Noname manuscript No.
(will be inserted by the editor)

Model-Based Simulation of Legal Policies: Framework, Tool Support, and Validation

Ghanem Soltana · Nicolas Sannier · Mehrdad Sabetzadeh · Lionel C. Briand

the date of receipt and acceptance should be inserted later

Abstract Simulation of legal policies 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. Legal policy simulation is currently implemented using a combination of spreadsheets and software code. Such a direct implementation poses a validation challenge. In particular, legal experts often lack the necessary software background to review complex spreadsheets and code. Consequently, these experts currently have no reliable means to check the correctness of simulations against the requirements envisaged by the law. A further challenge is that representative data for simulation may be unavailable, thus necessitating a data generator. A hard-coded generator is difficult to build and validate.

We develop a framework for legal policy simulation that is aimed at addressing the challenges above. 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.

Keywords Legal Policies, Simulation, UML Profiles, Model-Driven Code Generation, Probabilistic Data Generation

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

1 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 [15]. 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 [15], SYSIFF [10] and POLIMOD [45], 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, the spreadsheets tend to get complex, thus making it difficult to check whether the policy implementations match their specifications [20, 21, 35]. 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 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 article 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 [38].

Our simulation framework leverages our previous work [42], 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., the simulation of system behaviors.

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

1) We augment our previously-developed methodology for policy modeling [42] so as to enable policy simulation.
2) We develop a UML profile [18] to capture the probabilistic characteristics of a population. This profile shares some common goals with MARTE [33] in terms of supporting probabilistic analysis. However, MARTE is geared towards embedded systems and largely limited to probabilistic attributes. Our profile supports several additional probabilistic notions, including probabilistic 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 over six policies from Luxembourg's 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.

This article is an extension of a previous conference paper [43] published at the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2015). In addition to significantly extending the descriptions in this earlier conference paper and providing a more thorough treatment of our simulation framework, this article makes the following technical contributions: First, we have evolved the initial implementation of our framework into a tool, named PoliSim. We present this tool as part of this article. Second, we have made major improvements in our data generator, including new features that make the generated data more realistic in situations where conditional probabilities are used for guiding data generation. We conduct a complete re-evaluation of our revised data generator. The results suggest that (1) the revised data generator has a shorter execution time, (2) one can obtain more reliable simulation outcomes using smaller generated data samples than before, and (3) the

generated data shows a higher degree of consistency than before.

Structure. The remainder of the article is organized as follows. Section 2 provides an overview of our framework. Section 3 describes our policy modeling technique. Section 4 discusses how policy models are transformed into simulation code. Sections 5 elaborates our profile for specifying the probabilistic characteristics of the simulation population. Section 6 presents our simulation data generation algorithm. Section 7 outlines tool support. Section 8 reports on the empirical evaluation of our simulation framework. Section 9 points out the limitations of our simulation framework and the threats to the validity of our empirical evaluation. Section 10 compares our work with related work. Finally, Section 11 concludes the article with a summary and directions for future work.

2 Simulation Framework Overview

Fig. 1 presents an overview of our simulation 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. In Step 2, *Generate code*, our framework automatically transforms the policy models into executable code that will be used to run the simulation in Step 5.

If simulation data is already available (e.g., historical data), we move directly to Step 5, where the data is processed by the executable simulator and the simulation results produced. If no simulation data is available, we enrich in Step 3 the domain model with statistical information about the population over which simulation needs to be performed. Based on this statistical information, we generate in Step 4 the required simulation data. This data is processed in Step 5 by the executable simulator in exactly the same manner that existing data would be processed.

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 results comparison, meaning that the user can provide an original and a modified set of policies, execute both policy sets over the same simulation data, and compare the simulation results in order to quantify the impact.

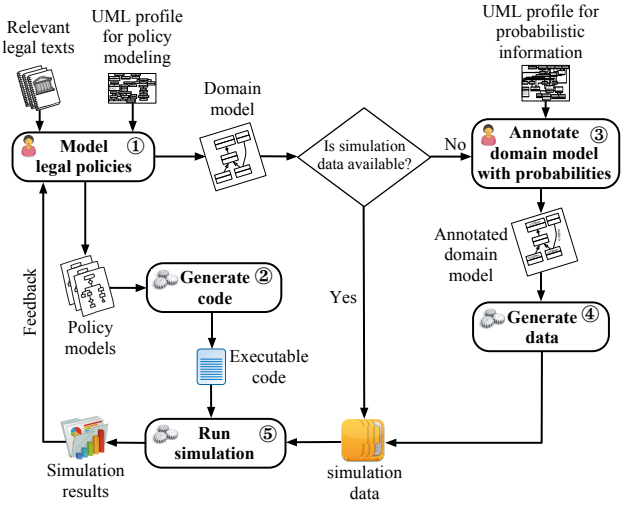


Fig. 1 Simulation Framework Overview

3 Modeling Legal Policies

The first step of our simulation framework, as depicted earlier in Fig. 1, is modeling the legal policies. This step is an adaptation of our previous work [42], where we developed a UML-based methodology and notation for specifying (procedural) legal policies. The conceptual basis for the work is a field study through which we built a metamodel for the information that needs to be captured in legal policies; see [42] for the details of our field study and metamodel.

Our earlier work was motivated not by simulation but rather by model-based testing [46]. In that context, the legal policy models were meant to serve as test oracles (verdicts). The evaluation of these oracles has no impact on the test data. In contrast, for simulation purposes, we need the ability to modify the simulation data during simulation execution in order to store the simulation results. To meet this requirement, we extend our modeling notation so as to allow policy models to perform operations with side effects. Specifically, the extensions make it possible for the policy models to update any object in the simulation data.

Legal policy modeling yields two types of models as illustrated in Fig. 1: (1) a set of policy models in the form of UML Activity Diagrams (ADs), and (2) a domain model in the form of a UML Class Diagram. The policy models provide a precise interpretation of the legal policies that need to be simulated; the domain model formalizes the input and output data for the legal policies. We defer our discussion and illustration of domain models to Section 5, where we cover domain models alongside the probabilistic information that we attach to them for characterizing the simulation popu-

lation. In the remainder of the section, we concentrate on the policy models (expressed as ADs).

To be able to specify the policy models in a precise manner, we customize ADs with additional semantics. The customization is performed via a UML profile [18].[1] The profile is shown in Fig. 2. The shaded elements in the figure represent UML metaclasses and the non-shaded elements represent the stereotypes. We provide definitions for the stereotypes in Table 1.

We illustrate the application of the profile over an example policy from Luxembourg's Income Tax Law. The policy concerns "invalidity" deduction, which is a special tax deduction granted to taxpayers who are infirm or have been disabled by injury or illness. The textual description of the policy (extracted from the relevant legal texts and translated into English from the original text in French) is shown in Fig. 4. The text excerpt defines: (1) the eligibility criteria for the deduction, (2) the annual lump-sums to use for determining the amount of the deduction to grant, and (3) instructions for computing the deduction. Specifically, the policy envisages no deduction for non-disabled taxpayers or disabled taxpayers with a disability rate below 25%. An annual lump-sum of 1455 € is used to compute the deduction for taxpayers with vision disabilities (case E. of the policy). For any other disability type (cases A. to D. of the policy), a lump-sum is determined based on the taxpayer's disability rate. The invalidity deduction is the lump-sum prorated to the period during which the taxpayer has been paying taxes (active) during the current taxation year.

In Fig. 3, we show how we model the above policy using our customized AD notation. For succinctness, we hereafter refer to the policy model of Fig. 3 as ID (Invalidity Deduction). At a high level, each policy model can be divided into three main parts: (1) the policy declaration on the top, (2) the activity flow in the center, and (3) the parameter declarations on the left. Below, we discuss these three parts alongside the stereotypes relevant to each part.

Policy declaration. Each policy model is annotated with the *«policy»* stereotype, providing the name of the policy. In ID, this name is `invalidity`. Each policy model further has a *«context»* stereotype indicating the OCL context for evaluating the OCL expressions used within the activity flow and parameter declarations (discussed next). For ID, the context is the *TaxPayer* class from the domain model (partially depicted in Fig. 7).

The activity flow. The activity flow in ID is composed of three alternative paths. The decision nodes that determine which path to take are annotated with the *«decision»* stereotype. Based on a given taxpayer's situation, the appropriate calculation is applied as defined in the text excerpt of Fig. 4. Each calculation in ID is denoted by an action annotated with the *«calculate»* stereotype. Attached to each calculation is a formula, annotated with the *«formula»* stereotype. For instance, the formula that is attached to the *No deduction* calculation returns the constant value zero. Both decision nodes and formulas are expressed using OCL expressions.

The result of a calculation can be stored in an intermediate variable annotated with the *«intermediate»* stereotype, e.g., `expected_amount` in ID. An intermediate variable can in turn be used by the update actions in a policy model (annotated with the *«update»* stereotype). Update actions modify the simulation data (instance of the domain model). For example, the *Store simulation results* in ID stores the computed deduction (in the `invalidity` attribute of a given taxpayer's tax card).

ID further takes into account the fact that a taxpayer may have multiple (simultaneous or sequential) incomes. Although not explicitly stated in the text excerpt of Fig. 4, the invalidity deduction applies to the individual incomes of a taxpayer (as opposed to the taxpayer or their household). To deal with taxpayers having multiple incomes, we use an expansion region.

Briefly, an expansion region is an activity region that executes multiple times over the elements of an input

Table 1 Description of the Stereotypes in the Profile of Fig. 2

Stereotype	Description
«policy»	Defines an activity as a legal policy and provides information for code generation
«context»	Defines the OCL context in which the legal policy is specified
«iterative»	Defines an iterative region
«iterator»	Defines the variable over which iteration is performed
«decision»	Defines a decision step within a legal policy
«direction» (abstract)	Defines the direction of a parameter (in or out)
«in»	Defines an input parameter to a legal policy
«out»	Defines an output from a region (the entire legal policy or an iterative region within the policy)
«query»	Defines a query for retrieving the value of a given input parameter
«formula»	Defines the formula for a computation
«calculate»	Defines an operation that computes a value
«update»	Defines an operation that updates an object or the value of a parameter
«intermediate»	Defines an intermediate value resulting from a computation
«source» (abstract)	Defines a traceability link to a source (circular, article, etc.)
«fromrecord»	Declares an (input) parameter as being retrieved from a record (instance)
«fromlaw»	Declares an (input) parameter as originating from a legal text
«fromagent»	Declares an input parameter as being provided by a subject matter expert or a user

[1] This profile is not to be confused with the profile that we present in Section 5 for extending domain models with probabilistic information.

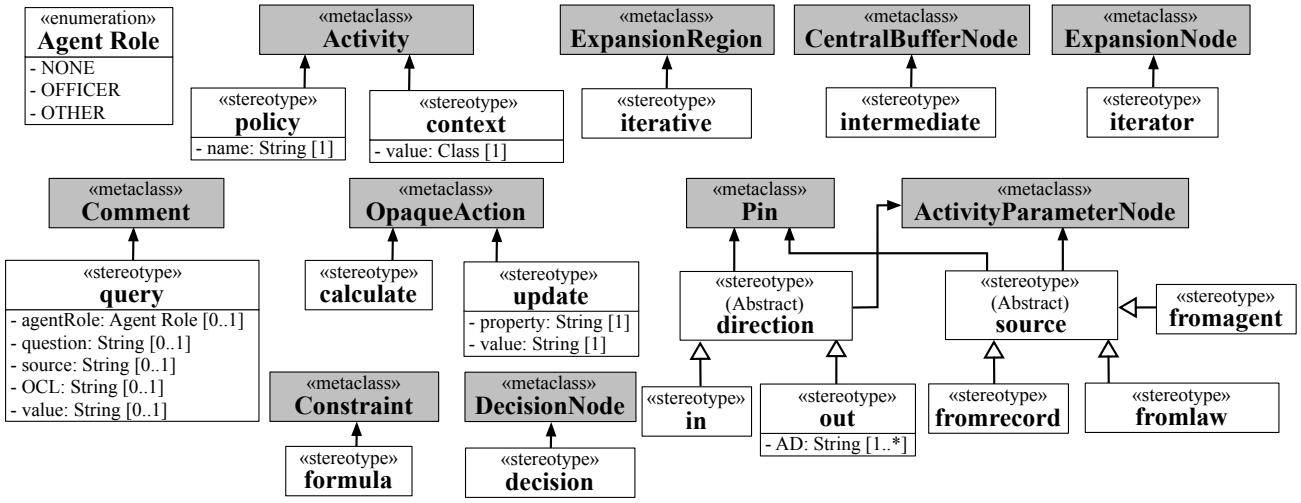


Fig. 2 Profile Customizing UML Activity Diagrams for Legal Policy Models

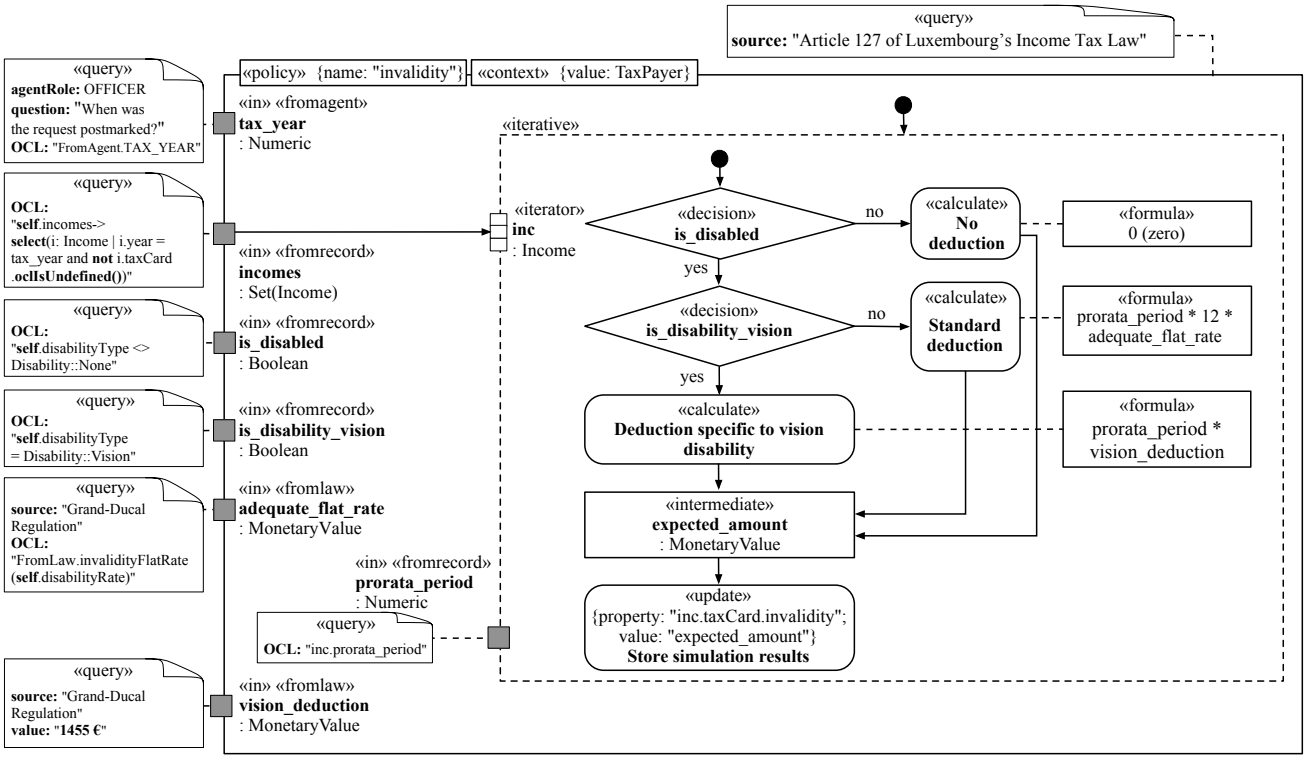


Fig. 3 (Simplified) Policy Model for Calculating Invalidity Deduction

collection [32]. UML defines three execution modes for expansion regions: *iterative*, *parallel*, and *stream* [32]. Of these, we use only the iterative mode, marked by the *«iterative»* stereotype. In this mode, executions are performed sequentially and according to the order of the elements in the input collection. If the elements in the collection are not ordered, e.g., when the collection is a bag, a random order will be used. In the case of our example, ID, the expansion region is iteratively executed over each income that is associated with a tax card. The name of the region's expansion node, `inc` in our example, serves as an alias for the iterator element in an individual execution. This alias which is annotated with the *«iterator»* stereotype can be used in the OCL expressions associated with the inputs of the expansion region.

Parameters declaration. The input parameters of a policy model are represented by small gray rectangles and annotated with the *«in»* stereotype. The origin of

Deduction for Disabled and Infirm Taxpayers

The deduction for the extraordinary expenses of the disabled and infirm is reserved for taxpayers […] who belong to the categories below:
A. disabled in war, receiving regular compensation for bodily war damages;
B. victims of a work accident;
C. physically-disabled persons other than those mentioned under A and B and mentally-handicapped persons;
D. taxpayers suffering from recognized occupational diseases;
E. taxpayers whose central vision is zero or less than 1/20.

The annual deduction payable to taxpayers described in E is set to 1455 €. The annual deduction payable to persons under A to D is determined as follows:

Disability rate	Annual deduction (lump-sum)
25% to 35% (excluded)	150 €
from 35% to 45% (excluded)	225 €
from 45% to 55% (excluded)	375 €
from 55% to 65% (excluded)	450 €
from 65% to 75% (excluded)	525 €
from 75% to 85% (excluded)	585 €
from 85% to 95% (excluded)	645 €
from 95% to 100% (included)	735 €

The deduction has to be prorated to the full-time equivalent period during which the taxpayer has been active over the course of the tax year.

Fig. 4 (Simplified) Policy Description for Invalidity Deduction

each parameter is captured using one of the following three stereotypes:

(1) *«fromagent»*, when an input parameter is provided by an agent, e.g., an *OFFICER* as is the case for the `tax_year` input in ID; (2) *«fromrecord»*, when an input parameter is retrieved from the simulation data. In such a case, a query, written in OCL, needs to be provided. For example, the `incomes` input parameter in ID, which denotes the set of a given taxpayer's incomes (associated with tax cards), is the result of evaluating an OCL query; and (3) *«fromlaw»*, when an input parameter is defined in a legal text. For example, the annual lump-sum `vision_deduction` is a value (1455 €) originating directly from the text of law.

Regardless of the origin of an input parameter, all the information relevant to the parameter is maintained in a (structured) comment, annotated with the *«query»* stereotype. For example, the comment attached to the `vision_deduction` parameter in ID captures the (legal) source and the value of this parameter. Furthermore, input parameters have scopes and can be defined either globally at a policy-model level, or locally at the level of nested expansion regions. In ID, all input parameters are global, except for `prorata_period` which is local and visible only within the expansion region to which the parameter is attached.

Our example, ID, covers all of the (non-abstract) stereotypes in the profile of Fig. 2, except *«out»*. This stereotype is used in some of our policy models to indicate that the output from the execution of a given policy model will be the input to other policy models.

4 Simulation Code Generation

To enable the simulation of a given set of policy models, we transform the models into code (Step 2 in Fig. 1). Below, we outline this transformation.

The transformation is an adaptation of the rule-based model-to-text transformation presented in our previous work [42]. The original transformation was aimed at generating from policy models OCL invariants that can be used as oracles (verdicts) for model-based testing. A simple but important requirement for simulation is to be able to store the simulation results. This requirement cannot be met in a straightforward manner through OCL, due to the language being side-effect-free.

To be able to store the simulation results, our adapted transformation has Java as its target language. The generated Java code makes calls to an OCL evaluator. The combination of Java and OCL makes it possible to handle updates and manage the simulation outcomes through Java, while still using OCL for querying domain model instances. For succinctness and due to the similarity of our adapted transformation algorithm and rules to the original ones [41], we do not present the technical details of the transformation in the main body of this article. These details are provided in Sections A.1 and A.2 of the appendix.

Fig. 5 shows a fragment of the simulation code generated for the policy model 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 defines the queries used to retrieve the appropriate inputs to process (e.g., L. 5-7).

In the code fragment, the interaction with the OCL evaluator is performed via methods in a utility class, named `OCLInJava`, which is an internal component of our simulation engine. This class is used, among other things, for defining the OCL context (L. 2), updating objects such as input parameters (L. 20), evaluating OCL queries (e.g., L. 7), and keeping track of intermediate values, e.g., `tax_year` (L. 4) and `inc` (L. 9).

As mentioned earlier, the resulting simulation code will be executed over either existing or generated simulation data to produce the simulation results.

5 Expressing Population Characteristics

In this section, we present our UML profile [18] for capturing the probabilistic characteristics of a population. This profile is the basis for Step 3 of our simulation

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

Fig. 5 Fragment of Generated Code for the Model of Fig. 3

framework (Fig. 1). The profile, which extends UML class diagrams, is shown in Fig. 6. The shaded elements in the figure represent UML metaclasses and the non-shaded elements – the stereotypes of the profile. Below, we describe the stereotypes and illustrate them over a (partial) domain model of Luxembourg's Income Tax Law, shown in Fig. 7. Rectangles with thicker borders in Fig. 7 are constraints (not to be confused with classes). References to Fig. 6 for the stereotypes and Fig. 7 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 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 *«OCL query»* (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 0.21 and 0.5, and so on. In this histogram, all the bins are ranges, e.g., "0..0.2". Bins can be single values as well, e.g., the first three bins of the histogram applied to the constraint named *taxpayers per address*.

  An example of a *«probabilistic value»* that uses OCL 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, and (2) attributes defined as collections. To illustrate, consider the association between *TaxPayer* and *Address*. The cardinality on the *Address* end of this association is expressed as a constraint named *main address mult*, stating that the cardinality is a random variable drawn from a certain bar chart. Similarly, the cardinality on the *TaxPayer* end is expressed as a constraint named *taxpayers per address*, which describes via a bar chart the number of taxpayers sharing the same address.

- ***«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 (constraints annotated with *«OCL query»*), and (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 the cases, the beneficiary is an existing household member as per specified in the *«OCL query»* named

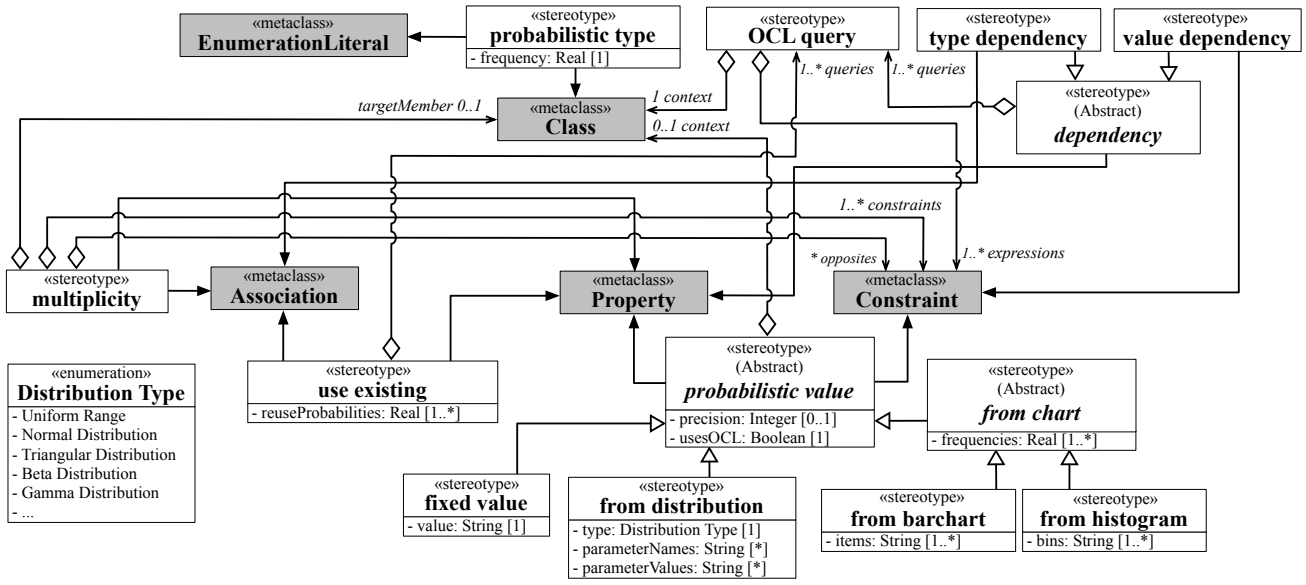


Fig. 6 Profile for Expressing the Probabilistic Characteristics of a Population

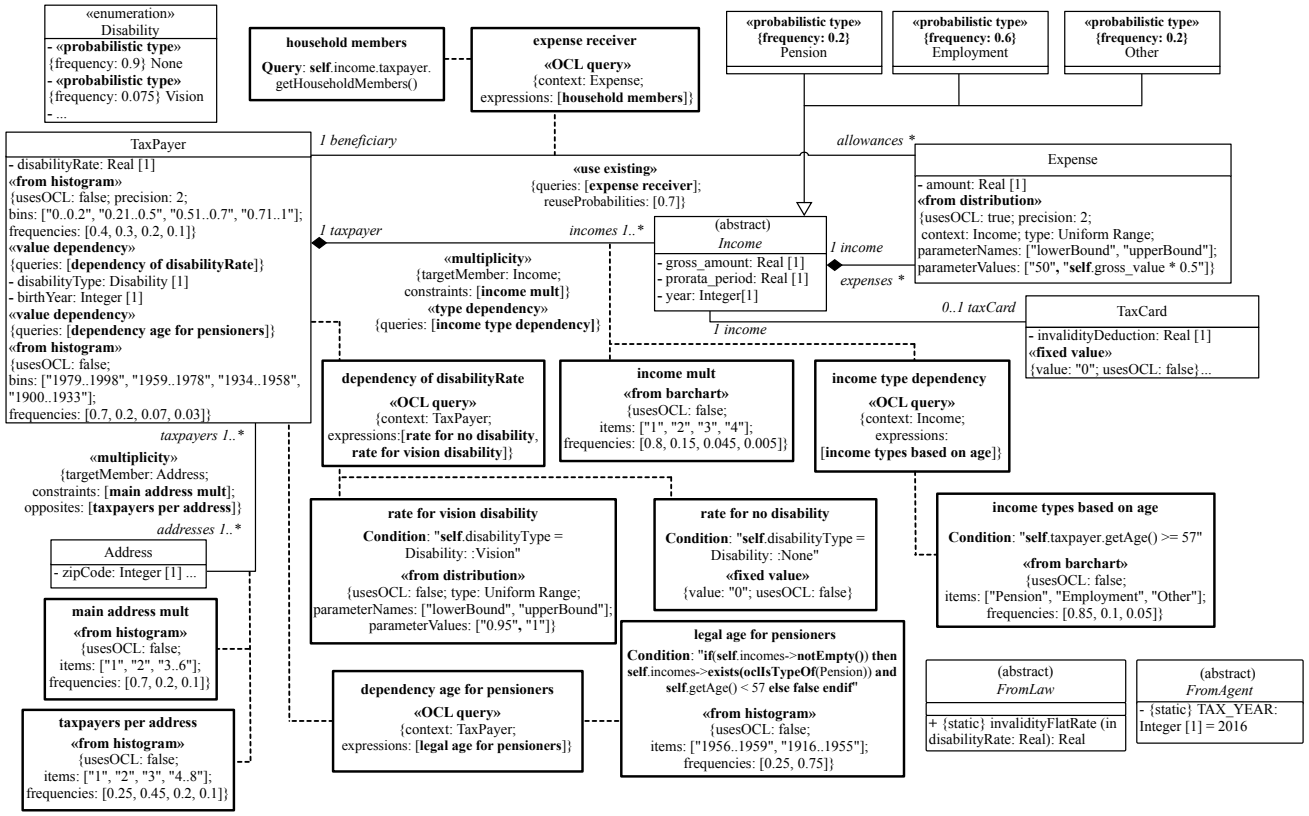


Fig. 7 Partial Domain Model for Luxembourg's Income Tax Law Annotated with Probabilistic Information

expense receiver; for the remaining 30%, a new *TaxPayer* needs to be created. *«use existing»* supports collections of OCL queries and probabilities, rather than merely an individual query and an individual probability. This is because, in UML, one can apply a particular stereotype only once to a model element. However, in the case of *«use existing»*, one may want to define multiple object pools. 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: *«type dependency»* and *«value dependency»*. The former extends the Property and Constraint metaclasses; whereas the latter extends the Property and Association metaclasses. In either case, the conditional probabilities are connected to a dependency via an *«OCL query»*. An *«OCL query»* is essentially a container for a set of Boolean expressions (represented as OCL constraints) along with the (OCL) context in which these expressions should be evaluated. If a certain expression evaluates to true, the *«probabilistic value»* applied to that expression will be used. We note that at most one expression from the set of expressions in an *«OCL query»* should evaluate to true at any given time.

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* must be zero. If *disabilityType* is vision, then the distribution of *disabilityRate* follows the histogram in the constraint named *rate for vision disability*. This constraint is contained in the *«OCL query»* named *dependency of disabilityRate*. 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 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 type for generating an attribute or for filling 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 57, *Income* is more likely to be a *Pension* (85%) than an *Employment* (10%) or *Other* (5%).

• ***Consistency constraints for the profile:*** 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 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.

Fig. 8 presents an example of a consistency constraint aimed at ensuring that the specializations of *«probabilistic value»* are applied mutually exclusively. For any element annotated with some specialization of *«probabilistic value»* (L. 2-8), the constraint verifies that one and only one of the specializations is applied to the element (L. 9-12). A complete list of the consistency constraints for our profile can be found in Appendix A.3.

```
1  context probabilistic_value inv:
2  let annotatedElement: Element =
3    if(self.base_Constraint.oclIsUndefined()) then
4       self.base_Property
5    else
6       self.base_Constraint
7    endif
8  in
9  annotatedElement.getAppliedStereotypes()
10   ->select(s: Stereotype |
11          s.oclIsKindOf(probabilistic_value))
12          ->size()=1
```

Fig. 8 Example Consistency Constraint for the Profile of Fig. 6

6 Simulation Data Generation

In this section, we describe the process for automated generation of simulation data, i.e., Step 4 of the framework in Fig. 1. An overview of this process is shown in Fig. 9. The inputs to the process are: a domain model annotated with the profile of Section 5, and the set of policy models to simulate. The process has four steps, detailed in Sections 6.1 through 6.4. In the remainder of this section, any reference to a particular "Step" concerns the steps of the process in Fig. 9, rather than the steps of our overall framework (Fig. 1).

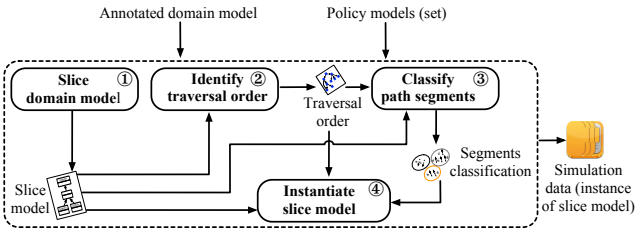


Fig. 9 Overview of Simulation Data Generation

6.1 Domain Model Slicing

In Step 1 of the process in Fig. 9, *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. When the added element is a class, its specializations are also added to the slice. 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. 10(a), we show an example slice model, obtained from the domain model of Fig. 7 specifically for the purpose of simulating the policy model in Fig. 3, named ID. Among other elements, the *Expense* class has been excluded from the slice because, to simulate the policy model of Fig. 3, we do not require instances of *Expense*. To avoid clutter in Fig. 10(a), we have not shown the constraints. All the constraints from the domain model of Fig. 7 except four are part of the slice model. The four constraints excluded from the slice model of Fig. 10(a) are: *main address mult*, *taxpayers per address*, *expense receiver*, and *household members*. These four constraints are not relevant for the simulation of ID.

The slice model of Fig. 10(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 evaluating OCL queries and may further play a role in determining the order of object instantiations. We describe how we determine this order next.

6.2 Identifying a Traversal Order

In Step 2 of the process in Fig. 9, *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 (belonging to a stereotype in the slice model) can only be evaluated once m' has been instantiated.

The orderings are computed via *topological sorting* [13] 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 ($C_i \neq C_j$), 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. 10(b), we illustrate DGs and topological sorting over the slice model of Fig. 10(a). The upper part of Fig. 10(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. 7), 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 edge in the attribute-level DG of Fig. 10(b) is induced by the *«value dependency»* stereotype of the attribute *disabilityRate* (Fig. 7), specifically by the OCL constraints *rate for vision disability* and *rate for no disability*. The numbers in the DGs of Fig. 10(b) denote one possible total ordering for the respective DGs. Computing these orderings is linear in the size of the DGs [13] 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

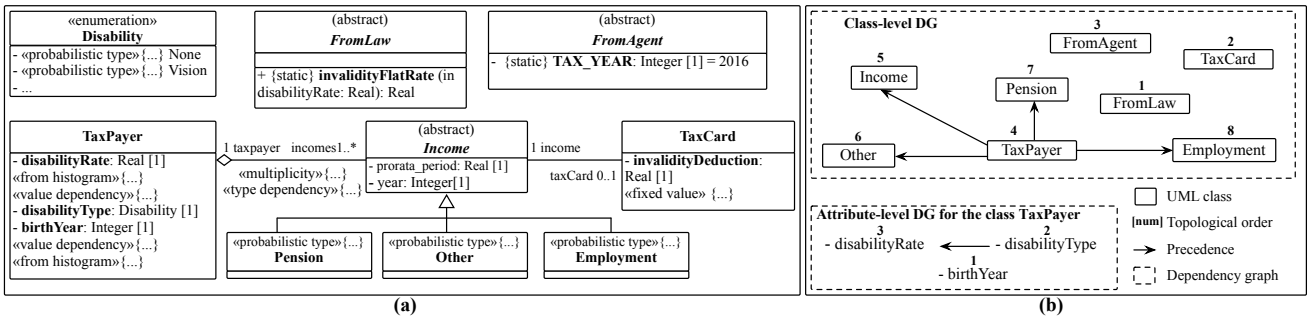


Fig. 10 (a) Excerpt of Slice Model for Simulating the Policy Model of Fig. 3, (b) Topological Sorting of Elements in (a)

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 will not encounter an uninstantiated object at the time the object 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.

6.3 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. 9, *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 avoid infinite recursion. 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. 3, 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 containment associations (i.e., compositions or aggregations). For example, if the model of Fig. 3 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 composition associations. 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.

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. 10(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*.

The above exploration is also applied to attributes typed by some class of the slice model, as assigning a value to such attributes amounts to traversing and 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. 10(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 the slice model of Fig. 10(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.

6.4 Instantiating the Slice Model

The last step of the process of Fig. 9, *Instantiate slice model*, generates the simulation data, i.e., an instance model of the slice model. 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, (5) the last traversed segment or attribute of the slice model, and (6) an initially empty map for keeping track of objects that have some unsatisfied multiplicity constraint. 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 customizable parameter (say 10,000). SDG has four main parts, explained below.

(1) Class selection and instantiation (L. 1–9). 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 (L. 6) and instantiates (L. 9) a *non-abstract* class from the following set: {C}∪{all descendants of C}. SDG's selection process (L. 6) is shown in Alg. 2. The selection is based on the *«type dependency»* and *«probabilistic type»* stereotypes attached to C (L. 1-6 in Alg. 2). As can be seen from Alg. 2, the *«type dependency»* stereotypes are prioritized over the *«probabilistic type»* stereotypes. This is necessary for correctly handling conditional probabilities. If both of these stereotypes are absent or fail to yield a specific class, a random (non-abstract) class from the set above, i.e., C and its descendants, is selected (L. 8 in Alg. 2).

To minimize potential deviations from the specified population characteristics, Alg. 2 dynamically adjusts, based on the characteristics of the current object pool, the probabilistic information to use for the creation of future objects (L. 9-10 in Alg. 2). These adjustments are aimed at better handling conditional probabilities. To illustrate, consider the classes *Employment*, *Pension*, and *Other* in Fig. 7. The *«probabilistic type»* stereotypes attached to these classes prescribe certain overall frequencies for these income types. At the same time, these income types are subject to a conditional probability captured by the *«type dependency»* stereotype applied to the association between *TaxPayer* and *Income*. Specifically, the *income type dependency* query in this stereotype conditions the selection of a concrete *Income* class upon the age of the taxpayer. Consequently, the instances generated for *Employment*, *Pension* and *Other* are constrained not only by the overall frequencies in the *«probabilistic type»* stereotypes, but also by the conditional probability.

To satisfy the above constraints simultaneously, we adaptively adjust the user-specified *«probabilistic type»* frequencies shown in Fig. 7. To do so, we proceed as follows: We first subtract from the user-specified frequencies the corresponding frequencies observed in the already-generated data. We then zero out any negative frequency resulting from the subtraction; this is to avoid generating objects that are already over-represented in the object pool. Next, to bring up the total area of the new histogram to 100%, we compute the sum S of the frequencies that result from the subtraction and zeroing of the negative frequencies. Subsequently, we distribute S proportionally over the non-zero frequencies. For example, suppose that the frequencies observed

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

Inputs : (1) a slice model $\mathcal{S}$; (2) a class C ∈ $\mathcal{S}$ to instantiate; (3) the orderings, $\mathcal{O}$, from Step 2; (4) path segment classifications, $\mathcal{P}$, from Step 3; (5) the last traversed segment or attribute, source ∈ $\mathcal{S}$ (initially *null*); (6) a map unsat (key: a segment with unsatisfied multiplicities, value: a list of instances) (initially *null*)

Output : an instance of class C

```
1  Let res be the instance to generate (initially null)
2  if (source is not null) then
3  |  res ← Attempt «use existing» of source (if the stereotype is present)
4  |  if (res is not null) then return res
5  Let chosen be the class to instantiate (initially null)
6  chosen ← CS(S, C, source)
7  if (chosen is null) then return null
8  else
9  |  res ←Instantiate (chosen)
10 |  Let attributes be the set of chosen's attributes
11 |  attributes ← SortAttributesByOrder (attributes , O)
12 |  atts_now ← RemoveUnreadySteryotypes (attributes, res)
13 |  atts_after ← RemoveReadySteryotypes (attributes, res)
14 |  foreach (att ∈ atts_now) do
15 |  |  AVG(S, att, O, P, unsat)
16 |  Let paths be the Safe and PotentiallyUnsafe outgoing segments from chosen
17 |  foreach (seg ∈ SortSegmentsByOrder (paths, O)) do
18 |  |  nextC ← target class of seg
19 |  |  mult ← Attempt «multiplicity» of seg
20 |  |  if (mult is null) then
21 |  |  |  mult ← random number from multiplicity range of seg
22 |  |  Let objects be an (initially empty) set of instances
23 |  |  for (i ← 0; i < mult) do
24 |  |  |  Let obj be an instance (initially empty)
25 |  |  |  P' ← P
26 |  |  |  if (seg is PotentiallyUnsafe in P) then
27 |  |  |  |  obj ← randomly pick, from the objects pool, an object of kind nextC that is not linked to res for the segment seg
28 |  |  |  |  if (obj is null) then
29 |  |  |  |  |  if (maximal traversal count of seg is reached) then
30 |  |  |  |  |  |  Switch seg from PotentiallyUnsafe to Excluded in P'
31 |  |  |  if (obj is not null) then
32 |  |  |  |  objects.add(obj)
33 |  |  |  else
34 |  |  |  |  if ((unsat.get(seg) \ res.seg) = ∅) then
35 |  |  |  |  |  objects.add (SDG (S, nextC, O, P', seg, unsat))
36 |  |  |  |  else
37 |  |  |  |  |  obj ← random object from unsat for the key seg
38 |  |  |  |  |  objects.add(obj)
39 |  |  |  |  |  remove obj from unsat for the key seg
40 |  |  Let association be the underlying association of seg
41 |  |  res.setLinks (association, objects)
42 |  |  if (minimal mult. of seg's opposite segment > 1) then
43 |  |  |  op_mult ← Attempt «multiplicity» of seg's opposite
44 |  |  |  if (op_mult is null) then
45 |  |  |  |  op_mult←random number from multiplicity range of seg's opposite
46 |  |  |  |  for (j ← 0; j < (op_mult − 1)) do
47 |  |  |  |  |  add objects.last() to the list of instances in unsat for the key seg
48 |  |  foreach (att ∈ atts_after) do
49 |  |  |  if (att has at least one steryotype) then
50 |  |  |  |  AVG(S, att, O, P, unsat)
51 |  return res
```

Alg. 2: Class Selector (**CS**)

Inputs : (1) a slice model $\mathcal{S}$; (2) a class C $\in \mathcal{S}$; (3) a segment or attribute, source $\in \mathcal{S}$
Output : a class res $\in \mathcal{S}$ (res can be either C or a class from one of C's descendants)

```
1  res ← null
2  if (source has «type dependency») then
3  | res ← Attempt «type dependency» of source
4  if (res is null) then
5  | if (C's immediate subclasses have «probabilistic type») then
6  | | res ← Attempt «probabilistic type» from C
7  | else
8  | | res ← Randomly pick, from C and all C's descendants, a
   | | non-abstract class
9  if (source's stereotypes need adjustment) then
10 | adjust «probabilistic type»'s frequencies for C's subclasses
11 return res
```

in the data that has been generated already are as follows: 80% for *Employment*, 15% for *Pension*, and 5% for *Other*. Subtracting these frequencies from those in Fig. 7 would yield the following: -20% for *Employment*, 5% for *Pension*, and 15% for *Other*. Since the frequency for *Employment* is negative, we set it to zero. The subtraction leaves a deficit of $100\% - (0\% + 5\% + 15\%) = 80\%$ in the total area of the new histogram. This deficit is proportionally distributed over the non-zero frequencies, i.e., *Pension* and *Other*. The adjusted frequencies would therefore be 0% for *Employment*, $5\%+80\%\times\frac{5}{5+15} = 25\%$ for *Pension*, and $15\% + 80\% \times \frac{15}{5+15} = 75\%$ for *Other*. To facilitate understanding, we summarize in Table 2 the calculations that we described above.

Table 2 Example of Adaptive Adjustment of Frequencies

	Frequency		
	Employment	*Pension*	*Other*
User-specified (Fig. 7)	60%	20%	20%
Observed in already-generated data	80%	15%	5%
After subtraction	-20%	5%	15%
After zeroing out negative frequencies	0%	5%	15%
After proportional distribution of the area deficit	0%	25%	75%

(2) Attribute value assignment (L. 10–15 and L. 48–50). SDG calls Alg. 3 (L. 15 and L. 50) to assign values to the attributes of C according to C's attribute-level ordering (computed in Step 2). First, Alg. 3 determines the required number of values to assign to a given attribute based on the attached *«multiplicity»* stereotype (L. 1 in Alg. 3). If this stereotype is absent or fails to determine the number of values to assign, a random number that satisfies the desired multiplicity will be picked (L. 2-3 in Alg. 3). Subsequently, values for primitive attributes are generated by processing the *«value dependency»* and *«probabilistic value»* stereotypes, if either stereotype is present (L. 7-12 in Alg. 3). We note that priority is given to *«value dependency»* over *«probabilistic value»* in order to correctly handle conditional probabilities. If a primitive attribute is still unassigned after processing the *«value dependency»* and *«probabilistic value»* stereotypes, a random value is assigned to it (L. 11-12 in Alg. 3). Similar to Alg. 2, the frequencies in Alg. 3 are dynamically adjusted when necessary (L. 14-15 in Alg. 3). For an attribute typed by a class from the slice model, SDG recursively creates a random (but adequate) number of objects (L. 18-20 in Alg. 3).

Alg. 3: Attribute Value Generator (**AVG**)

Inputs : (1) a slice model $\mathcal{S}$; (2) an attribute att $\in \mathcal{S}$; (3) the orderings, $\mathcal{O}$, from Step 2; (4) path segment classifications, $\mathcal{P}$, from Step 3; (5) a map unsat (key: a segment with unsatisfied multiplicities, value: a list of instances)
Output : void, this procedure assigns one or more values/objects to the attribute att

```
1  mult ← Attempt «multiplicity» of att
2  if (mult is null) then
3  | mult ← random value from multiplicity range of att
4  Let att_values be an (initially empty) set of values
5  if (att is not typed by some class of S) then
6  | for (i ← 0; i < mult) do
7  | | value ← null
8  | | value ← Attempt «value dependency» of att
9  | | if (value is null) then
10 | | | value ← Attempt «probabilistic value» of att
11 | | if (value is null) then
12 | | | value ← a random value
13 | | att_values.add(value)
14 | | if (att's stereotypes need adjustment) then
15 | | | adjust distributions of att's «probabilistic value»
16 | att ← att_values
17 else
18 | Let att_objects be an (initially empty) set of instances
19 | for (i ← 0; i < mult) do
20 | | att_objects.add (SDG (S, typeOf (att), O, P, att,
   | | unsat))
21 | att ← att_objects
```

An important remark about assigning values (or objects) to the attributes of a given class is that these values are tentative and may change over the course of data generation. The need for changing the value of an attribute after it has been assigned arises from the fact that some of the stereotypes attached to the attribute may be referring to objects that have not yet been instantiated. In other words, not all the stereotypes attached to an attribute can be processed at the time that the attribute is first assigned (i.e., immediately after the object to which the attribute belongs has been created). Any unprocessed stereotype therefore needs to be revisited at the end of a particular call to the SDG algorithm. To this end, for any given attribute, SDG first determines which stereotypes attached to the attribute should be processed immediately (L. 12) and which ones should be deferred (L. 13).

To illustrate, consider attributes *disabilityRate* and *birthYear* of *TaxPayer* in the slice model of Fig. 10(a). After creating an instance of *TaxPayer*, a value will be immediately assigned to *disabilityRate* by applying the *«from histogram»* and *«value dependency»* stereo-

types (shown in Fig. 7). Since both of these stereotypes can be processed immediately, a final value is assigned to *disabilityRate*. As for the *birthYear* attribute, only the *«from histogram»* stereotype can be processed immediately; the processing of the *«value dependency»* stereotype is deferred because this stereotype requires an instantiation of the taxpayer's incomes. Once the incomes have been generated, SDG will, under certain conditions (when the taxpayer turns out to be a pensioner), update the initially-assigned value based on the *«value dependency»* stereotype. As illustrated by this example, SDG gives preference to deferred stereotypes in determining the value of an attribute. The rationale behind this decision is that deferred stereotypes are often associated with the consistency of the data being generated, while the non-deferred stereotypes are typically concerned with the representativeness of the generated data. If non-deferred stereotypes are given priority, the internal consistency of the data will be reduced. In contrast, prioritizing deferred over non-deferred stereotypes can only cause drifts from the desired distributions, for which we already have safeguards through the dynamic adjustments implemented in Alg. 2 and Alg. 3.

(3) Segment traversal (L. 16–41). 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 earlier (L. 31-33). The traversal ignores *Excluded* segments and traverses, based on the ordering of classes computed in Step 2, *Safe* and *PotentiallyUnsafe* segments (L. 16-17). The instantiation process for traversed segments is recursive (L. 35). Nevertheless, since traversing *PotentiallyUnsafe* segments may cause infinite recursions, SDG attempts first to reuse existing objects from the object pool instead of making a new recursive call to SDG (L. 26-27). If no suitable object is found for reuse, SDG allows *PotentiallyUnsafe* segments to be traversed for a finite number of times (L. 28-30). The maximum number of traversals permitted is a configurable parameter. We set this parameter to 10. Handling *PotentiallyUnsafe* and *Excluded* segments in the manner described above avoids the possibility of infinite recursions while still allowing, among other things, the instantiation of reflexive associations.

(4) Handling unsatisfied multiplicities (L. 37–39 and L. 42–47). Since SDG traverses associations in one direction only (*Safe* and *PotentiallyUnsafe* segments), the multiplicity at *Excluded* segment ends is always equal to one. Therefore, multiplicity constraints for *Excluded* segments might be left unsatisfied. SDG defers handling unsatisfied multiplicities to future recursions. Specifically, the algorithm detects and stores all segments that have unsatisfied multiplicities alongside all the objects that have been associated with these segments so far (L. 42–47). Subsequently and in future recursions, SDG attempts to use newly-created objects for meeting the unsatisfied multiplicities (L. 37–39). This strategy makes it more likely to satisfy m-to-n multiplicities.

For the sake of argument, suppose that the slice model of Fig. 10(a) also includes the *Address* class from Fig. 7 and the association between *Address* and *TaxPayer*. Further, suppose the *«multiplicity»* stereotype attached to this association requires (based on a random choice from the underlying barchart) that there should be two taxpayers living at a given address, say *addr1*. Since traversal is from *TaxPayer* to *Address*, the segment *Address*→*TaxPayer* will be *Excluded* and thus the multiplicity constraint of *addr1* will be left unsatisfied. SDG keeps track of *addr1* and the involved segment. The next time SDG instantiates *TaxPayer*, it will link the newly-created instance to *addr1* for that particular segment instead of traversing the segment and generating new instances of *Address*.

7 Tool Support

We provide an implementation of our simulation framework in a tool named PoliSim (Policy Simulation). The tool is available at people.svv.lu/tools/polisim.

PoliSim has been developed as an Eclipse plugin (eclipse.org). Fig. 11 shows the overall architecture of the tool. In the figure, we distinguish between the roles of "legal expert" and "modeler". While a key objective of our work is to make modeling accessible to legal experts, it may be difficult for legal experts to manage the model construction activities on their own. To this end, legal experts may need assistance from analysts with modeling expertise.

In line with what was discussed in Section 2, our tool takes as input two types of models: the policy models and a domain model annotated with probabilistic information about the simulation population. The modeler may create the input models in any EMF-based modeling environment (eclipse.org/modeling/emf/) that supports UML and UML profiles. An example of such a modeling environment is Papyrus (eclipse.org/papyrus/).

In addition to the input models, the modeler needs to configure the simulator's output. Specifically, PoliSim enables one to define, via OCL, any variables, e.g., the revenue, that one would like to derive from the simulation results. Furthermore, the modeler may choose to group the input policy models into two sets: the "original" set and the "modified" set. In such a case, PoliSim independently simulates the two sets (over the same

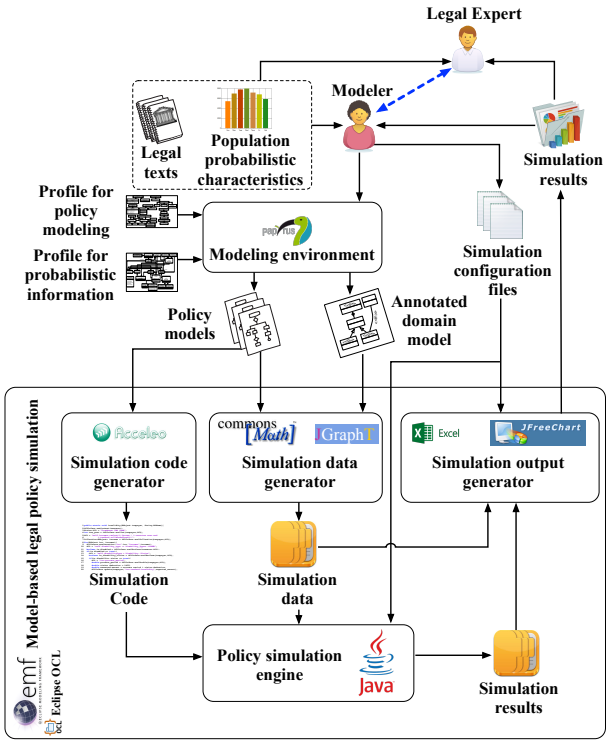


Fig. 11 PoliSim Architecture

input data) and calculates the difference between the variables of interest resulting from the two simulations. This type of analysis is useful for analyzing the impact of policy changes on the variables of interest.

The core components of PoliSim are shown in the rectangle labeled *Model-based legal policy simulation* in Fig. 11. All of these components rely on: (1) the Eclipse Modeling Framework (EMF) for manipulating the input models and the simulation data, and (2) EclipseOCL for parsing and evaluating OCL expressions (eclipse.org/ocl/).

The *Simulation code generator* component uses Acceleo (eclipse.org/acceleo/) for transforming policy models into simulation code (discussed in Section 4). Before running a simulation, PoliSim compiles the generated simulation code. The resulting bytecode is then executed by the *Policy simulation engine* over the generated simulation data.

The component *Simulation code generator* implements the process discussed in Section 6. This component relies mainly on: (1) JGraphT (jgrapht.org) for graph analysis, including topological sorting and cycle detection; and (2) Apache Commons Mathematics Library (commons.apache.org) for generating random values based on given probability distributions. Statistical tools such as R (r-project.org) would provide an alternative to Apache Commons Mathematics Library as used in PoliSim. However, such statistical tools are not a replacement for our data generator. In particular, these tools cannot instantiate object-oriented models as they do not provide a mechanism to handle the instantiation order and the interdependencies between model elements (see Section 6).

The *Simulation output generator* creates the final output from the tool. Fig. 12 presents the output obtained for a (hypothetical) scenario, where the invalidity policy presented in Section 3 has been abolished. As shown by the figure, the output comes in two different formats: spreadsheets and charts. The spreadsheet are generated using Apache POI (poi.apache.org) and the charts – using JFreeChart (jfree.org/jfreechart/). The design and display of the charts are configurable based on the needs and preferences in a given context. For example, the spreadsheet of Fig. 12(a) shows the difference in revenue before and after the invalidity policy abolishment scenario mentioned above. The chart of Fig. 12(b) visualizes the distribution of the granted tax deductions for the same scenario.

Our implementation contains approximately 13K lines of code, excluding comments, third-party libraries, and the automatically-generated simulation code.

8 Evaluation

In this section, we report on a case study where we apply our simulation framework to Luxembourg's Income Tax Law. We investigate through this case study the following Research Questions (RQs):

RQ1: Do data generation and simulation run in practical time? One should be able to generate sufficient 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 practical executions times, given our purpose.

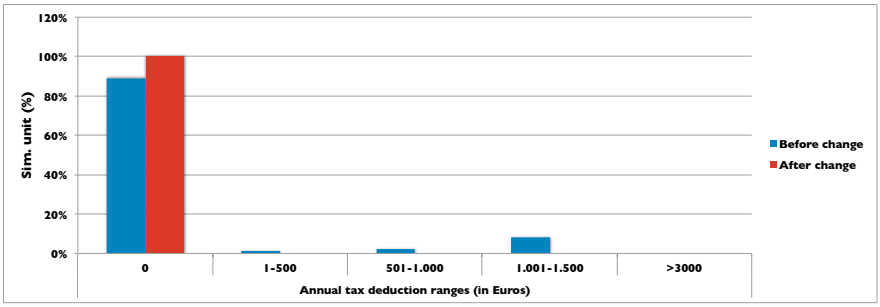
RQ2: Does our data generator produce data that is consistent with the specified characteristics of the population? A basic and yet important requirement for our data generator is that the generated data should be aligned with what is specified via the profile. RQ2 aims to provide confidence that our 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

Sim. unit	Updated features	Deductions/Credits	Old value	New value	Tax class	Income type	Gross income (per year)	Taxable income (per year)	New taxes (per year)	Old taxes (per year)	Loss/ Gain
TaxPayer 1	Income 1	invalidityDeduction (TaxCard)	0	0	One	Employment	14823,83	13150	73	73	0
TaxPayer 2	Income 1	invalidityDeduction (TaxCard)	225	0	Two	Employment	59211,47	52400	4504	4563	+59
TaxPayer 3	Income 1	invalidityDeduction (TaxCard)	1455	0	One	Employment	15666	12450	13	138	+125
TaxPayer 4	Income 1	invalidityDeduction (TaxCard)	0	0	One_A	Other	50958,84	45300	511	511	0
	Income 2	invalidityDeduction (TaxCard)	0	0		Pension	20651,4	20650	0	0	0

•••

(a)



(b)

Fig. 12 Excerpt of Simulation Results Presented as a (a) Spreadsheet and (b) Chart

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. 3. 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 [42].

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. As we reported in our earlier work [42], the six policy models and the domain model were built by the first author who has 6 years of formal training in computer science and 4 years of experience in model-driven engineering. Building the models took on average ≈3.3 person-hours (ph) per policy model and ≈8 ph for the domain model, excluding the effort for validating the models and extending the domain model with probabilistic information.

The policy models and the domain model from our previous work [42] were enhanced to support simulation. These models were then validated with (already-trained) legal experts in a series of meetings totaling ≈12 hours. Subsequently, the first author annotated the (validated) domain model with probabilistic information derived from publicly-available census data provided by STATEC (statistiques.public.lu/). Specifically, from this data, we extracted information about 15 quantities including, among others, age, income, income type, disability types, and household size. The annotation process took ≈10 ph, including the effort spent on extracting the relevant information from census data. The annotations in the partial domain model of Fig. 7 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 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

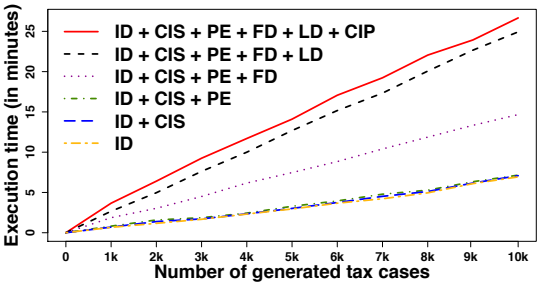


Fig. 13 Execution Times for Data Generation

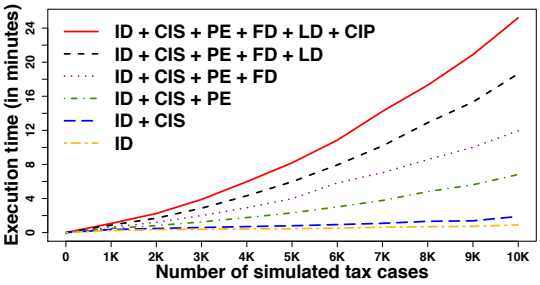


Fig. 14 Execution Times for Simulation

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. 13 and 14, we show the execution times (average of the five runs) for the data generator and for the simulator, respectively.

As suggested by Fig. 13, 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. 13. 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.

The revised simulation data generator presented in this article has a better execution time than its predecessor in our earlier work [43]. Generating 10,000 tax cases covering all six policies takes on average ≈25 minutes with the current generator, while the same task took ≈30 minutes with the previous generator. This improvement in execution time is mainly the result of using a faster strategy for handling unsatisfied multiplicity constraints (discussed in Section 6). In particular, the new strategy no longer performs expensive operations such as (deep) cloning of objects.

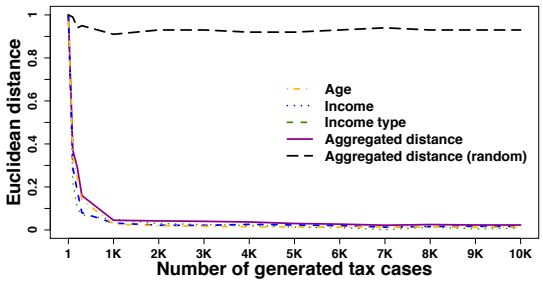


Fig. 15 Euclidean Distances between Generated Data & Real Population Characteristics

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. 14 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 practical, such optimization is warranted only when the execution times need to be further reduced.

As shown by Figs. 13 and 14, generating 10,000 tax cases covering all six policies took ≈25 minutes. Simulating the policies over 10,000 tax cases took ≈24 minutes. These results suggest that our data generator and simulator are highly scalable, noting that the data generator has to be run only once for a given set of policy models, and that the simulation has to be performed once, or in the case where we are analyzing the impact of a particular change, twice.

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 [11]. Fig. 15 presents Euclidean distances for the age, income, and income type histograms as well as

Table 3 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	0.009	0.013	0.013	0.009	0.011	0.008	0.008	0.002	0.001	0.001	0.002	0.002
	p-value	0.99	0.82	0.83	0.8	0.98	0.95	0.99	0.98	1	1	1	1
P_2	D	-	0.012	0.012	0.011	-	0.009	0.006	0.011	-	0.001	0.002	0.004
	p-value	-	0.8	0.8	0.91	-	0.98	0.99	0.93	-	1	1	1
P_3	D	-	-	0.001	0.012	-	-	0.008	0.009	-	-	0.001	0.003
	p-value	-	-	1	0.93	-	-	0.99	0.97	-	-	1	1
P_4	D	-	-	-	0.011	-	-	-	0.009	-	-	-	0.001
	p-value	-	-	-	0.93	-	-	-	0.97	-	-	-	1

the Euclidean distance for the normalized aggregation of the three. As indicated by the figure, the Euclidean distance for the aggregation converges to a very small value (< 0.05) for 1000 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 five criteria considered.

In comparison, when our previous data generator [43] was applied, the Euclidean distance for the aggregation converged only for 2000 or more tax cases. Achieving the same level of quality with a smaller generated data sample is primarily the result of using dynamic updates in our revised data generator (see Section 6).

Our 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 (1,000) to reach a high level of data quality.

RQ3. We answer RQ3 using the Kolmogorov-Smirnov (KS) test [12], 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, 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 3 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 0.013 and the minimum p-value is 0.8. The KS tests thus give no counter-evidence for the samples being from different populations. We note that there are other statistical tests that can be used as alternatives or complements to the KS test, e.g., the Anderson-Darling test and the Cramér-von-Mises criterion [12]. In our evaluation, we did not employ additional tests given the clear results obtained with the KS test ($p > 0.8$ everywhere).

In our previous data generator, the maximum D was 0.017 and the minimum p-value was 0.39. This suggests that the revised data generator is more likely to produce consistent simulation data across several runs than its predecessor. This improvement results from the revised data generator accounting for the distribution of already-generated data in order to dynamically guide the generation of the remaining data (see Section 6).

The results in Table 3 provide confidence that our data generator yields consistent data across different runs.

9 Limitations and Threats to Validity

In this section, we describe the limitations of our simulation framework and discuss the validity threats that are pertinent to our evaluation in Section 8.

9.1 Limitations

We explain the limitations of our framework across three dimensions: our profile for expressing probabilistic characteristics, our simulation data generation algorithm, and our tool support.

Profile. The UML profile of Section 5 was designed with the goal of expressing a static snapshot of the simulation population. This profile cannot capture the dynamic evolution of probabilistic quantities, e.g., the evolution of the distributions of taxpayers' ages over the next decade. Consequently, our current profile will need to be further enhanced if it is to be used for dynamic simulation, e.g., time-series simulation.

Simulation Data Generation. 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 [26] 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 argued in Section 8, our data generation strategy meets the above scalability requirement. However, the strategy has limitations: (1) As noted in Section 6.2, the strategy works only when cyclic OCL dependencies between classes and attributes 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 be left unsatisfied if appropriate objects are not generated over the future course of data generation. (3) To avoid infinite loops, the strategy allows the traversal of cyclic association paths only for a bounded number of times. As a consequence, multiplicities on cyclic association paths may not be satisfied, and further unsatisfiable multiplicity constraints will go undetected. (4) The strategy does not guarantee that constraints other than those specified in our profile will be satisfied.

With regard to the implications of the above limitations, we do not anticipate (1) to pose major difficulties for the modelers, as we surmise that occurrences of cyclic dependencies in the OCL expressions are not frequent in practice. In fact, we did not see any such cyclic dependencies in our case study (described in Section 8). This said, such dependencies, had they been present, would have prompted us to revise the logic behind the stereotypes attached to the domain model.

As for (2), the potential for multiplicity constraint violations exists only when the domain model contains m-to-n or many-to-many associations.[2] Again, we have not seen major problems arising from this limitation in our application context. Specifically, we observed from an examination of the data generated in our case study of Section 8 that less than 1% $(45/10,000)$ of the generated tax cases have unsatisfied multiplicities. Such small levels of inconsistency are unlikely to have a significant negative impact on the overall quality of the generated data. The reason why we could not satisfy all the multiplicity constraints was that the object pool had reached the desired size (and the data generation was thus stopped), while some objects were still waiting for their multiplicity constraints to be satisfied by data to be generated in the future.

For (3), the main consideration is choosing the right bound (i.e., the maximum number of times to traverse cyclic association paths). This bound has to be set based on knowledge about the domain and examining the level of multiplicity constraint violations in the object pool for a chosen bound value. As noted earlier in Section 6.4, we set this bound to 10. The small level of multiplicity constraint violations noted above suggests that this bound is adequate for our domain. Larger bounds may be required for other applications.

Finally and with regard to (4), although we cannot directly account for additional constraints, we can use the stereotypes in our profile to guide the data generator towards satisfying additional constraints. For example, for the policy model of Fig. 3, we need to ensure that the disability rate is zero when a taxpayer is not disabled, and within the range $\frac{1}{20}..1$ when a taxpayer has a vision disability (as stated in the policy description of Fig. 4). We account for these additional constraints during data generation through the *«value dependency»* stereotype applied to the *disabilityRate* attribute of *TaxPayer* (as shown in the domain model of Fig. 7). In our context, we could successfully incorporate such additional constraints into our domain model relying only on our profile. Nevertheless, we acknowledge that a more general and flexible solution is needed for handling additional constraints. We leave the development of such a solution to future work.

Tool Support. In our framework, we narrow the use of the UML notion to what is necessary for our purposes. Our tool support nevertheless does not yet have a custom modeling environment exclusively supporting the UML fragment that is used within our framework. Instead, for model construction, we rely on generic UML modeling environments, e.g. Papyrus. Unless customized according to needs, generic modeling environments can introduce accidental complexity, as modelers are not shielded from the complexity of the entire UML. In the future, we need to provide a simplified modeling environment for our framework, potentially by customizing a generic modeling environment and hiding all the notational elements, functions and features that are not used by our framework. A related limitation in our tool support is that, currently, the users are directly exposed to our profile for specifying probabilistic quantities. A

[2] We note that the traversal strategy of Alg. 1, discussed in Section 6.4, ensures that associations that have an end with a cardinality of 1, 0..1, 1..n or 1..$*$ will satisfy the multiplicity constraints. We further note that the satisfaction of many-to-many multiplicity constraints is a given. Nevertheless, our data generator interprets many-to-many multiplicity constraints as m-to-n ones, with m and n chosen either randomly or by the *«multiplicity»* stereotype.

streamlined user interface is required for enabling the specification of these quantities in a simpler manner and without direct exposure to the underlying profile.

9.2 Threats to Validity

Construct and external validity are the most relevant validity considerations for our evaluation of Section 8. Below, we address these two dimensions of validity.

Construct validity. We did not have access to real tax data for our evaluation. Consequently, when measuring how closely our generated data represented real data (RQ2 in Section 8), we considered only those characteristics of the underlying population for which we had explicit statistical data. Had we had access to real data, we may have been able to derive further characteristics by establishing, e.g. through regression analysis, further relationships among different quantities. To increase construct validity for measuring the representativeness of the generated data, it is important to conduct in the future case studies where real data is available.

External validity. Thus far, we have applied our simulation framework to tax policies only. Additional case studies in other legal domains are necessary for increasing external validity. In particular, and as we discussed earlier in the article, our framework is targeted primarily at prescriptive laws. A thorough investigation needs to be conducted for determining the extent to which our framework is useful for laws that are more declarative in nature, e.g., data protection and privacy laws.

10 Related Work

In this section, we describe and compare with several strands of related work. We organize our discussion along four areas: (1) modeling of legal policies, (2) simulation of legal policies, (3) model execution and simulation in the broader context of software engineering, and (4) model instance generation.

Modeling of legal policies. Modeling legal policies has been a subject of study in the Artificial Intelligence (AI) community for more than 50 years [3]. For example, Rissland and Skal [37] combine example-based and rule-based reasoning to mimic the reasoning of legal experts in tax court cases; Melz and Valente [28] build a domain-specific ontology for the US internal revenue code and discuss the application of this ontology for building a query assistance system for taxation regulations. AI representations of legal policies are generally aimed at performing expert search and question answering. In contrast, our policy models are primarily aimed at simulation and testing.

Legal policies have also been studied in the software engineering community. Breaux and Antón [6] develop a rule-based framework for modeling rights and obligations from the US Health Insurance Portability and Accountability Act (HIPAA). Breaux and Powers [7] derive business process models from the clauses in HIPAA. Ghanavati et al. [16] use a combination of goal models and use cases for specifying legal requirements in the healthcare domain. Islam et al. [25] use UMLsec [27] –a UML profile for security– for modeling security requirements imposed by regulations. van Engers et al. [48] propose a UML-based methodology that was applied to modeling the Dutch tax legislation and support the implementation of a new system. Nevertheless, none of the approaches discussed above are meant at expressing (procedural) legal policies in an executable representation. In contrast, our approach provides executable semantics for legal policies which is a fundamental prerequisite for simulation.

Finally, legal ontologies have been proposed as a mechanism for characterizing legal domains and the information that needs to be captured in legal policies [47]. As we explained in Section 3, our modeling methodology is based on a field study through which we developed a metamodel (analogous to an ontology) for the information requirements that legal policies need to meet. Our metamodel most closely relates to the Functional Ontology of Law (FOL) [8] – an abstract ontology with a functional perspective on legal knowledge. In particular, FOL envisages a *reactive knowledge* category to enable the specification of procedures that need be executed in response to certain events. Our legal policies can be viewed as instantiations of this category in FOL.

Simulation of legal policies. Several policy simulation tools exist in the field of applied economics. For example, SYSIFF [4, 10] in France, SPSD/M [44] in Canada, and POLIMOD [45] in the UK have been used in tax-benefit simulation for several years. Most recently and at a European level, EUROMOD [15] has been developed to provide a tax-benefit simulation infrastructure for the EU member states. In addition to the above tools, generic statistical workbenches such as SAS [39] have been customized for use in policy simulation and prediction [40].

All the above tools use a combination of spreadsheets, hard-coded formulae, and programming languages for implementing legal policies. As we argued in Section 1, this complicates the validation of the resulting policies with legal experts. Our framework aims at addressing this issue by raising the level of abstraction at which legal policies are specified. Furthermore, the above tools typically assume that historical data is available for simulation. In contrast, our framework provides a built-in

data generator that can produce artificial but realistic simulation data based on historical aggregate distributions and/or expert estimates.

Model execution and simulation. The execution semantics of activity diagrams in our work is based on a combination of Java and OCL. An alternative for the execution semantics would be fUML [34], e.g., as used by Mijatov et al. [29] for executing activity diagrams during testing. Currently, fUML does not support UML profiles and stereotype-specific semantics. Since our simulation framework relies on profiles, Java and OCL provide a more straightforward and effective basis for executing our legal policy models.

There are a number of plugin tools for existing modeling environments that support the simulation of UML behavioral models. These tools include, among others, IBM Rational Software Architect Simulation Toolkit [22], IBM Rhapsody Simulation Toolkit [23], Papyrus' Moka [36], and MagicDraw's Cameo [31]. These tools are mainly targeted at the simulation of system designs and architectures, and are not readily applicable to legal policies.

Model instance generation. Automated instantiation of (meta-)models is useful in many situations, e.g., during testing [24] and system configuration [2]. Several instance generation approaches are based on exhaustive search, using tools such as Alloy [26] 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 9, do not scale well in our application context.

Another class of instance generation approaches rely on non-exhaustive techniques, e.g., graph-based rules [17], metaheuristic search [1], mutation analysis [14], and model cloning [5]. 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.

Generating very large model instances has been addressed before by Mougenot et al. [30] and Hartmann et al. [19]. Mougenot et al. are motivated by building models that are large enough to be used for evaluating the scalability of automated analysis techniques such as consistency checking. To this end, Mougenot et al. propose a randomized method for generating model instances with linear complexity in the size of the generated models. This method, in contrast to our data generator, is not meant at creating data that is representative of a certain population. Furthermore, the method is restricted to tree structures and does not provide fine-grained control over the instantiation of attributes and associations. With regard to the work of Hartmann et al., the underlying motivation is the simulation of smart grids. This work uses hard-coded rules, derived from a field study of smart grid applications, for guiding data generation. In contrast, our data generator is parameterized and guided by a generic UML profile for probabilistic information.

11 Conclusion

In this article, we proposed a model-based framework and associated tool support for legal policy simulation. The framework includes an automated data generator. The main 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 need to perform user studies in order to evaluate the usability of our modeling approach and to measure the effort required for defining new policies. Another area of future work is adapting our framework to work with the business process modeling notation (BPMN). Such an adaptation will make it possible to apply our framework in the broader context of business processes. With regard to data generation, we would like to investigate in the future whether our probabilistic approach can be enhanced with constraint solving capabilities, e.g., via metaheuristic search, in order to support additional constraints. We further intend to conduct a more detailed evaluation to examine the overall accuracy of our simulation framework. To do so, we need to validate the generated data and the simulation results 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. Ali, S., Iqbal, M., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Transactions on Software Engineering **39**(10), 1376–1402 (2013)

2. Behjati, R., Nejati, S., Briand, L.: Architecture-level configuration of large-scale embedded software systems. ACM Transactions on Software Engineering and Methodology **23**(3), 25 (2014)
3. Bench-Capon et al., T.: A history of AI and Law in 50 papers: 25 years of the International Conference on AI and Law. Artificial Intelligence and Law **20**(3), 215–319 (2012)
4. Bourguignon, F., Chiappori, P., Sastre, J.: SYSIFF: a simulation program of the french tax-benefit system. Tax Benefit Models **10** (1988)
5. Bousse, E., Combemale, B., Baudry, B.: Scalable armies of model clones through data sharing. In: Proceedings of the 17th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'14), pp. 286–301 (2014)
6. Breaux, T., Anton, A.: Analyzing regulatory rules for privacy and security requirements. IEEE Transactions on Software Engineering **34**(1), 5–20 (2008)
7. Breaux, T.D., Powers, C.: Early studies in acquiring evidentiary, reusable business process models from laws for legal compliance. In: Proceedings of the 6th IEEE International Conference on Information Technology: New Generations (ITNG'09), pp. 272–277 (2009)
8. Breuker, J., Valente, A., Winkels, R., et al.: Legal ontologies: a functional view. In: Proceedings of the 1st LegOut Workshop on Legal Ontologies, pp. 23–36. Citeseer (1997)
9. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**, 1–23 (2014)
10. Canova, L., Piccoli, L., Spadaro, A.: SYSIFF 2006: A microsimulation model for the French tax system. Tech. rep., MicroSimula - Paris School of Economics (2009)
11. Cha, S.H.: Comprehensive survey on distance/similarity measures between probability density functions. International Journal of Mathematical Models and Methods in Applied Sciences **1**(3), 300–307 (2007)
12. Corder, G.W., Foreman, D.: Nonparametric Statistics: A Step-by-Step Approach. John Wiley & Sons (2014)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
14. Di Nardo, D., Pastore, F., Briand, L.C.: Generating complex and faulty test data through model-based mutation analysis. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15), pp. 1–10 (2015)
15. Figari, F., Paulus, A., Sutherland, H.: Microsimulation and policy analysis. Handbook of Income Distribution **2** (2014)
16. Ghanavati, S., Amyot, D., Peyton, L.: Towards a framework for tracking legal compliance in healthcare. In: Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE'07), pp. 218–232 (2007)
17. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software & Systems Modeling **4**(4), 386–398 (2005)
18. Object Management Group: UML 2.2 Superstructure Specification (2009)
19. Hartmann et al., T.: Generating realistic smart grid communication topologies based on real-data. In: Proceedings of the 5th IEEE International Conference on Smart Grid Communications (SmartGridComm'14), pp. 428–433 (2014)
20. Hermans, F., Pinzger, M., van Deursen, A.: Detecting and refactoring code smells in spreadsheet formulas. Empirical Software Engineering **20**(2), 549–575 (2015)
21. Hermans, F., Pinzger, M., van Deursen, A.: Detecting and visualizing inter-worksheet smells in spreadsheets. In: Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE'12), pp. 441–451 (2012)
22. IBM: IBM Rational Software Architect Simulation Toolkit. `http://www-03.ibm.com/software/products/en/ratisoftarchsimutool`
23. IBM: Rational Rhapsody Designer for Systems Engineers. `http://www-03.ibm.com/software/products/en/ratirhapdesiforsystengi`
24. Iqbal, M., Arcuri, A., Briand, L.: Environment modeling and simulation for automated testing of soft real-time embedded software. Software & Systems Modeling **14**(1), 483–524 (2013)
25. Islam, S., Mouratidis, H., Jürjens, J.: A framework to support alignment of secure software engineering with legal regulations. Software & Systems Modeling **10**(3), 369–394 (2011)
26. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
27. Jürjens, J.: Umlsec: Extending UML for secure systems development. In: Proceedings of the 5th International Conference on the Unified Modeling Language (UML'02), pp. 412–425 (2002)
28. Melz, E., Valente, A.: Modeling the tax code. In: Proceedings of the 2nd International Workshop on Regulatory Ontologies (WORM'04), pp. 652–661 (2004)
29. Mijatov, S., Mayerhofer, T., Langer, P., Kappel, G.: Testing functional requirements in UML activity diagrams. In: Proceedings of the 9th International Conference on Tests and Proofs (TAP'15), pp. 173–190 (2015)
30. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09), pp. 130–145 (2009)
31. No Magic: Cameo Simulation Toolkit. `http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`
32. Object Management Group: The Unified Modelling Language. Version 2.2 . `http://www.omg.org/spec/UML/2.1.2/` (2007)
33. OMG: Modeling and Analysis of Real-time and Embedded Systems (MARTE), version 1.1 (2011). `http://www.omg.org/spec/MARTE/1.1/`
34. OMG: Semantics of a foundational subset for executable UML models (fUML), version 1.1 (2013). `http://www.omg.org/spec/FUML/1.1`
35. Panko, R.: What we know about spreadsheet errors. Journal of End User Computing **10**, 15–21 (1998)
36. Papyrus: Moka overview. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`
37. Rissland, E., Skalak, D.: CABARET: Rule interpretation in a hybrid architecture. International Journal of Man-Machine Studies **34**(6), 839–887 (1991)
38. Ruiter, D.: Institutional legal facts: legal powers and their effects, vol. 18. Springer Science & Business Media (1993)
39. SAS Institute: Statistical Analysis System (SAS). `http://www.sas.com/`
40. SAS Institute: Statistical Analysis System (SAS) for Econometrics and Time Series Analysis (ETS). `https://support.sas.com/documentation/onlinedoc/ets/`
41. Soltana, G., Fourneret, E., Adedjouma, M., Sabetzadeh, M., Briand, L.C.: Using UML for modeling legal rules: Supplementary material. Tech. Rep. TR-SnT-2014-3, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg (2014). `http://people.svv.lu/soltana/Models14.pdf`
42. Soltana, G., Fourneret, E., Adedjouma, M., Sabetzadeh, M., Briand, L.C.: Using UML for modeling procedural legal rules: Approach and a study of Luxembourg's Tax Law. In: Proceedings of the 17th ACM/IEEE International Conference on

Model-Driven Engineering Languages and Systems (MODELS'14), pp. 450–466 (2014)
43. Soltana, G., Sannier, N., Sabetzadeh, M., Briand, L.C.: A model-based framework for probabilistic simulation of legal policies. In: Proceedings of the 18th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'15) (2015)
44. Statistics Canada: The Social Policy Simulation Database and Model (SPSD/M). `http://www.statcan.gc.ca/eng/microsimulation/spsdm/spsdm`
45. Sutherland, H.: The development of tax-benefit models: a view from the UK. Tech. rep., Faculty of Economics, University of Cambridge (1995)
46. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers (2007)
47. Van Engers, T., Boer, A., Breuker, J., Valente, A., Winkels, R.: Digital Government: E-Government Research, Case Studies, and Implementation, chap. Ontologies in the Legal Domain, pp. 233–261. Springer US (2008)
48. van Engers, T., Gerrits, R., Boekenoogen, M., Glassée, E., Kordelaar, P.: POWER: using UML/OCL for modeling legislation - an application report. In: Proceedings of the 8th International Conference on Artificial Intelligence and Law (ICAIL'01), pp. 157–167 (2001)

A Appendix

A.1 Transformation of Legal Policies to Java: Detailed Algorithm

The core algorithm for transforming legal policies to simulation code, named PMToJava, is presented in Alg. 4. This algorithm takes as input a Policy Model, PM, an element $\in \mathcal{PM}$, the set of $\mathcal{PM}$'s inputs, $inputs\_all$, and the set of inputs that have been already declared, $inputs\_declared$. Note that, we assume that $\mathcal{PM}$ uses only deterministic decisions as a modeling restriction (since loops are handled using expansion regions). The transformation is based on a set of predefined patterns. These patterns are detailed in A.2. The transformation process is recursive and mimics a depth-first traversal of the underlying graph of $\mathcal{PM}$. There are four main parts to this process: (1) pattern recognition (Alg. 4, L. 4-7); (2) input declarations (Alg. 4, L. 8-14); (3) transformation of all elements other than decisions (Alg. 4, L. 15-28). Within this class of elements, additional processing is necessary for expansion regions to propagate their output if they have any (Alg. 4, L. 25-27); and (4) transformation of decision nodes (Alg. 4, L. 29-36).

Main (Alg. 5) initializes the transformation's variables and makes the first call to PMToJava over the input policy model $\mathcal{PM}$. During its execution, PMToJava makes calls to Algs. 4-8. retrieveDependentInputs (Alg. 6) retrieves all input parameters that need to be declared before transforming a given pattern. getInitialNode (Alg. 7) returns the initial node of an `Activity` or an `ExpansionRegion`. getFlows (Alg. 8) returns, based on the current pattern being processed, the outgoing flows targeting the next elements that should be visited. operatorsToJava (Alg. 9) transforms the equality operators as well as the conditional operators from OCL to Java. recognizePattern (Alg. 10) identifies and creates the appropriate pattern that matches the shape formed by: the visited element, its stereotype, its container (`Activity` or `ExpansionRegion`) and its neighborhood (adjacent elements).

Alg. 4: PMToJava

Inputs : (1) A policy model, $\mathcal{PM}$. (2) An element $\in \mathcal{PM}$. (3) The set of $\mathcal{PM}$'s inputs, $inputs\_all$. (4) The set of declared inputs, $inputs\_declared$.
Output : The generated Java code, result.

```
1  result ← ''
2  if (element is NULL) then
3  |  return result
4  Let P be the pattern that should be applied to element.
5  P ← recognizePattern(PM, element)
6  if (P is NULL) then
7  |  return result
8  Let inputs be the non-declared inputs required by P.
9  Let inputs_dependent a set of inputs required by inputs (inputs_dependent ← {}).
10 foreach (input_i) do
11 |  result ← result + PMToJava(PM, input_i, inputs_all, inputs_declared)
12 |  inputs_dependent ← inputs_dependent ∪ retrieveDependentInputs(PM, input_i, inputs_all, inputs_declared, {})
13 Let inputs_declared_save be a copy of inputs_declared.
14 inputs_declared ← inputs_declared ∪ inputs ∪ inputs_dependent
15 if (element is not a DecisionNode) then
16 |  Let st_1 be the opening Java fragment obtained from applying P (See Appendix A.2).
17 |  Let st_2 be the closing Java fragment obtained from applying P (See Appendix A.2).
18 |  Let next be the set of the next elements that have to be visited.
19 |  if (element is an ExpansionRegion or element is an Activity) then
20 |  |  next ← {getInitialNode(element)}
21 |  else
22 |  |  next ← {getFlows(P).target}
23 |  foreach (next_i ∈ next) do
24 |  |  result ← result + st_1 + PMToJava(PM, next_i, inputs_all, inputs_declared) + st_2
25 |  if (P is an Expansion Region With Output Pattern) then
26 |  |  Let out denote the output element of P.
27 |  |  result ← result + PMToJava(PM, out, inputs_all, inputs_declared_save)
28 else
29 |  Let f_1, ..., f_m be the outgoing flows from element.
30 |  if (m ≥ 1) then
31 |  |  foreach (f_i | i ∈ [1..m]) do
32 |  |  |  if (i = 1) then
33 |  |  |  |  result ← result + 'if(' + operatorsToJava(element.name) + '==' + operatorsToJava(f_i.name) + '){' +
   |  |  |  |  PMToJava(PM, f_i.target, inputs_all, inputs_declared)
34 |  |  |  else
35 |  |  |  |  result ← result + '} else {if(' + operatorsToJava(element.name) + '==' + operatorsToJava(f_i.name) + '){' +
   |  |  |  |  PMToJava(PM, f_i.target, inputs_all, inputs_declared)
   |  |  |
36 |  |  result ← result + '} else {System.err.println("Unhandeled situation!");}' + '}' + ... + '}'   (m − 1 times)
37 return result
```

Alg. 5: Main

Inputs : A policy model, $\mathcal{PM}$.
Output : A file containing the Java simulation code resulting from $\mathcal{PM}$'s transformation.

if *($\mathcal{PM}$.root is* ***not*** `NULL`*)* **then**
  Let *inputs_all* be the set of parameter inputs modeled in $\mathcal{PM}$.
  Let *inputs_declared* be the set of declared inputs in $\mathcal{PM}$ (*inputs_declared* = {}).
  Let file be the Java output file of $\mathcal{PM}$.
  write(**PMToJava**($\mathcal{PM}$, $\mathcal{PM}$.*root*, *inputs_all*, *inputs_declared*), file)

Alg. 6: retrieveDependentInputs

Inputs : (1) A policy Model, $\mathcal{PM}$. (2) An input element $\in \mathcal{PM}$.
(3) The set of $\mathcal{PM}$'s inputs, *inputs_all*. (4) The set of declared inputs, *inputs_declared*. (5) The set of visited inputs, *inputs_visited*.
Output : The set of inputs that element depends on, *inputs_dependent*.

if *(element* $\in$ *inputs_visited)* **then return** {}
else
  if *(*element *is* `NULL`*)* **then return** {}
  else *inputs_visited* $\leftarrow$ {element}
inputs_dependent $\leftarrow$ {}
Let *inputs* be the non-declared inputs required by element.
foreach *($input_i \in inputs$)* **do**
  inputs_dependent $\leftarrow$
  inputs_dependent $\cup$ **retrieveDependentInputs**($\mathcal{PM}, input_i, inputs\_all, inputs\_declared \cup \{input_i\}, inputs\_visited$)
return *inputs_dependent*

Alg. 7: getInitialNode

Inputs : (1) A UML element.
Output : The initial node contained by element.

if *(*element *is an* `ExpansionRegion`*)* **then**
  return element.*oclAsType(ExpansionRegion).owned->select(oclIsTypeOf(InitalNode))->first()*
else
  if *(*element *is an* `Activity`*)* **then**
    return element.*oclAsType(Activity).owned->select(oclIsTypeOf(InitalNode))->first()*
  else
    return `NULL`

Alg. 8: getFlows

Inputs : (1) A pattern P.
Output : The outgoing flows from P (can be `NULL`).

if *(P is an* `Intermediate Value Pattern`*)* **then**
  return *P.element.outFlows->select(f targeting a* `CentralBufferNode`).target.outFlows
else
  return *P.element.outFlows*

Alg. 9: operatorsToJava

Inputs : (1) A String expression S (containing logical operators written in OCL).
Output : A String expression where OCL logical operators are mapped to Java's logical operators.

return *S.replace(' = ', ' == ')*
.replace(' <> ', ' != ')
.replaceIgnoreCase(' AND ', ' && ')
.replaceIgnoreCase(' OR ', ' || ')
.replaceIgnoreCase(' NOT ', ' ! ')
.replaceIgnoreCase(' XOR ', ' ^ ')

Alg. 10: recognizePattern

Inputs : (1) A policy model, $\mathcal{PM}$. (2) An element $\in \mathcal{PM}$.
Output : The appropriate pattern, P, corresponding to element.

```
1  switch (TypeOf(element)) do
2      case InitialNode:
3          return new Pattern(element,'Initial Node Pattern')
4      case FinalNode:
5          if (element.owner is an Expansion Region) then
6              return new Pattern(element,'Final Node Inside Expansion Pattern')
7          else
8              return new Pattern(element,'Final Node Pattern')
9      case Policy Model Root:
10         return new Pattern(element,'Policy Pattern')
11     case Input:
12         if (element has stereotype «fromagent») then
13             return new Pattern(element,'FromAgent Input Pattern')
14         if (element has stereotype «fromlaw») then
15             if (element has an OCL expression) then
16                 return new Pattern(element,'FromLaw Variable Input Pattern')
17             else
18                 return new Pattern(element,'FromLaw Constant Input Pattern')
19         if (element has stereotype «fromrecord») then
20             return new Pattern(element,'FromRecord Input Pattern')
21         if (element has stereotype «temporary») then
22             return new Pattern(element,'Temporary Input Pattern')
23     case OpaqueAction:
24         if (element has stereotype «update») then
25             return new Pattern(element,'Update Pattern')
26         if (element has stereotype «assert») then
27             return new Pattern(element,'Assert Pattern')
28         if (element has stereotype «calculate») then
29             if (element.owner is an Expansion Region With Output) then
30                 return new Pattern(element,'Action Inside an Expansion Region With Output Pattern')
31             if ((element.flows->size()) = 1 and (element.flows->first().target is a CentralBufferNode)) then
32                 return new Pattern(element,'Intermediate Value Pattern')
33     case DecisionNode:
34         return new Pattern(element,'Decision Node Pattern')
35     case ExpansionRegion:
36         if (element has an Output) then
37             return new Pattern(element,'Expansion Region With Output Pattern')
38         else
39             return new Pattern(element,'Expansion Region Without Output Pattern')
40     Default: return NULL
```

A.2 Patterns for Transforming Legal Policies to Java Simulaiton Code

Table 4 shows the patterns used by our transformation algorithm PMToJava (Alg. 4 of Appendix A.1) for generating the appropriate Java simulation code (L. 19-20 of Alg. 4). The first column of table 4 provides the name, the generic shape, and a brief description of each pattern. A pattern can be *Elementary*, composed of a single UML element, or *Aggregated*, composed of several UML elements. The second column shows the resulting Java fragments for each pattern. These fragments correspond to the *opening* and *closing* expressions respectively denoted in PMToJava by st_1 and st_2 (L. 19-20 of Alg. 4). The *opening* and *closing* expressions delimit the beginning and the end of a branch or a loop, respectively. Patterns that do not result in the creation of a branch or loop do not require a *closing* expression.

Table 4: Patterns for Transforming Legal Policies to Java Simulation Code

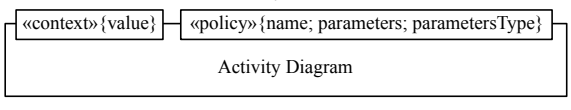
Pattern	Generated Code Fragments
Initial Node Pattern  **Description:** *Initial Node Pattern* is an *Elementary* pattern composed of an InitialNode.	**opening Java fragment**: ' ' **closing Java fragment**: ' '
Final Node Pattern  **Description:** *Final Node Pattern* is an *Elementary* pattern composed of a FinalNode.	**opening Java fragment**: ' return ;' **closing Java fragment**: ' '
Policy Pattern 	**opening Java fragment**: 'public static void '+ name +'('+ **forEach**(parameters$_i$){parametersType$_i$ + ' ' + parameters$_i$} +'){ OCLInJAVA.setContext('+getParameterByContext(value) +'); String OCL="";' **closing Java fragment**: '}'

Table 4: Patterns for Transforming Legal Policies to Java Simulation Code

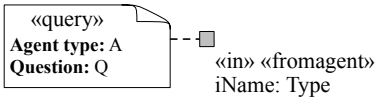
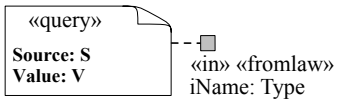
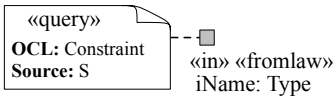
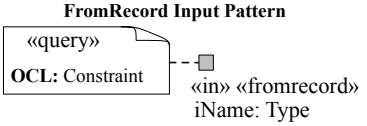
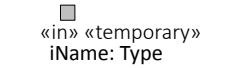
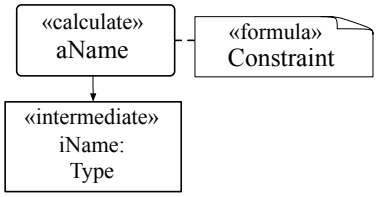
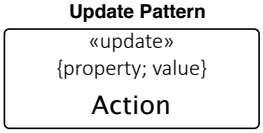
Pattern	Generated Code Fragments
Description: *Policy Pattern* is an *Elementary* pattern is composed of an Activity having *«context»* and *«policy»* stereotypes. This pattern has the highest priority and is always the first pattern to be applied.	
FromAgent Input Pattern  **Description:** *FromAgent Input Pattern* is an *Aggregated* pattern composed of an input having the *«in»* and *«fromagent»* stereotypes, and a comment with the *«query»* stereotype. This kind of input is stored in the instance model and is provided by an agent.	**opening Java fragment**: ′ String OCL = "FromAgent. ′ + **toUpperCase**(iName) + ′";′ + **typeToJava**(Type) + ′ ′ + iName + ′ = ′ OCLInJava.eval′ + **getAbreviationType**(Type) + ′(′ + **getEObjectFromPM**() + ′, OCL);′ **closing Java fragment**: ′ ′
FromLaw Constant Input Pattern  **Description:** *FromLaw Constant Input Pattern* is an *Aggregated* pattern composed of an input having the *«in»* and *«fromlaw»* stereotypes, and a comment with the *«query»* stereotype. This kind of input defines a static information which is defined in the modeled legal text.	**opening Java fragment**: **typeToJava**(Type) + ′ ′ + iName + ′ = ′ V + ′;′ +′ /*TRACEABILITY: ′+ S +′ ∗ /′ **closing Java fragment**: ′ ′
FromLaw Variable Input Pattern  **Description:** *FromLaw Variable Input Pattern* is an *Aggregated* pattern composed of an input having the *«in»* and *«fromlaw»* stereotypes, and a comment with the *«query»* stereotype. The value of such input is defined in the legal text law but the value of such input changes according to the simulated object.	**opening Java fragment**: ′ String OCL = " ′ + Constraint + ′";′ + **typeToJava**(Type) + ′ ′ + iName +′ = ′ OCLInJava.eval′+ **getAbreviationType**(Type) + ′(′ + **getEObjectFromPM**() + ′, OCL);′ +′ /*TRACEABILITY: ′+ S +′ ∗ /′ **closing Java fragment**: ′ ′
FromRecord Input Pattern  **Description:** *FromRecord Input Pattern* is an *Aggregated* pattern composed of an input having the *«in»* and *«fromlaw»* stereotypes, and a comment with the *«query»* stereotype. This kind of input obtains queries the simulates instance object to retrieve the adequate value.	**opening Java fragment**: ′ String OCL = " ′ + Constraint + ′";′ + **typeToJava**(Type) +′ ′ + iName +′ = ′ OCLInJava.eval ′ + **getAbreviationType**(Type) + ′(′ + **getEObjectFromPM**() + ′, OCL);′ **closing Java fragment**: ′ ′
Temporary Input Pattern  **Description:** *Temporary Input Pattern* is an *Elementary* pattern composed of an input having the *«in»* and *«temporary»* stereotypes, and a comment.	**opening Java fragment**: **typeToJava**(Type) +′ ′ + iName + ′ = ′ InitializeByType(Type) + ′;′ **closing Java fragment**: ′ ′
Intermediate Value Pattern  **Description:** *Intermediate Value Pattern* is an *Aggregated* pattern composed of an OpaqueAction with the *«calculate»* stereotype that is: (1) annotated by a Constraint having the *«formula»* stereotype, and (2) linked to a CentralBufferNode having the *«intermediate»* stereotype.	**opening Java fragment**: **typeToJava**(Type) +′ ′ + iName + ′ =′ **operatorsToJava**(Constraint) + ′;′ **closing Java fragment**: ′ ′
Update Pattern 	**opening Java fragment**: **If property is primitive:** property + ′= ′ + value + ′;′ **If property is not primitive:** ′ OCLInJava.update(′ + getPMContext().parameters$_0$ + ′," ′ + value + ′",′ + property + ′);′ **closing Java fragment**: ′ ′

Table 4: Patterns for Transforming Legal Policies to Java Simulation Code

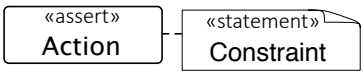
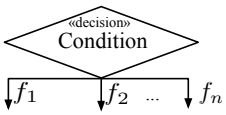
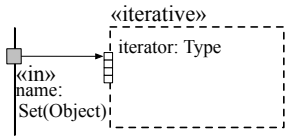
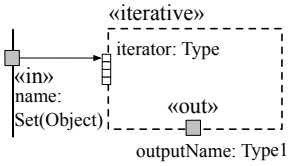
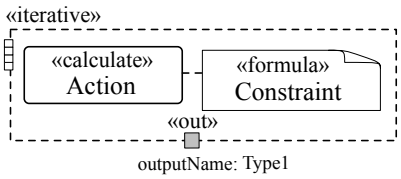
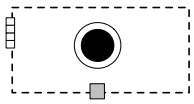
Pattern	Generated Code Fragments
Description: *Update Pattern* is an *Elementary* pattern composed of an OpaqueAction with the *«update»* stereotype.	
Assert Pattern  **Description:** *Assert Pattern* is an *Aggregated* pattern composed of an OpaqueAction with the *«assert»* stereotype annotated by a Constraint having the *«statement»* stereotype. This kind of pattern is used to verify if a given Constraint holds during the simulation. Note that the Constraint must return a boolean value.	**opening Java fragment**: ′ assert(′ + **operatorsToJava**(Constraint) + ′); ′ **closing Java fragment**: ′ ′
Decision Node Pattern  **Description:** *Decision Node Pattern* is an *Elementary* pattern composed of a DecisionNode having the *«decision»* stereotype. The flows were explicitly included to the shape to ease the formalization of the generated Java fragments.	**opening Java fragment for** $F_i = F_1$: ′ if(′ + **operatorsToJava**(Condition) + ′ == ′ + **operatorsToJava**(f_i.name) + ′){′ **opening Java fragment for** $F_i = F_2 \ldots F_n$: ′ } else {if(′+ **operatorsToJava**(Condition) +′) = ′ + **operatorsToJava**(f_i.name) +′{′ **closing Java fragment**: ′} else {System.err.println("Unhandeled situation!");}′+$\overbrace{'}'+\ldots+'}'}^{\text{number of } f-1}$
Expansion Region Without Output Pattern  **Description:** *Expansion Region Without Output Pattern* is an *Aggregated* pattern composed of an input of type Set having the *«in»* stereotype and an ExpansionRegion having an *«iterative»* stereotype. The input is the set over which the ExpansionRegion iterates. Note that the ExpansionRegion does not have an output; otherwise, the transformation will be different.	**opening Java fragment**: ′for(EObject ′ iterator + ′:′ + name + ′){OCLInJava.newIteration("′ + iterator + ′",′ + iterator + ′,"′ + name + ′",′ +name + ′);′ **closing Java fragment**: ′} OCLInJAVA.iterationExit();′
Expansion Region With Output Pattern  **Description:** *Expansion Region Without Output Pattern* is an *Aggregated* pattern composed of: (1) an input of type Set having the *«in»* stereotype, (2) an ExpansionRegion having the *«iterative»* stereotype, and (3) an output having the *«out»* stereotype. The input is the set over which the ExpansionRegion performs a sum of value based on the conditions contained in the ExpansionRegion.	**opening Java fragment**: **typeToJava**(Type1) + ′ ′ + outputName + ′= 0;for(EObject ′ iterator + ′:′ + name + ′){OCLInJava.newIteration("′ + iterator + ′",′ + iterator + ′,"′ + name + ′",′ +name + ′);′ **closing Java fragment**: ′} OCLInJAVA.iterationExit();′
Calculate Inside an Expansion Region With Output Pattern  	**opening Java fragment**: outputName + ′ = ′ + outputName + ′+′ **operatorsToJava**(Constraint) **closing Java fragment**: ′ ′

Table 4: Patterns for Transforming Legal Policies to Java Simulation Code

Pattern	Generated Code Fragments
Description: *Action Inside an Expansion Region With Output Pattern* is an *Aggregated* pattern composed of an OpaqueAction having the *«calculate»* stereotype. This OpaqueAction is annotated by a Constraint having the *«formula»* stereotype. Elements composing this pattern must be inside an ExpansionRegion having an output with the *«out»* stereotype. This pattern allows the formula of a Constraint to be considered in the sum performed by the ExpansionRegion.	
Final Flow Inside an Expansion Region With Output Pattern  *Final Node Inside an Expansion Region With Output Pattern* is an *Aggregated* pattern composed of a FinalFlowNode contained inside an ExpansionRegion having an output with the *«out»* stereotype. This pattern indicates that the value calculated by the ExpansionRegion should not change.	**opening Java fragment**: ' ' **closing Java fragment**: ' '

A.3 Consistency Constraints of the Profile for Expressing the Probabilistic Characteristics of the Simulation Population

Table 5 shows the consistency constraints that check the sound application of our profile (the profile for probabilistic information) on a given domain model. The first column provides a description of the consistency constraints alongside their OCL expressions. The second column lists the stereotypes over which the consistency constraints apply.

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes

Description: This constraint ensures that one and only one stereotype from *«probabilistic value»*'s subtypes is applied to a given element. In other words, the stereotypes *«from distribution»*, *«from histogram»*, *«from barchart»*, and *«fixed value»* cannot be applied simultaneously to a given element (either an attribute or a constraint)
 	
OCL constraint:
 	
`context probabilistic_value inv:`
`let annotatedElement: Element = if(self.base_Constraint.oclIsUndefined()) then self.base_Property else self.base_Constraint endif in`
`annotatedElement.getAppliedStereotypes()->select(s: Stereotype | s.oclIsKindOf(probabilistic_value))->size() = 1`
	«from distribution»
	«from histogram»
	«from barchart»
	«fixed value»

Description: This constraint ensures that a context is specified when a stereotype from *«probabilistic value»*'s subtypes uses OCL.
 	
OCL constraint:
 	
`context probabilistic_value inv:`
`self.usesOCL implies not self.context.oclIsUndefined()`
	«from distribution»
	«from histogram»
	«from barchart»
	«fixed value»

Description: This constraint validates the well-formedness of the parameter of a distribution by ensuring that: (1) the number of parameter names matches the number of parameter values, and (2) the number of parameters required to define the distribution is met. For example, two parameters are required for defining a uniform range distribution (lower and upper bounds). We note that the list of distributions verified by this constraint is not exhaustive. So far, we only needed the discrete distributions above in our work. Nevertheless, this list and the consistency constraint can be extended (if needed) to cover other distributions, e.g., the Poisson distribution. Verifying that the actual inserted parameters are the lower and the upper bounds is assessed by the next constraint.	*«from distribution»*

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes
OCL constraint: **context** from_distribution **inv**: **let** type:Distribution_Type = **self**.type **in** (**self**.parameterNames->**size**() = **self**.parameterValues->**size**()) **and** ((type = Uniform_Range **and self**.parameterNames->**size**() = 2) **or** (type = NormalDistribution **and self**.parameterNames->**size**() = 2) **or** (type = TriangularDistribution **and self**.parameterNames->**size**() = 3) **or** (type = BetaDistribution **and self**.parameterNames->**size**() = 2) **or** (type = GammaDistribution **and self**.parameterNames->**size**() = 2) **or** (type = CauchyDistribution **and self**.parameterNames->**size**() = 2) **or** (type = ChiSquaredDistribution **and self**.parameterNames->**size**() = 1) **or** (type = ConstantRealDistribution **and self**.parameterNames->**size**() = 1) **or** (type = ExponentialDistribution **and self**.parameterNames->**size**() = 1) **or** (type = FDistribution **and self**.parameterNames->**size**() = 2) **or** (type = GumbelDistribution **and self**.parameterNames->**size**() = 2) **or** (type = LevyDistribution **and self**.parameterNames->**size**() = 2) **or** (type = LogisticDistribution **and self**.parameterNames->**size**() = 2) **or** (type = LogNormalDistribution **and self**.parameterNames->**size**() = 2) **or** (type = NakagamiDistribution **and self**.parameterNames->**size**() = 2) **or** (type = ParetoDistribution **and self**.parameterNames->**size**() = 2) **or** (type = TDistribution **and self**.parameterNames->**size**() = 1) **or** (type = WeibullDistribution **and self**.parameterNames->**size**() = 2))	
Description: This constraint ensures that the parameters names used to define a distribution are correct. For example, lower and upper bounds need to be specified when defining a uniform distribution; whereas the mean and the standard deviation need to be specified when defining a normal distribution.	*«from distribution»*

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes

OCL constraint:
 	
`context from_distribution inv:`
`let type:Distribution_Type = self.type in`
`(type = Uniform_Range and self.parameterNames->exists(s:String | s.toLower() = 'lowerbound') and self.parameterNames->exists(s:String| s.toLower() = 'upperbound'))`
`or`
`(type = NormalDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mean') and self.parameterNames->exists(s:String | s.toLower() = 'sd'))`
`or`
`(type = TriangularDistribution and self.parameterNames->exists(s:String | s.toLower() = 'a') and self.parameterNames->exists(s:String | s.toLower() = 'c') and self.parameterNames->exists(s:String | s.toLower() = 'b'))`
`or`
`(type = BetaDistribution and self.parameterNames->exists(s:String | s.toLower() = 'alpha') and self.parameterNames->exists(s:String | s.toLower() = 'beta'))`
`or`
`(type = GammaDistribution and self.parameterNames->exists(s:String | s.toLower() = 'shape') and self.parameterNames->exists(s:String | s.toLower() = 'scale'))`
`or`
`(type = CauchyDistribution and self.parameterNames->exists(s:String | s.toLower() = 'median') and self.parameterNames->exists(s:String | s.toLower() = 'scale'))`
`or`
`(type = ChiSquaredDistribution and self.parameterNames->exists(s:String | s.toLower() = 'degreesoffreedom'))`
`or`
`(type = ConstantRealDistribution and self.parameterNames->exists(s:String | s.toLower() = 'value'))`
`or`
`(type = ExponentialDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mean'))`
`or`
`(type = FDistribution and self.parameterNames->exists(s:String | s.toLower() = 'numeratordegreesoffreedom') and self.parameterNames->exists(s:String | s.toLower() = 'denominatordegreesoffreedom'))`
`or`
`(type = GumbelDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mu') and self.parameterNames->exists(s:String | s.toLower() = 'beta'))`
`or`
`(type = LevyDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mu') and self.parameterNames->exists(s:String | s.toLower() = 'c'))`
`or`
`(type = LogisticDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mu') and self.parameterNames->exists(s:String | s.toLower() = 'upperbound'))`
`or`
`(type = LogNormalDistribution and self.parameterNames->exists(s:String | s.toLower() = 'lowerbound') and self.parameterNames->exists(s:String | s.toLower() = 's'))`
`or`
`(type = NakagamiDistribution and self.parameterNames->exists(s:String | s.toLower() = 'mu') and self.parameterNames->exists(s:String | s.toLower() = 'omega'))`
`or`
`(type = ParetoDistribution and self.parameterNames->exists(s:String | s.toLower() = 'scale') and self.parameterNames->exists(s:String | s.toLower() = 'shape'))`
`or`
`(type = TDistribution and self.parameterNames->exists(s:String | s.toLower() = 'degreesoffreedom'))`
`or`
`(type = WeibullDistribution and self.parameterNames->exists(s:String | s.toLower() = 'alpha') and self.parameterNames->exists(s:String | s.toLower() = 'beta'))`	

Description: This constraint ensures that all parameter values of a given distribution are numeric.
 	
OCL constraint:
 	
`context from_distribution inv:`
`if(self.parameterValues->notEmpty()) then`
`self.parameterValues->forAll(s:String | not s.toReal().oclIsUndefined())`
`else true endif`
	«from distribution»

Description: This constraint ensures that the frequencies specified in a stereotype from *«from chart»*'s subtypes add-up to 1 (or to 100).	*«from histogram»* *«from barchart»*

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes

OCL constraint:
`context from_chart inv:`
`let total:Real = self.frequencies->`
`iterate(s: String ; acc : Real = 0 | acc + s.toReal()) in`
`let tolerance:Real = 0.001 in`
`let diff:Real = if (total > 1 + tolerance) then total - 100 else total - 1 endif in`
`diff.abs() <= tolerance`	

Description: This constraint ensures that the frequencies specified for a stereotype from *«from chart»*'s subtypes are positive.
OCL constraint:
`context from_chart inv:`
`if(self.frequencies->notEmpty()) then`
`self.frequencies-> forAll(s: String | s.toReal() >= 0) else true endif`
	«from histogram»
	«from barchart»

Description: This constraint verifies that the number of frequencies in a *«from histogram»* is equal to the number of bins.
OCL constraint:
`context from_histogram inv:`
`self.frequencies->size() = self.bins->size()`
	«from histogram»

Description: This constraint verifies that the number of frequencies specified for a *«from barchart»* is equal to the number of items.
OCL constraint:
`context from_barchart inv:`
`self.frequencies->size() = self.items->size()`
	«from barchart»

Description: This constraint ensures that the *«multiplicity»* stereotype is only used over associations and attributes that specify a collection of objects. For instance, one cannot use *«multiplicity»* over an attribute that has 1 as cardinality.
OCL constraint:
`context multiplicity inv:`
`let annotatedElement: Element = if(self.base_Association.oclIsUndefined()) then`
`self.base_Property else self.base_Association endif in`
`if(annotatedElement.oclIsTypeOf(Association)) then`
`annotatedElement.oclAsType(Association).memberEnd->at(1).upperBound() <> 1`
`or`
`annotatedElement.oclAsType(Association).memberEnd->at(2).upperBound() <> 1`
`else`
`annotatedElement.oclAsType(Property).upperBound() <> 1 endif`
	«multiplicity»

Description: This constraint verifies that the choice of the targetMember class for a *«multiplicity»* stereotype is correct. For instance, when annotating an association, the targetMember must reference a class from the underlying association's ends; When annotating an attribute the targetMember class must be either the type of the attribute or the class owning the attribute. However, targetMember cannot be the type of the attribute if the attribute is primitive.	*«multiplicity»*

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes

OCL constraint:
`context multiplicity inv:`
`if(not self.targetMember.oclIsUndefined()) then`
`let annotatedElement: Element = if(self.base_Association.oclIsUndefined()) then self.base_Property else self.base_Association endif in`
`if(annotatedElement.oclIsTypeOf(Association)) then`
`annotatedElement.oclAsType(Association).memberEnd->at(1) = self.targetMember`
`or`
`annotatedElement.oclAsType(Association).memberEnd->at(2) = self.targetMember`
`else`
`if(annotatedElement.oclAsType(Property).type.oclIsTypeOf(String) or annotatedElement.oclAsType(Property).type.oclIsTypeOf(Boolean) or annotatedElement.oclAsType(Property).type.oclIsTypeOf(Integer) or annotatedElement.oclAsType(Property).type.oclIsTypeOf(Real)) then`
`self.targetMember = annotatedElement.oclAsType(Property).class`
`else`
`self.targetMember = annotatedElement.oclAsType(Property).class`
`or`
`self.targetMember = annotatedElement.oclAsType(Property).type`
`endif endif else true endif`	

Description: This constraint ensures that containers (constraint) of a *«multiplicity»* stereotype carry some probabilistic information to apply. In other words, constraints used to define a *«multiplicity»* stereotype must be annotated by at least one stereotype from *«probabilistic value»* or *«dependency»*.
OCL constraint:
`context multiplicity inv:`
`if(self.constraints->notEmpty()) then`
`self.constraints->forAll(c: Constraint | c.getAppliedStereotypes()->exists(s: Stereotype | s.oclIsKindOf(probabilistic_value) or s.oclIsKindOf(dependency))) else true endif`
`and`
`if(self.opposites->notEmpty()) then`
`self.opposites->forAll(c: Constraint | c.getAppliedStereotypes()->exists(s: Stereotype | s.oclIsKindOf(probabilistic_value) or s.oclIsKindOf(dependency))) else true endif`
	«multiplicity»

Description: This constraint ensures that the number of reuse probabilities specified in a *«use existing»* stereotype is equal to the number of OCL queries.
OCL constraint:
`context use_existing inv:`
`self.reuseProbabilities->size() = self.queries->size()`
	«use existing»

Description: This constraint ensures that the reuse probabilities specified for a *«use existing»* are positive.
OCL constraint:
`context use_existing inv:`
`if(self.reuseProbabilities->notEmpty()) then self.reuseProbabilities-> forAll(r: Real | r >= 0) else true endif`
	«use existing»

Description: This constraint ensures that OCL queries specified using *«OCL query»* are not empty.
OCL constraint:
`context OCL_query inv:`
`self.expressions->forAll(s : String | s.specification.toString().trim().size() > 0)`
	«OCL query»

Description: This constraint ensures that *«type dependency»* stereotype is not applied on primitive attributes.	*«type dependency»*

Table 5: Consistency Constraints (Profile for Express Probabilistic Characteristics)

Consistency constraint	Involved stereotypes
OCL constraint: **context** type_dependency **inv**: **let** annotatedElement: Element = **if**(**self**.base_Property.**oclIsUndefined**()) **then** **self**.base_Association **else** **self**.base_Property **endif** **in** **if**(annotatedElement.**oclIsTypeOf**(Property)) **then** **if**(annotatedElement.**oclAsType**(Property).type.**oclIsTypeOf**(**String**) **or** annotatedElement.**oclAsType**(Property).type.**oclIsTypeOf**(**Boolean**) **or** annotatedElement.**oclAsType**(Property).type.**oclIsTypeOf**(**Integer**) **or** annotatedElement.**oclAsType**(Property).type.**oclIsTypeOf**(**Real**)) **then** **false** **else** **true** **endif** **else** **true** **endif**	