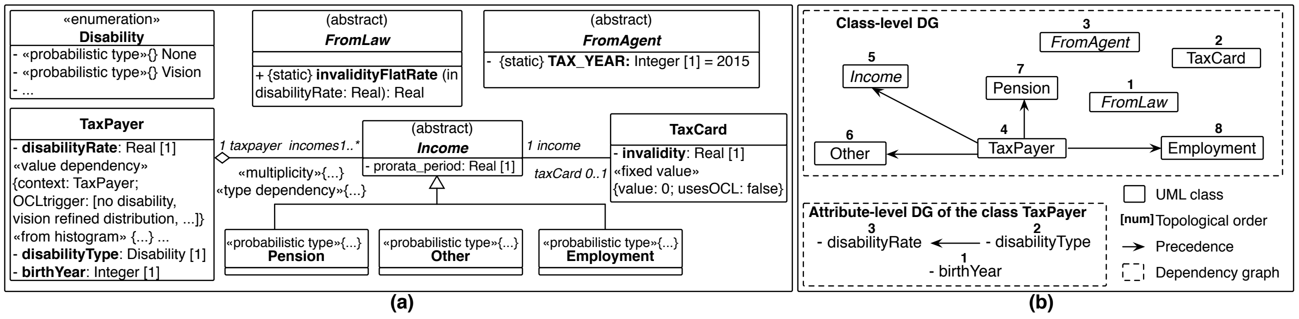

Fig. 8. (a) Excerpt of Slice Model for Simulating Policy Model of Fig. 2, (b) Topological Sorting of Elements in (a)

Given a root class, segment classification is performed as follows: We sort the outgoing segments from the current class (starting with root) based on the indices of the classes at the target ends of the segments. We then recursively traverse the segments in ascending order of the indices. The indices come from the ordering of classes computed in Step 2. For example, the index for *TaxPayer* is 4, as shown in Fig. 8(b). A segment is *Safe* if it reaches a class that is visited for the first time. A segment is *PotentiallyUnsafe* if it reaches a class that has been already visited. A segment going in the opposite direction of a *Safe* or a *PotentiallyUnsafe* segment is *Excluded*.

That above exploration is further extended to attributes typed by some class of the slice model, as assigning a value to such attributes amounts to instantiating a class. For a given class, the traversal order of attributes is determined by the attribute ordering for that class, as computed in Step 2.

To illustrate, consider the slice model of Fig. 8(a). Starting from the root class, *TaxPayer*, the outgoing segments, *TaxPayer→Income* and *Income→TaxCard*, are classified as *Safe*; and the opposite segments, *Income→TaxPayer* and *TaxCard→Income*, as *Excluded*. In Fig. 8(a), there is no *PotentiallyUnsafe* segment as there is no cyclic association path in the slice model. For the sake of argument, had there been an association between *TaxCard* and *TaxPayer*, the segment *TaxCard→TaxPayer* would have been *PotentiallyUnsafe*.

In the next step, we use the segment classification to ensure that simulation data generation terminates.

D. Instantiating the Slice Model

The last step of the process of Fig. 7, *Instantiate slice model*, generates the simulation data. This data is generated by the recursive algorithm of Alg. 1, named SDG. SDG takes as input: (1) the slice model from Step 1, (2) a class to instantiate, (3) the orderings computed in Step 2, (4) the path segment classification from Step 3, and (5) the last traversed segment or attribute of the slice model. The algorithm is initially called over the root class discussed in Step 3 with the last traversed segment or attribute being *null*. The number of executions of SDG over the root class is a user-customizable parameter (say 10,000). SDG has four main parts, explained below.

(1) Class selection and instantiation (L. 1–14). If the *«use existing»* stereotype is present, SDG attempts to return an object from already-existing ones (L. 2-4). If this fails, the input class, C in Alg. 1, has to be instantiated. To do so, SDG selects and instantiates a *non-abstract* class from the following set: {C}∪{all descendants of C}. The selection is based on the *«type dependency»* and *«probabilistic type»* stereotypes attached to C. If these stereotypes are absent or fail to yield a specific class, a random (non-abstract) class from the above set is selected and instantiated (L. 11).

(2) Attribute value assignment (L. 15–28). C's attributes are assigned values based on C's attribute-level ordering from Step 2 (L. 15). Values for primitive attributes are generated by processing *«value dependency»* and *«probabilistic value»*, if either stereotype is present (L. 16-19). If a primitive attribute is unassigned after this processing, a random value is assigned to it (L. 21). For an attribute typed by a class from the slice model, we determine, based on the attribute's multiplicity and any attached *«multiplicity»* stereotype, the required number of objects and recursively create these objects (L. 23-28).

(3) Segment traversal (L. 29–41). For each outgoing (association) segment from C, the required number of objects is determined and the objects are created similarly to non-primitive attributes described above (L. 31-33). The traversal exercises (based on the ordering of classes): (1) all *Safe* segments, and (2) any *PotentiallyUnsafe* segment which has not been already traversed at that specific recursion depth (L. 37-38). *Excluded* segments are ignored during traversal. Handling *PotentiallyUnsafe* and *Excluded* segments in the above-described manner avoids the possibility of infinite recursions. The instantiation process for traversed segments is recursive (L. 39-41).

(4) Handling Excluded segment multiplicities (L. 42–49). Since the algorithm traverses the associations in one direction, the multiplicities of *Excluded* segments need separate treatment. The algorithm attempts to satisfy these multiplicities by: (1) randomly selecting an appropriate number of objects (of the desired type) from the pool of existing objects, (2) cloning the selected objects and all related objects, and (3) updating the association underlying the *Excluded* segment in question to avoid the violation of multiplicity constraints (L. 47-49).

E. Practical Considerations and Limitations

Our simulation data generation strategy is aimed at producing a *large* instance model (i.e., with *thousands* of objects) while respecting the probabilistic characteristics of the underlying population. The strategy was prompted by the scalability challenge that we faced when attempting to use constraint solving for simulation data generation. In particular, we observed that, in our context, current constraint solving tools, e.g., Alloy [8] and UML2CSP [9], could generate, within reasonable time, only small instance models. These tools further lack means for data generation based on probabilistic characteristics.

As we will argue in Section VI, our data generation strategy meets the above scalability requirement. However, the strategy has limitations: (1) As noted in Section V-B, the strategy works only when cyclic OCL dependencies between classes are absent. (2) The strategy guarantees the satisfaction of multiplicity constraints only in the direction of the traversal. Multiplicity constraints in the opposite direction may not be satisfied if appropriate objects cannot be found in the already-existing object pool. Further, to avoid infinite loops, the strategy traverses cyclic association paths only once. Consequently, multiplicities on cyclic associations paths may be left unsatisfied and further unsatisfiable multiplicity constraints will go undetected. (3) The strategy does not guarantee that constraints other than those specified in our profile will be satisfied.

VI. Tool Support and Evaluation

In this section, we describe the implementation of our simulation framework and report on a case study where we apply the framework to Luxembourg's Income Tax Law.

Alg. 1: Simulation Data Generator (**SDG**)

```
Inputs : (1) a slice model $\mathcal{S}$; (2) a class $C \in \mathcal{S}$ to instantiate; (3) the orderings,
         $\mathcal{O}$, from Step 2; (4) path segment classifications, $\mathcal{P}$, from Step 3; and
         (5) the last traversed segment or attribute, source $\in \mathcal{S}$ (initially null)
Output : an instance of class C

Let res be the instance to generate (initially null)
if (source is not null) then
    res ← Attempt «use existing» of source (if the stereotype is present)
    if (res is not null) then return res
chosen ← null /* chosen will be set to either C or some descendant thereof */
if (source is not null) then
    chosen ← Attempt «type dependency» of source
if (chosen is null) then
    if C's immediate subclasses have «probabilistic type» then
        chosen ← Attempt «probabilistic type» from C
    else chosen ← Randomly pick, from C and all descendants, a non-abstract class
if (chosen is null) then return null
else
    res ← Instantiate (chosen)
    foreach (att ∈ SortAttributesByOrder ($\mathcal{O}$, chosen)) do
        if (att is not typed by some class of $\mathcal{S}$) then
            $att$ ← Attempt «value dependency» of att
            if (att is not defined) then
                $att$ ← Attempt «probabilistic value» of att
            if (att is not defined) then
                att ← a random value
        else
            mult ← Attempt «multiplicity» of att
            if (mult is null) then mult ← random value from multiplicity range of att
            Let att_objects be an (initially empty) set of instances
            for ($i \leftarrow 0;\ i <$ mult) do
                att_objects.add (SDG ($\mathcal{S}$, typeOf (att), $\mathcal{O}$, $\mathcal{P}$, att))
            att ← att_objects
    Let paths be the Safe and PotentiallyUnsafe outgoing segments from chosen
    foreach (seg ∈ SortSegmentsByOrder (paths, $\mathcal{O}$)) do
        nextC ← target class of seg
        mult ← Attempt «multiplicity» of seg
        if (mult is null) then mult ← random number from multiplicity range of seg
        Let $\text{objects}_1$ and $\text{objects}_2$ be two (initially empty) sets of instances
        for ($i \leftarrow 0;\ i <$ mult) do
            $\mathcal{P}' \leftarrow \mathcal{P}$
            if (seg is PotentiallyUnsafe in $\mathcal{P}$) then
                Switch seg from PotentiallyUnsafe to Excluded in $\mathcal{P}'$
            $\text{objects}_1$.add (SDG ($\mathcal{S}$, nextC, $\mathcal{O}$, $\mathcal{P}'$, seg))
            Let association be the underlying association of seg
            res.setLinks (association, $\text{objects}_1$)
            if (minimal multiplicity of seg's opposite segment > 1) then
                op_mult ← Attempt «multiplicity» of seg's opposite
                if (op_mult is null) then op_mult←rand. number from mult. range of seg's opposite
                for ($j \leftarrow 0;\ j <$ (op_mult $- 1$)) do
                    Let clone be a deep clone of a randomly-picked instance from the
                    object pool having the same type as the target class of seg's opposite
                    segment (clone $\neq \text{objects}_1$.last () and clone $\notin \text{objects}_2$)
                    clone.removeRandomLink (association)
                    $\text{objects}_2$.add (clone)
                $\text{objects}_1$.last ().setLinks (association, $\text{objects}_2$)
    return res
```

A. Implementation

The manual steps (Steps 1 and 2) in the framework of Fig. 1 can be done using any modeling environment that supports UML and profiles, e.g., Papyrus (eclipse.org/papyrus/). The implementation for Steps 3 and 4 of the framework is based on the Eclipse Modeling Framework (eclipse.org/modeling/emf/). We use Acceleo (eclipse.org/acceleo/) for deriving the Java simulation code from legal policies. To evaluate and parse OCL expressions, we use EclipseOCL (eclipse.org/modeling/). For graph analyses, including topological sorting and cycle detection, we use JGraphT (jgrapht.org). And, for generating random values based on given probability distributions, we use Apache Commons Mathematics Library (commons.apache.org). Statistical tools such as R (r-project.org) would provide an alternative to Apache Commons Mathematics Library, but not to our data generator (Alg. 1). Without additional implementation, these tools are unable to instantiate object-oriented models as they do not provide a mechanism to handle the instantiation order and the interdependencies between model elements (see Section V). Our implementation is approximately 11K lines of code, excluding comments, the third-party libraries above, and the automatically-generated simulation code.

B. Case Study

We investigate, through a case study on Luxembourg's Income Tax Law, the following Research Questions (RQs):

RQ1: Do data generation and simulation run in reasonable time? One should be able to generate large amounts of data and run the policy models of interest over this data reasonably quickly. The goal of RQ1 is to determine whether our data generator and simulator have reasonable executions times.

RQ2: Does our data generator produce data that is consistent with the specified characteristics of the population? A basic and yet important requirement for our data generator is that the generated data should be aligned with what is specified via the profile. RQ2 aims to provide confidence that our data generation strategy, including the specific choices we have made for model traversal and for handling dependencies and multiplicities, satisfies the above requirement.

RQ3: Are the results of different data generation runs consistent? Our data generator is probabilistic. While multiple runs of the generator will inevitably produce different results due to random variation, one would expect some level of consistency across the data produced by different runs. If the results of different runs are inconsistent, one can have little confidence in the simulation outcomes being meaningful. RQ3 aims to measure the level of consistency between data generated by different runs of our data generator.

For our case study, we consider six representative policies from Luxembourg's Income Tax Law (circa 2013). Two of these policies concern tax credits and the other four – tax deductions. The credits are for salaried workers (CIS) and pensioners (CIP); the deductions are for commuting expenses (FD), invalidity (ID), permanent expenses (PE), and long-term debts (LD). A simplified version of ID was shown in Fig. 2. Initial versions of these six policy models and the domain model supporting these policies (as well as other policies not considered here) were built in our previous work [5].

The six policy models in our study have an average of 35 elements, an element being an input, output, decision, action, flow, intermediate variable, expansion region, or constraint. The largest model is FD (60 elements); the smallest is PE (25 elements). The domain model has 64 classes, 17 enumerations, 53 associations, 43 generalizations, and 344 attributes.

These existing models were enhanced to support simulation and validated with (already-trained) legal experts in a series of meetings, totaling ≈12 hours. The probabilistic information for annotating the domain model was derived from publicly-available census data provided by STATEC (statistiques.public.lu/). Specifically, from this data, we extracted information about 13 quantities including, among others, age, income and income type. The stereotype annotations in the partial domain model of Fig. 6 are based on the extracted information, noting that the actual numerical values were rounded up or down to avoid cluttering the figure with long decimal-point values.

To answer the RQs, we ran the simulator (automatically derived from the six policy models) over simulation data (automatically generated by Alg. 1). We discuss the results below. All the results were obtained on a computer with a 3.0GHz dual-core processor and 16GB of memory.

RQ1. The execution times of the data generator and the simulator are influenced mainly by two factors: the size of the data to produce –here, the number of tax cases– and the number and complexity of the policy models to simulate. Note that the data generator instantiates only the slice model that is relevant to the policies of interest and not the entire domain model. This is why the selected policy models have an influence on the the execution time of the data generator.

To answer RQ1, we measured the execution times of the data generator and the simulator with respect to the above two factors. Specifically, we picked a random permutation of the six policies –ID, CIS, PE, FD, LD, CIP– and generated 10,000 tax cases, in increments of 1,000, first for ID, then for ID combined with CIS, and so on. When all the six policies are considered, a generated tax case has an average of ≈24 objects. We then ran the simulation for different numbers of tax cases and the different combinations of policy models considered. Since the data generation process is probabilistic, we ran the process (and the simulations) five times. In Figs. 9(a) and (b), we show the execution times (average of the five runs) for the data generator and for the simulator, respectively.

As suggested by Fig. 9(a), the execution time of the data generator increases linearly with the number of tax cases. We further observed a linear increase in the execution time of the data generator as the size of the slice model increased. This is indicated by the proportional increase in the slope of the curves in Fig. 9(a). Specifically, the slice models for the six policy sets used in our evaluation, i.e., (1) ID, (2) ID + CIS, . . . , (6) ID + CIS + PE + FD + LD + CIP, covered approximately 4%, 5%, 7%, 13%, 20%, and 22% of the domain model, respectively. We note that as more policies are included, the slice model will eventually saturate, as the largest possible slice model is the full domain model.

With regards to simulation, the execution times partly depend on the complexity of the workflows in the underlying policies (e.g., the nesting of loops), and partly on the OCL queries that supply the input parameters to the policies. The latter factor deserves attention when simulation is run over a large instance model. Particularly, OCL queries containing iterative operations may take longer to run as the instance model grows. The non-linear complexity seen in the fifth and sixth curves (from the bottom) in Fig. 9(b) is due to an OCL allInstances() call in LD, which can be avoided by changing the domain model and optimizing the query. This would result in the fifth and sixth curves to follow the same linear trend seen in the other curves. Since the measured execution times are already small and reasonable, such optimization is warranted only when the execution times need to be further reduced.

As suggested by Figs. 9(a) and (b), our data generator and simulator are highly scalable: Generating 10,000 tax cases covering all six policies took ≈30 minutes. Simulating the policies over 10,000 tax cases took ≈24 minutes.

RQ2. To answer RQ2, we compare information from STATEC for age, income and income type, all represented as histograms, against histograms built over generated data of various sizes. Similar to RQ1, we ran the data generator five times and took the average for analysis. Among alternative ways to compare histograms, we use Euclidean distance which is widely used for this purpose [10]. Fig. 9(c) presents Euclidean distances for the age, income, and income type histograms as well as the Euclidean distance for the normalized aggregation of the three. As indicated by the figure, the Euclidean distance for the aggregation falls below 0.05 for 2000 or more tax cases produced by our data generator. This suggests a close alignment between the generated data and Luxembourg's real population across the three criteria considered.

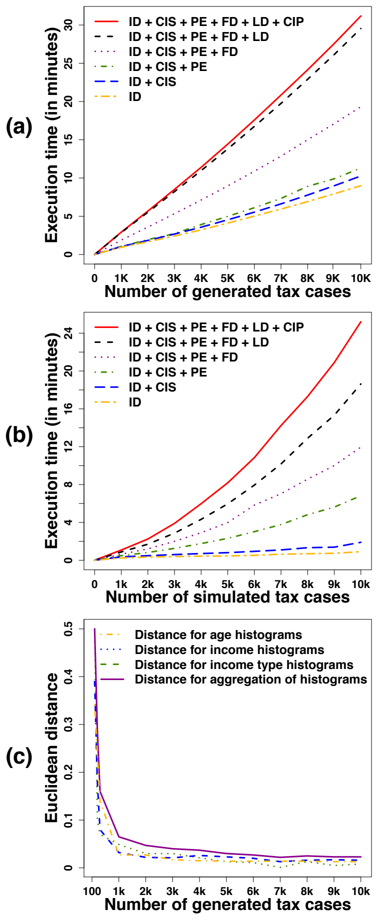

Fig. 9. Execution Times for Data Generation (a) & Simulation (b); Euclidean Distances between Generated Data & Real Population Characteristics (c)

The above analysis provides confidence about the quality of the data produced by our data generator. The analysis further establishes a lower-bound for the number of tax cases to generate (2,000) to reach a high level of data quality.

RQ3. We answer RQ3 using the Kolmogorov-Smirnov (KS) test [11], a non-parametric test to compare the cumulative frequency distributions of two samples and determine whether they are likely to be derived from the same population.

This test yields two values: (1) D, representing the maximum distance observed between the cumulative distributions of the samples. The smaller D is, the more likely the samples are to be derived from the same population; and (2) p-value, representing the probability that the two cumulative sample distributions would be as far apart as observed if they were derived from the same population. If the p-value is small (< 0.05), one can conclude that the two samples are from different populations.

To check the consistency of data produced across different runs of our data generator, we ran the generator five times,

TABLE I
PAIRWISE KOLMOGOROV-SMIRNOV (KS) TEST APPLIED TO FIVE SAMPLES ($P_1, \cdots, P_5$) OF 5000 TAX CASES

		Age				**Income**				**Income type**			
		P_2	P_3	P_4	P_5	P_2	P_3	P_4	P_5	P_2	P_3	P_4	P_5
P_1	D	0.015	0.009	0.012	0.009	0.013	0.015	0.012	0.016	0.007	0.004	0.011	0.012
	p-value	0.63	0.98	0.8	0.98	0.7	0.62	0.79	0.44	0.99	1	0.87	0.93
P_2	D	-	0.011	0.011	0.017	-	0.015	0.013	0.012	-	0.007	0.004	0.002
	p-value	-	0.86	0.85	0.41	-	0.61	0.7	0.9	-	0.99	1	1
P_3	D	-	-	0.016	0.011	-	-	0.016	0.012	-	-	0.011	0.005
	p-value	-	-	0.51	0.91	-	-	0.5	0.83	-	-	0.89	1
P_4	D	-	-	-	0.015	-	-	-	0.017	-	-	-	0.005
	p-value	-	-	-	0.53	-	-	-	0.39	-	-	-	1

each time generating a sample of 5000 tax cases. We then performed pairwise KS tests for the age, income, and income type information from the samples. Table I shows the results, with $P_1, \cdots, P_5$ denoting the samples from the different runs. As shown by the table, the maximum D is 0.017 and the minimum p-value is 0.39. The KS tests thus give no counter-evidence for the samples being from different populations.

The results in Table I provide confidence that our data generator yields consistent data across different runs.

VII. RELATED WORK

Legal policy simulation. As we discussed in the introduction, there are a number of legal policy simulation tools in the area of applied economics, e.g., [1], [2]. These tools do not adequately address the expertise gap between legal experts and system analysts. Our framework takes a step towards addressing this gap by providing a more abstract way to specify legal policies and the simulation data generation process, so that the resulting specifications would be palatable to legal experts with a reasonable amount of training.

Model-based instance generation. Automated instantiation of (meta-)models is useful in many situations, e.g., during testing [12] and system configuration [13]. Several instance generation approaches are based on exhaustive search, using tools such as Alloy [8] and UML2CSP [9]. Model instances generated by Alloy are typically counter-examples showing the violation of some logical property. As for UML2CSP, the main motivation is to generate a valid instance as a way to assess the correctness and satisfiability of the underlying model. Approaches based on exhaustive search, as we noted in Section V-E, do not scale well in our application context.

A second class of instance generation approaches rely on non-exhaustive techniques, e.g., predefined generation patterns [14], [15], metaheuristic search [16], mutation analysis [17], and model cloning [18]. Among these, metaheuristic search shows the most promise in our context. Nevertheless, further research is necessary to address the scalability challenge and generate large quantities of data using metaheuristic search.

VIII. CONCLUSION

We proposed a model-based framework for legal policy simulation. The framework includes an automated data generator. The key enabler for the generator is a UML profile for capturing the probabilistic characteristics of a given population. Using legal policies from the tax domain, we conducted an empirical evaluation showing that our framework is scalable, and produces consistent data that is aligned with census information.

In the future, we plan to investigate whether our data generation process can be enhanced with constraint solving capabilities via metaheuristic search in order to support additional constraints. We further plan to conduct a more detailed evaluation to investigate the overall accuracy of our simulation framework. This requires the generated data and the simulation results to be validated with legal experts and further against complex correlations in census information.

Acknowledgment. We thank members of Luxembourg's Inland Revenue Office (ACD) and National Centre for Information Technologies (CTIE), particularly T. Prommenschenkel, L. Balmer, and M. Blau for sharing their valuable time and insights with us. Financial support was provided by CTIE and FNR under grants FNR/P10/03 and FNR9242479.

REFERENCES

[1] F. Figari, A. Paulus, and H. Sutherland, "Microsimulation and policy analysis," in *Handbook of Income Distribution*. Elsevier, 2015, vol. 2.

[2] S. Hohls, "How to support (political) decisions?" in *Electronic Government*. Springer, 2013.

[3] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *ICSE'12*, 2012.

[4] D. Ruiter, *Institutional Legal Facts: Legal Powers and Their Effects*. Kluwer Academic Publishers, 1993.

[5] G. Soltana, E. Fourneret, M. Adedjouma, M. Sabetzadeh, and L. Briand, "Using UML for modeling procedural legal rules: Approach and a study of Luxembourg's Tax Law," in *MODELS'14*, 2014.

[6] G. Soltana et al., "Using UML for modeling legal rules: Supplementary material," University of Luxembourg, Tech. Rep., 2014, http://people.svv.lu/soltana/Models14.pdf.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[8] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[9] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming," *JSS*, vol. 93, 2014.

[10] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *Mathematical Models and Methods in Applied Sciences*, vol. 1, 2007.

[11] G. W. Corder and D. Foreman, *Nonparametric Statistics: A Step-by-Step Approach*. John Wiley & Sons, 2014.

[12] M. Iqbal, A. Arcuri, and L. Briand, "Environment modeling and simulation for automated testing of soft real-time embedded software," *SoSyM*, vol. 14, 2015.

[13] R. Behjati, S. Nejati, and L. Briand, "Architecture-level configuration of large-scale embedded software systems," *ACM TOSEM*, vol. 23, 2014.

[14] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *SoSyM*, vol. 4, 2005.

[15] T. Hartmann et al., "Generating realistic smart grid communication topologies based on real-data," in *SmartGridComm'14*, 2014.

[16] S. Ali, M. Iqbal, A. Arcuri, and L. Briand, "Generating test data from OCL constraints with search techniques," *IEEE TSE*, vol. 39, 2013.

[17] D. Di Nardo, F. Pastore, and L. Briand, "Generating complex and faulty test data through model-based mutation analysis," in *ICST'15*, 2015.

[18] E. Bousse, B. Combemale, and B. Baudry, "Scalable armies of model clones through data sharing," in *MODELS'14*, 2014.