



PhD-FSTC-2017-47
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defence held on 16/10/2017 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

GHANEM SOLTANA

Born on 11th December 1986 in Tunis, Tunisia

A MODEL-BASED FRAMEWORK FOR LEGAL POLICY SIMULATION AND COMPLIANCE CHECKING

DISSERTATION DEFENCE COMMITTEE

DR. LIONEL C. BRIAND, Dissertation Supervisor
Professor, University of Luxembourg

DR. JACQUES KLEIN, Chairman
Senior Research Scientist, University of Luxembourg

DR. MEHRDAD SABETZADEH, Vice Chairman
Senior Research Scientist, University of Luxembourg

DR. JORDI CABOT, MEMBER
Professor, Open University of Catalonia

Abstract

Information systems implementing requirements from laws and regulations, such as taxes and social benefits, need to be thoroughly verified to demonstrate their compliance. Several Verification and Validation (V&V) techniques, such as reliability or statistical testing, can be used for assessing that such systems meet their legal requirements with a high level of confidence. A common way to do so is through modeling and simulation. Typically, one has to model the expected (legal) behavior of the system in a form that can be executed (simulated), subject the resulting models and the system to the same input data, and then compare the observed behavior of the model simulation and system execution. Another complementary application of modeling and simulation that we explore in this dissertation is enabling impact analysis of legal changes as a way for supporting decision-making. In particular, we observe that the models and tools supporting legal compliance analysis can also be used to accurately anticipate the implications (economic or otherwise) of given changes in the law, e.g., how a change in the tax law is likely to impact the revenue. Nevertheless, in practice and as briefly discussed below, modeling and simulation raise several challenges, especially when applied to systems that are subject to laws and regulations.

Existing V&V techniques often rely on code and complex logical expressions with no intuitive appeal to legal experts for specifying the expected behavior of a given system. Subsequently, one has no practical way to double-check with legal experts that the underlying legal requirements are indeed complete and constitute a faithful representation of what needs to be implemented. Further, manually defining the expected behavior of a system and its test oracles is a tedious and error-prone task. The challenge here is to find a suitable knowledge representation that can be understood by all the involved stakeholders, e.g., software engineers and legal experts, but that remains complete and precise enough to enable automated analysis such as simulation and testing.

As real data is seldom accessible in highly regulated domains, V&V requires the generation of synthetic testing data that can be used to build confidence in the reliability of the system under test. In particular, such data has to be structurally and logically well-formed, not to be discarded by the early sanity checks of the system under test. Further, the data should exhibit as much as possible the actual or anticipated system usage to help mimic how the system would behave under realistic circumstances (reliability testing). Generating such data is not a trivial task as the underlying data schemas are usually large, complex, and subject to numerous domain-related logical constraints.

In this dissertation, we investigate the use of the Unified Modeling Language (UML) and model-driven technologies, e.g., model to code transformations, to facilitate V&V activities for information systems that have to conform to laws and regulations, while tackling the above challenges. We exclusively focus on legal policies (requirements) from *prescriptive* legal frameworks, e.g., frameworks where policies are specified in a procedural and imperative manner, such as tax and customs and excise laws. All our technical solutions have been developed and empirically evaluated in close collaboration with a government administration.

Concretely, the technical solutions presented in this dissertation include:

- **A modeling notation and methodology for formalizing legal policies.** We propose a modeling notation and methodology for building abstract interpretations of the law. Models built using our methodology are simple enough to be understood by the involved stakeholders and are, at the same time, detailed enough to enable automated V&V activities. Our modeling methodology relies on the Unified Modeling Language (UML) and a UML profile for capturing in a precise manner the semantics of (prescriptive) legal policies.
- **A model-based simulation framework.** We develop a model-based framework and associated tool support for simulating legal policies, when formalized using the aforementioned modeling methodology. Simulation provides a comparison baseline of how a compliant system should behave since the model can be validated by legal experts. Further, simulation is a mean to support decision-making when considering legal changes. Specifically, we report in this dissertation on a sizable case study where we assess the anticipated economic implications of a given policy change in Luxembourg's tax law. Our framework relies on model transformation to build the underlying simulation infrastructure (i.e., simulation code and input data) from models. In particular, our framework produces artificial but yet realistic data, i.e., data that is statistically representative of the simulation population. This is made possible through a UML profile that captures the statistical characteristics of the underlying population.
- **A model-based generator of test cases to support reliability testing.** To enable reliability testing of data-intensive systems, e.g., a tax management system, we develop a heuristic approach for generating test cases (data). In addition to being statistically representative of a given population, test cases generated using our approach are also logically valid, i.e., they satisfy a set of well-formedness domain constraints. Instead of starting from scratch, the approach iteratively tweaks simulation data (generated by the simulation framework mentioned earlier) to satisfy validity constraints while maintaining the representativeness of the initial data. Our approach is meant to be scalable and dynamically guides a search-based constraint solver to simultaneously meet validity and representativeness requirements.

Acknowledgements

I would like to state my sincere gratitude to my supervisor, Lionel Briand, for giving me the chance to do such a work within the SVV (Software Verification and Validation) laboratory and for his valuable comments and advice during our meetings. It was an honour to have him as a supervisor and to have had a chance to gain experience while working beside him. This work would not have gone far without his guidance, patience, enthusiasm and encouragement.

I would like to express my infinite gratitude to, Mehrdad Sabetzadeh, who has co-supervised this thesis. I have learned a lot from him both professionally and personally. His care and guidance were very constructive and rich so that they helped me to hone my writing and presentation skills.

I am grateful to the members of my defence committee: Jacques Klein, Jordi Cabot and Richard Paige for dedicating time and effort to review this dissertation. I would like to thank further Jordi Cabot for taking the time to travel to Luxembourg to attend my defence. Unfortunately, Richard Paige could not make it to the defence. Nevertheless, I appreciate his remote participation in the post-presentation discussions, in particular, his suggestions for avenues for future work.

I thank members of Luxembourg's Inland Revenue Office (ACD) and National Centre for Information Technologies (CTIE), particularly Thierry Prommenschenkel, Ludwig Balmer, and Marc Blau for sharing their valuable time and insights with us. In particular, I would like to thank ACD for offering me the opportunity to conduct several case studies in realistic settings during my thesis, and CTIE for its financial and mentoring support. Big thank you to Thierry Prommenschenkel who was always ready to help me understanding complex (French) legal texts.

Last but not least, I would like to thank all my family members who kept encouraging me and supporting me from day one of this academic journey. In particular, I would like to thank my lovely wife and colleague, Ines Hajri, who always believed in me even when I had doubts! Finally, thanks to all my friends and colleagues who made me feel that four years were much shorter than what I initially expected.

Contents

Contents	v
List of Figures	vi
List of Tables	vii
List of Acronyms	ix
1 Introduction	1
1.1 Context	1
1.2 Contributions and Organization	3
2 Background	7
2.1 Modeling Laws and Regulations	7
2.2 Unified Modeling Language (UML)	8
2.3 UML Profiles	9
2.4 Object Constraint Language (OCL)	10
3 Modeling Legal Policies	12
3.1 Field Study of Legal policies	14
3.2 Modeling Methodology	17
3.3 UML Profile for Legal policies	18
3.4 Transforming Legal Policies into OCL	22
3.5 Evaluation	25
3.6 Related Work	27
3.7 Conclusion	28
4 Simulating Legal Policies	30
4.1 Simulation Framework Overview	32
4.2 Background and Running Example	32
4.3 Simulation Code Generation	36
4.4 Expressing Population Characteristics	37
4.5 Simulation Data Generation	41
4.5.1 Domain Model Slicing	41

4.5.2	Identifying a Traversal Order	43
4.5.3	Classifying Path Segments	44
4.5.4	Instantiating the Slice Model	45
4.6	Tool Support	49
4.7	Evaluation	51
4.8	Limitations and Threats to Validity	56
4.8.1	Limitations	56
4.8.2	Threats to Validity	58
4.9	Related Work	58
4.10	Conclusion	60
5	Simulating the Impact of Changes in Legal Policies (Case Study)	61
5.1	Approach	63
5.1.1	Policy Model Example	64
5.1.2	Example of Statistical Guidance for Generating data	66
5.2	Case Study Design	67
5.2.1	Case Selection	67
5.2.2	Data Collection Procedure	68
5.2.3	Analysis Procedure	69
5.3	Results and Discussion	70
5.3.1	RQ1: Can we bring together and model all the information necessary for performing a real-world simulation scenario?	70
5.3.2	RQ2: Are the simulation results credible?	73
5.3.3	Threats to Validity	75
5.4	Lessons Learned	75
5.5	Related Work	77
5.6	Conclusion	78
6	Synthetic Data Generation for Statistical Testing	79
6.1	Background	81
6.2	Approach Overview	82
6.3	Generating Synthetic Data	83
6.3.1	Solving OCL Constraints	85
6.3.2	Generating Valid and Representative Data	87
6.4	Evaluation	92
6.4.1	Research Questions (RQs)	93
6.4.2	Implementation	93
6.4.3	Case Study Description	93
6.4.4	Results and Discussion	94
6.4.5	Threats to Validity	98
6.5	Related Work	98
6.6	Conclusion	99

7 Conclusion	100
7.1 Summary	100
7.2 Future Work	101
List of Papers	103
Bibliography	104
A Details of our UML Profile for Specifying Legal Policies	113
A.1 (Reduced) Activity Diagram Metamodel	113
A.2 Supported Data Types	113
A.3 Consistency Constraints	114
B Activity Diagrams to Text Transformations	122
B.1 Transforming Activity Diagrams to OCL: Detailed Algorithms	122
B.2 Patterns for Transforming Activity Diagrams to OCL	125
B.3 Patterns for Transforming Activity Diagrams to (Java) Simulation Code	128
C Consistency Constraints of the Profile for Expressing Probabilistic Information	131
D Details of the Case Study for Generating Valid and Representative Data	138
D.1 OCL Validity Constraints	138
D.2 Example of Dynamically Generated Corrective Constraints	149
D.2.1 Example of a Corrective Constraint Derived from an Attribute (<i>birth_year</i>)	149
D.2.2 Example of a Corrective Constraint Derived from an Association (<i>household-children</i>)	149
D.2.3 Example of a Corrective Constraint Derived from a Generalization (Types of Tax Cases)	149

List of Figures

1.1	Dissertation Objectives	2
1.2	Dissertation Overview and Organization	4
2.1	A Simple (Excerpt) UML Profile Borrowed from MARTE [Object Management Group, 2011a]	9
2.2	Applying the Stereotypes in Fig. 2.1 on: (a) a Class and (b) an Instance Specification . . .	10
2.3	Example of OCL Expressions Defining (a) an Invariant and (b) a Query Operation . . .	11
3.1	(a) Domain Model for a Legal Article, (b) Procedural Policy for the Article (Expressed in OCL)	13
3.2	Excerpt of Article 105bis from <i>LITL</i> (Translated from French)	15
3.3	Information Model for Legal Policies	16
3.4	Methodology for Specifying Legal Policies	18
3.5	Profile Customizing UML Activity Diagrams for Legal Policy Models	19
3.6	Activity Diagram for Commuting Expenses Deduction (FD)	21
3.7	Generated OCL Expression for the Example of Fig. 3.6 (FD)	23
3.8	Number of AD Elements (Distribution)	25
4.1	Simulation Framework Overview	33
4.2	(Simplified) Policy Model for Calculating Invalidity Deduction	34
4.3	(Simplified) Policy Description for Invalidity Deduction	34
4.4	Fragment of Generated Code for the Model of Fig. 4.2	37
4.5	Profile for Expressing the Probabilistic Characteristics of a Population	38
4.6	Partial Domain Model for Luxembourg’s Income Tax Law Annotated with Probabilistic Information	38
4.7	Example Consistency Constraint for the Profile of Fig. 4.5	41
4.8	Overview of Simulation Data Generation	42
4.9	(a) Excerpt of Slice Model for Simulating the Policy Model of Fig. 4.2, (b) Topological Sorting of Elements in (a)	42
4.10	PoliSim Architecture	50
4.11	Excerpt of Simulation Results Presented as a (a) Spreadsheet and (b) Chart	51
4.12	Execution Times for Data Generation	53
4.13	Execution Times for Simulation	53

4.14	Euclidean Distances between Generated Data & Real Population Characteristics	54
5.1	Overview of our Simulation Approach	63
5.2	Policy Model for Tax Class Categorization (TCC)	65
5.3	Domain Model Fragment Extended with Statistical Guidance	67
5.4	Euclidean Distances between (Selected) Distributions of the Actual Simulation Population and Distributions of the Generated Data Samples	73
5.5	Simulated and Actual Contributions of Households to Revenue	74
5.6	Euclidean Distance Box Plot for Simulation Results	75
6.1	Data Schema (Excerpt) Annotated with Statistical Characteristics	81
6.2	Approach for Generating Valid and Representative Synthetic Data	82
6.3	Overview of our Data Generation Strategy	83
6.4	Intermediate OCL Constraint for Distribution $H1'$ in Table 6.1	91
6.5	Final Corrective OCL Constraint for the Example of Table 6.1	92
6.6	Execution Times for Generating Valid Data Samples of Different Sizes	95
6.7	Distance between Generated Sample and Desired Distributions: (a) Euclidean Distance, (b) Manhattan Distance, and (c) Canberra Distance	97
A.1	Excerpt of UML Metamodel for Modeling Activities	113
A.2	Datatype Library of the Profile for Specifying Legal Policies	114

List of Tables

3.1	Our Profile’s (Non-Abstract) Stereotypes	20
3.2	Comparison of Complexity: Direct Use of OCL vs. OCL Fragments in ADs	27
4.1	Example of Adaptive Adjustment of Frequencies	47
4.2	Pairwise Kolmogorov-Smirnov (KS) Test Applied to Five Samples (P_1, \dots, P_5) of 5000 Tax Cases	55
5.1	Glossary for the Inputs to the Policy Model of Fig. 5.2	66
5.2	Effort for Building and Validating the Models	71
5.3	Statistics from Public Census Data and Governmental Sources	72
5.4	Impact of Amendments on Policy Models	77
6.1	Illustrative Example for Alg. 6	91
6.2	Scenarios for Composing Corrective Constraints	92
6.3	Comparison against the Baseline Solver (RQ1)	94
A.1	Consistency Constraints (Profile for Specifying Legal Policies)	114
B.1	Patterns for Transforming Activity Diagrams to OCL	125
B.2	Patterns for Transforming Legal Policies to Java Simulation Code	129
C.1	Consistency Constraints (Profile for Expressing Probabilistic Characteristics)	131
D.1	Complete List of the Input OCL Validity Constraints	138

List of Algorithms

1	Activity Diagram to OCL (ADToOCL)	23
2	Simulation Data Generator (SDG)	46
3	Class Selector (CS)	47
4	Attribute Value Generator (AVG)	47
5	Process Instance Model (PIM)	88
6	Generate Corrective Constraints (GCC)	90
7	Transform Policy Models to Code (Main)	122
8	Choose the Next Flow to Traverse (getFlow)	122
9	Get First Pattern to Transform into Code (getInitialNode)	123
10	Identifying the Appropriate Pattern to Transform to Code (recognizePattern)	123
11	Get Dependent Variables to Declare (retrieveDependentInputs)	124
12	Detailed Algorithm for Transforming Activity Diagrams into OCL (ADToOCL)	124

List of Acronyms

- ACD** Administration des Contributions Directes.
- AD** UML Activity Diagram.
- CD** UML Class Diagram.
- CFO** Conceptual Frame-based Ontology.
- CTIE** Centre des Technologies de l'Information de l'Etat.
- DAG** Directed Acyclic Graph.
- DSM** Domain-Specific Modeling.
- EMF** Eclipse Modeling Framework.
- FOL** Functional Ontology of Law.
- GT** Grounded Theory.
- IT** Information Technology.
- LITL** Luxembourg's Income Tax Law.
- MDE** Model-Driven Engineering.
- MOF** Meta-Object Facility.
- OCL** Object Constraint Language.
- OMG** Object Management Group.

RQ Research Question.

SMT Satisfiability Modulo Theories.

UML Unified Modeling Language.

V&V Verification and Validation.

Chapter 1

Introduction

1.1 Context

Many laws and regulations, such as taxation and social benefits, often need to be implemented by information systems. Demonstrating that such systems comply with legal requirements is a challenging task. Legal texts often contain ambiguities and software engineers lack the necessary legal background to elicit legal requirements in a precise and complete manner. Any violation or misunderstanding of legal requirements, no matter how small, might cause serious undesirable consequences.

Several computer-assisted analyzers have been proposed for facilitating the validation of legal requirements and the verification of the systems implementing them [Hassan and Logrippo, 2008, Ghanavati et al., 2007, Goedertier and Vanthienen, 2006]. Nevertheless, in practice, building compliant systems is still a challenging task. In particular, we observed a communication gap between software engineers and legal experts. The former are usually not acquainted with the legal jargon; the latter cannot understand complex software artifacts such as source code. As a result, one cannot ensure that system requirements are indeed a faithful and precise interpretation of the law. The first challenge addressed in this dissertation is to devise a modeling methodology to capture a given interpretation of the law in a precise and analyzable form. The resulting models must be, on the one hand, intuitive enough to be understood by both legal experts and software engineers, and on the other hand, detailed and precise enough to support most software Verification and Validation (V&V) activities.

Generally, applying V&V over systems that are subject to legal compliance is challenging due to, among other reasons, the large number of legal scenarios to consider and the unavailability of readily usable data that can support the analysis, e.g., test cases. Therefore, automation is key to perform V&V activities in a practical and scalable manner, especially given that the involved systems are often large and data-intensive, e.g., public administration systems. Fig. 1.2 highlights our global vision of how well-designed models of the law could contribute to facilitate several complex and tedious V&V activities and beyond.

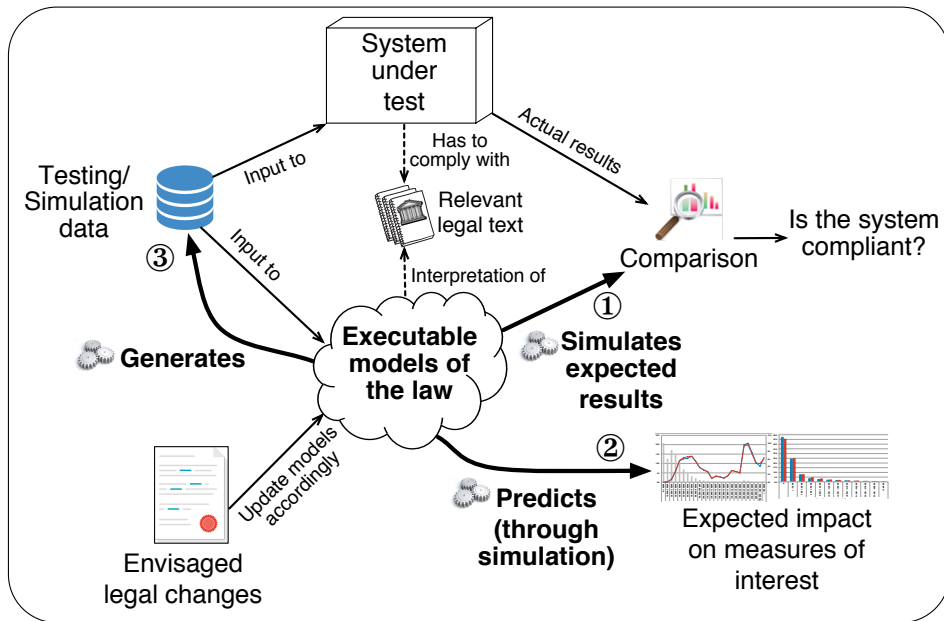


Figure 1.1. Dissertation Objectives

The first V&V activity of interest is legal system compliance verification. Broadly and as shown by Fig. 1.2, our strategy to facilitate such activity is to feed some executable interpretations of the law (models) and the underlying system under test with the same input data. The compliance of the system can be then assessed by comparing, at the level of abstraction of the models, the model simulation results and the system execution results. As shown by the thick arrow denoted by (1) in Fig. 1.2, the core enabler of the above compliance checking strategy is a model simulation engine. In this dissertation, we aim to devise such an engine to mimic (simulate) in a precise manner the expected behavior of the system under test.

Additionally and as shown by the thick arrow denoted by (2) in Fig. 1.2, we rely on the same models and simulation engine that underly automated compliance checking for providing decision-making support to legal experts. In other words, by relying on the same simulation infrastructure, one can accurately anticipate the (economic or otherwise) implications that changing the law would have on the real world. For example, we assist, through a case study, legal experts in quantifying how the state fiscal revenues and the taxpayers' fiscal burden are likely to evolve in response to a given legal change.

Ideally, the two simulation-based activities discussed above should take as input archived or actual data, e.g., population of citizens' tax records, if such data exists. However, in most cases, structure and granularity mismatches can occur between archived data and data needed for testing/simulation. Further, data is often inaccessible due to privacy regulations, e.g., EU's General Data Protection Regulation [General Data Protection Regulation, 2016], making it impossible to share any personal information with third-parties. In the rare cases where actual data happens to be available and at the right level of granularity and precision, anonymization might be an alternative solution, but this often comes at the cost of reduced data quality [De Capitani di Vimercati et al., 2010] and proneness to deanonymization attacks [Al-Azizy et al., 2015].

As shown by the thick arrow denoted by (3) in Fig. 1.2, the last V&V activity of interest concerns test data (case) generation. In particular, we aim to produce data that support system reliability testing (also known as usage-based statistical testing [Runeson and Wohlin, 1995]) from a legal standpoint. The main goal of such testing is to simulate how the system under test behaves under a realistic operational environment. Typically, reliability testing boils down to running the system under test over input data that capture (as much as possible) the expected or actual usage of the system. In this dissertation, we aim to automate the generation of synthetic test data for supporting reliability testing. Among others, such data has to be logically valid to bypass the sanity checks of the system under test and should, at the same time, be statistically representative of the data that would be processed by the system once deployed. Generating such data is challenging as the underlying data structure schemas are usually complex and large and are often subject to numerous domain-related logical constraints.

In a nuanced field such as law, providing adequate support for any kind of automated analysis depends mainly on having a focused scope. This dissertation targets *prescriptive* laws, i.e., highly regulated laws, such as taxation and social benefits, that are defined by a set of legal policies, where a legal policy is a textual definition describing a procedure [Petersen, 2002]. We believe that scoping the research to a particular domain is most appropriate when one is dealing with highly-specialized fields such as law. In this way, one can ensure that the developed solutions will be concrete enough to address a real-world need while avoiding the risk of over-generalization by overlooking the subtleties in different legal jurisdictions.

In summary, this dissertation aims to capitalize on the benefits of modeling to facilitate communication between legal experts and software engineers and provide automation for a number of complex and laborious (mostly V&V) tasks discussed earlier. The core enabling technologies used in this dissertation are the Unified Modeling Language (UML), UML profiles, model transformation and execution, and search-based constraint solving. Most of the research presented in this dissertation has been done in collaboration with: (1) *Centre des Technologies de l'Information de l'Etat* (CTIE), Luxembourg's government computing center, and (2) *Administration des Contributions Directes* (ACD), the Inland Revenue Office of Luxembourg.

1.2 Contributions and Organization

In this dissertation, we develop novel solutions to (1) model policies from prescriptive laws, (2) simulate these policies to determine expected outcomes of legal scenarios and the impact of legal changes on the real world, and (3) generate representative and valid test cases aimed at checking the reliability of the systems implementing these policies. Fig. 1.2 depicts the different solutions developed throughout this dissertation, which are as follows:

- *Modeling Legal Policies*, labelled (1) in Fig. 1.2, incorporates our modeling methodology for specifying legal policies and a customizable model-to-text transformation that converts models built using our methodology to test oracles. Many laws, e.g., those concerning taxes, need

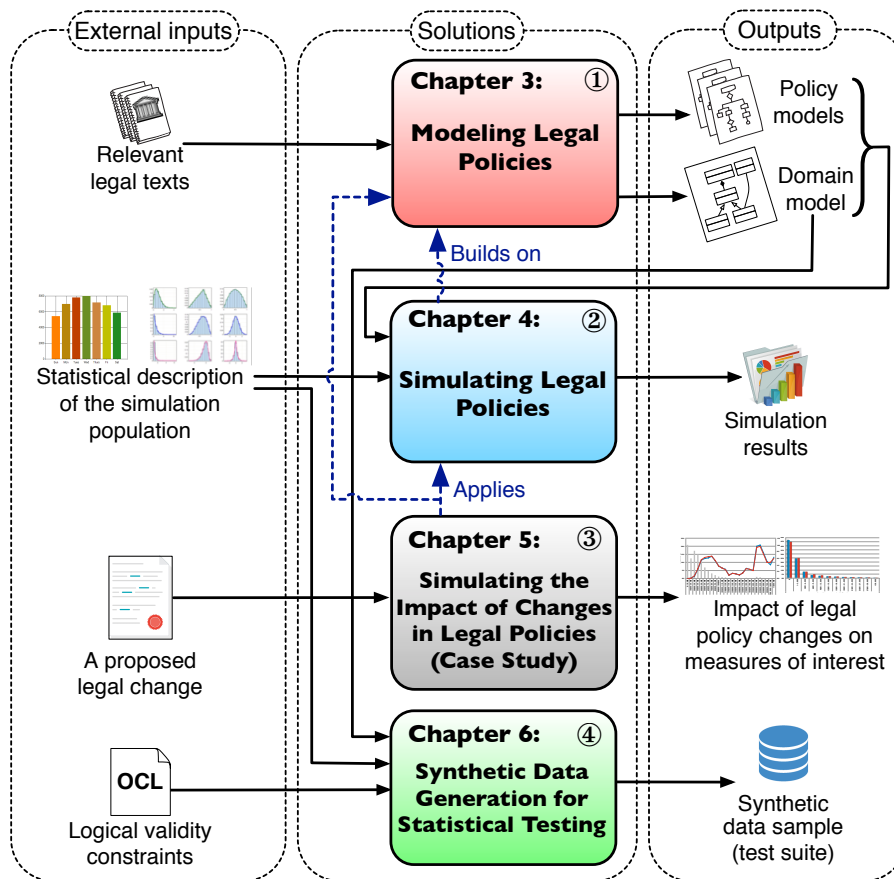


Figure 1.2. Dissertation Overview and Organization

to be operationalized and implemented into public administration procedures and various information systems (e.g., eGovernment applications). We propose a UML-based approach for modeling procedural legal policies. With help from legal experts, we investigate actual legal texts, identifying both the information needs and sources of complexity in the formalization of legal policies. Building on this study, we develop a (first) UML profile that enables more precise modeling of such legal policies. To be able to use logic-based tools for compliance analysis, we automatically transform models of procedural legal rules into the Object Constraint Language (OCL). We report on an application of our approach to Luxembourg’s Income Tax Law providing initial evidence of the feasibility and usefulness of our approach. Our modeling methodology and its underlying model-to-text transformation were published as a conference paper [Soltana et al., 2014] and are covered in Chapter 3.

- *Simulating Legal Policies*, labelled (2) in Fig. 1.2, presents a model-based framework for simulating legal policies that are captured by using our modeling methodology (output of solution labelled with (1) in Fig. 1.2). In addition to checking system compliance, the simulation of legal policies is an important decision-support tool. For example, legal policy simulation can predict how changes in the law affect measures of interest in the real world, e.g., tax revenue, so that changes in legal requirements can be thoroughly assessed before they are implemented. As stated earlier, legal policy simulation is currently implemented using a combination of spread-

sheets and software code. Such an 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. We develop a framework for legal policy simulation that is aimed at addressing the challenges above. This framework automatically derives the simulation infrastructure, i.e., simulation data and code, from legal requirement models. In particular, the framework describes the simulation data via a (second) UML profile that captures the probabilistic characteristics of the underlying simulation population. The framework has an automated mechanism to generate simulation data samples that are aligned with the probabilistic characteristics provided as input. Our model-based simulation framework was initially published as a conference paper [Soltana et al., 2015] and was subsequently extended into a journal paper [Soltana et al., 2016c]. We present our simulation framework in Chapter 4.

- *Simulating the Impact of Changes in Legal Policies (Case Study)*, labelled (3) in Fig. 1.2, reports on a case study where we use the solutions labelled (1) and (2) in Fig. 1.2 for simulating the impact of a real policy change in the taxation domain. Through this case study, we demonstrate how models of legal requirements, on the one hand, facilitate communication between software engineers and legal experts, and on the other hand, are detailed enough to support accurate automated simulation. The simulation scenario we consider is aimed at analyzing the impact of a real tax law reform proposal in Luxembourg. In addition to assisting the government in analyzing potential legal reforms, the case study was a vehicle for assessing the accuracy and credibility of the obtained simulation results when no input data, e.g., historical tax record, is available. We present the case study and further reflect on the lessons learned from this experience in Chapter 5. This contribution was published as a conference paper [Soltana et al., 2016a].
- *Synthetic Data Generation for Statistical Testing*, labelled (4) in Fig. 1.2, provides a novel solution for generating synthetic data (test cases) aimed at supporting usage-based (reliability) testing for data-intensive systems. We note that information systems that must comply with stringent legal requirements are usually very complex, that their exhaustive testing is not possible, and that ensuring their reliability by prioritizing frequent scenarios is a viable strategy in such context. Usage-based statistical testing employs knowledge about the actual or anticipated usage profile of the system under test for estimating system reliability. For many systems, usage-based statistical testing involves generating synthetic test data. Such data must possess the same statistical characteristics as the actual data that the system will process during operation. Synthetic test data must further satisfy any logical validity (OCL) constraints that the actual data is subject to, otherwise the data will often be rejected by the sanity checks of the system under test. Targeting data-intensive systems, e.g., a tax management system, we developed a scalable approach for generating synthetic test data that is both statistically representative and logically valid. The approach works by first taking the simulation data that is generated in the solution labelled (2) in Fig. 1.2, which does not account for logical validity. Then, the approach

tweaks the generated data sample to fix any logical constraint violations. The tweaking process is iterative and continuously guided toward achieving the desired statistical characteristics. We report on a realistic evaluation of the approach, where we generate a synthetic population of citizens' records for testing a public administration IT system. Results suggest that our approach is scalable and capable of simultaneously fulfilling statistical representativeness and logical validity requirements. This work, which is further detailed in Chapter 6, was accepted for publication as a conference paper [Soltana et al., 2017].

Before proceeding to present the dissertation contributions in Chapters 3 through 6, we provide in Chapter 2 the overall background for our work.

Chapter 2

Background

This chapter provides general background information for the dissertation. The content of the chapter is organized under four headings: (1) Modeling Laws and Regulations (Section 2.1), (2) Unified Modeling Language (UML, Section 2.2), (3) UML Profiles (Section 2.3), and (4) Object Constraint Language (OCL, Section 2.4).

2.1 Modeling Laws and Regulations

Regulation modeling is the process of transforming a given law, often written in natural language, into a precise and unambiguous conceptual representation(s). Creating such a representation for a given legal domain requires a conceptualization of the building blocks of the underlying legal knowledge. Typically, one has to create a model that abstracts the target legal domain, or stated differently, a model that ignores irrelevant details and focuses on the aspects of the domain that are necessary to support the pursued activities, e.g., to support system compliance checking.

Ontologies have been widely used as mechanisms for characterizing legal domains (see [Van En-gers et al., 2008]). Some existing ontologies define reusable building blocks for legal knowledge system designs. For example, the Conceptual Frame-based Ontology (CFO) [van Kralingen, 1997] and the Functional Ontology of Law (FOL) [Breuker et al., 1997] define different legal knowledge categories that one should cover when modeling a given law.

Broadly, CFO divides legal knowledge over three distinct categories: *norms*, *acts* and *concept descriptions*. Norms are the general rules, standards and principles stipulated by the law. Acts represent the events and processes that might change the state of the world, e.g., a prison sentence. Concept descriptions deal with the meanings of the legal concepts of the underlying domain. Each entity is defined via a template (also referred to as frames) composed of the relevant attributes for the entity.

In a similar vein, FOL proposes several primitive legal knowledge categories. For example, FOL defines the *responsibility knowledge* for modeling parts of the law that either extend or restrict the responsibility of a person based on its social behavior. Another example of primitive knowledge

category in FOL is the *reactive knowledge* which enables the specification of legal procedures that need to be executed in response to certain events or stimuli.

In this dissertation, we delimited the legal knowledge that needs to be modeled for simulation and test case generation based on a field study that we conducted over prescriptive laws (Section 3.1). Through this field study, we developed a metamodel (analogous to an ontology) for defining the information requirements that legal policies need to cover. Nevertheless, our metamodel also relates to the CFO and FOL ontologies. In particular, the legal policies models built using our modeling methodology (presented in Section 3.5) can be viewed as instantiations of the *acts* and *reactive knowledge* legal knowledge categories in CFO and FOL ontologies, respectively.

2.2 Unified Modeling Language (UML)

The Unified Modeling Language (UML) [Object Management Group, 2015] is probably the most commonly used general purpose modeling language in software engineering. In this dissertation, we tailor UML for modeling laws and regulations. UML is defined and maintained in an open process by the Object Management Group (OMG). In general, UML is often used for specifying, visualizing, constructing, and documenting the artifacts of object-oriented software systems [Larman, 2004]. Further, UML is also a better vehicle for communicating, through models, information regarding the different artifacts of a given software when compared to using natural language for communication.

UML diagrams can be classified into two categories: structural and behavioral diagrams. Structural diagrams present a static view (snapshot) of the system being modeled and behavioral diagrams describe the functionalities of the system. Specifically, the structural diagrams are class diagrams, object diagrams, component diagrams and deployment diagrams. The behavioral diagrams are activity diagrams, sequence diagrams and state machine diagrams. All of the above diagrams conform (obey) to a certain metamodel that is expressed in the Meta-Object Facility (MOF) [Object Management Group, 2006].

In this dissertation, class diagrams are employed to define the structure of either input simulation data or test cases, whereas activity diagrams are the basis for capturing precise interpretations of the (procedural) policies envisaged by the law. Specifically, Chapters 3 through 5 refer to class and activity diagrams, whereas the test case generation approach discussed in Chapter 6 takes as input a given class diagram (that needs to be instantiated for generating test case data). In general:

- Class diagrams describe a (static) structural view of a software system. In particular, class diagrams captures important domain concepts and their relationships using, among others, classes, attributes, operations, and associations. Class diagrams are also commonly used for representing domain models [Larman, 2004].
- Activity diagrams are used for specifying the business processes that describe a given functionality of the system. In particular, activity diagrams can model the flow of data, control and physical items.

In this dissertation, we do not use the default version of UML models but we rather rely on a customized version of these models. Specifically, we customize UML models using UML profiles (introduced next).

2.3 UML Profiles

The UML standard provides a mechanism, namely UML profiles [Object Management Group, 2011b], for extending the definition of its concepts so that they can be customized for use in diverse domains, e.g., laws. A UML profile defines stereotypes (annotations) which can extend elements from the metamodels of the UML models. Stereotypes are used to introduce new domain-specific terminology and tag values, i.e., attribute values, to model conceptual elements.

A simple example (excerpt) UML profile, borrowed from the well-known MARTE profile [Object Management Group, 2011a] for modeling real-time and embedded systems, is shown in Fig. 2.1. The shaded elements in the figure represent UML metaclasses and the non-shaded elements represent the profile's stereotypes (and enumerations). This profile allows standard UML models such as class diagrams to duly model a clock for a given system.

As indicated by the filled-in arrow in Fig. 2.1, the «clock» stereotype extends the `Property` (metaclass for attributes), and `InstanceSpecification` (metaclass for class instances in object diagrams) metaclasses. This means that the «clock» stereotype can be latter attached to any attribute or class instance that should represent a clock. A clock has a type as indicated by the association between «clock» and «clockType» stereotypes. The «clockType» stereotype extends the UML metaclass `Class` (from class diagrams). This means that the «clockType» stereotype can be attached to any class construct from a class diagram that applies the profile in Fig. 2.1. The «clockType» stereotype has an attribute named *nature* that is used to state whether the underlying clock is discrete or intense (as shown by the enumeration named *TimeNatureKind* in Fig. 2.1). Further, the «clockType» stereotype has a second attribute named *unitType* to define the timing unit to use, e.g., seconds.

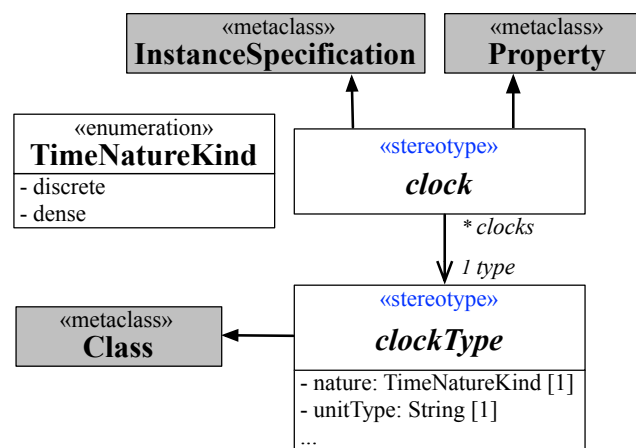


Figure 2.1. A Simple (Excerpt) UML Profile Borrowed from MARTE [Object Management Group, 2011a]

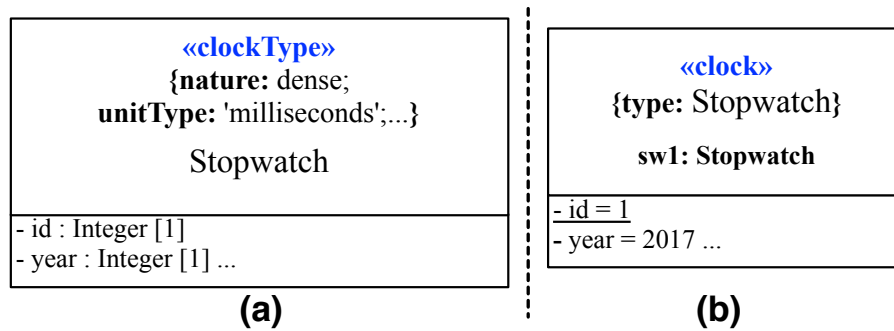


Figure 2.2. Applying the Stereotypes in Fig. 2.1 on: (a) a Class and (b) an Instance Specification

Fig. 2.2 illustrates the application of the stereotypes in Fig. 2.1 to extend (a) a class and (b) an instance specification. As shown in Fig. 2.1 (a), the class *Stopwatch* defines a specific type of stopwatches. The relevant information for the modeled stopwatch are shown in the attribute values (*nature* and *unitType* tag values) of the stereotype «clockType». Similarly and as shown by the «clock» stereotype of Fig. 2.1 (b), the instance specification named *sw1* represents a concrete stopwatch.

2.4 Object Constraint Language (OCL)

The Object Constraint Language (OCL) OCL is a general-purpose textual language that was standardized by the OMG group [Object Management Group, 2004]. OCL is used to define numerous type of expressions, e.g., invariants and post-conditions, that complement UML models including profiles. OCL is: (1) typed, meaning that each OCL expression must evaluate to either a primitive OCL type or to a type defined in the underlying model(s), (2) declarative, meaning that OCL expressions are free from imperative instructions such as assignments, and (3) *side-effect free*, meaning that the evaluation of any OCL expression cannot affect (change) the state of the system.

As stated earlier, OCL has several usages as listed below (not exhaustive):

- Invariants (define conditions that must always hold).
- Initialization of class properties.
- Definition of how certain UML elements (e.g., attribute values) should be derived.
- Query operations (define the body of a query operation such as getters).
- Operation contracts (i.e., set of pre- and post-conditions for UML operations).

In this dissertation, we exclusively use OCL as invariants or as query operations. In particular, in Chapters 3 through 5 we use OCL to embed queries into certain UML models. When evaluated, these queries retrieve the appropriate information (data) to process during simulation. Further, we also use OCL to define sets of consistency constraints (invariants) that aim at ensuring that UML models are well-formed in accordance to their underlying profiles. In Chapter 6, we define, via OCL invariants, the logical constraints that test cases (data to generate) must satisfy.

Fig. 2.3 presents an example of (a) an OCL invariant and (b) an OCL query operation. Both OCL expressions are defined over the class diagram in Fig. 2.2(a). The invariant in Fig. 2.3(a) enforces

- (a) `context Stopwatch inv example:
self.year > 0`
- (b) `Stopwatch.allInstances()->select(id = 1).year`

Figure 2.3. Example of OCL Expressions Defining (a) an Invariant and (b) a Query Operation

that all values for the *year* attribute of the class *Stopwatch* in Fig. 2.2 are positive. According to this invariant, the instance specification in Fig. 2.1 (b) is valid since it satisfies the invariant. The query operation in Fig. 2.3(b) returns the value of the *year* attribute (see Fig. 2.2) for the specific object with an *id* equal to one. When evaluated over the instance specification in Fig. 2.1 (b), this query operation returns 2017.

Chapter 3

Modeling Legal Policies

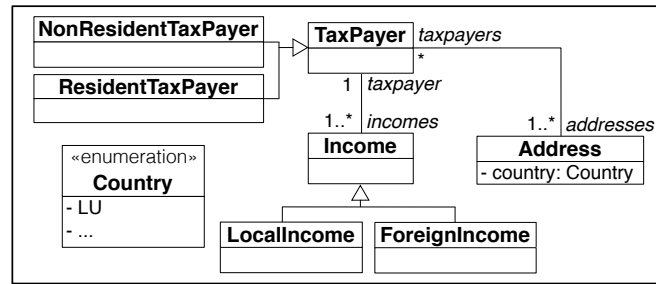
Legal compliance is a major concern for governments. In domains such as taxation and social benefits, laws need to be operationalized so that they can be implemented into administrative procedures and software systems. Such operationalization is typically performed by putting in place a legal framework, comprised of legislation, regulations, and circulars, aimed at providing a detailed interpretation of the underlying laws. These frameworks are often *prescriptive*: they provide step-by-step guidance in the form of *procedural rules* as to what needs to be done for compliance. Procedural legal rules are closely linked to the behavior of eGovernment applications. To illustrate, consider Article 2 from Luxembourg’s Income Tax Law [Government of Luxembourg, 2013], describing how taxpayers are classified as resident and non-resident:

Article 2.¹ Individuals are considered resident taxpayers if they have their address in the Grand Duchy. Individuals are considered non-resident taxpayers if they do not reside in the Grand Duchy but have a local income within the definition of Article 156.

To be able to analyze whether a software system complies with the taxpayer classification described in the law, one could develop a UML model like the one in Fig. 3.1: The domain model in Fig. 3.1(a) captures the main concepts and associations in Article 2 of the Income Tax Law; and the OCL expression in Fig. 3.1(b), written in the context of `TaxPayer`, provides a procedural rule (policy) for distinguishing between resident and non-resident taxpayers (L. 2-5 and 7-13, respectively).

To be a resident taxpayer, one must have a Luxembourgish address (L. 2-3). If such an address exists (L. 4), the taxpayer is deemed resident (L. 5). To be a non-resident taxpayer, one must have a local income (L. 7-8) but no local address. If these requirements are met (L. 9), the taxpayer is deemed non-resident (L. 10). A model like that in Fig. 3.1 makes the underlying legal article amenable to automated analysis. In particular, one can use such a model to check whether the outcome produced by a software system is consistent with the law. For example, using existing OCL evaluators such as EclipseOCL [Eclipse Foundation, NAa], one can verify if a system correctly classifies taxpayers (instances of the model in Fig. 3.1(a)) into resident and non-resident.

¹The article has been translated from the original French text and simplified.



(a)

```

1 context TaxPayer ResidentialStatus:
2 let hasLocalAddress: Boolean = self.addresses->
3   select (a: Address | a.country = Country::LU) -> notEmpty ()
4 if hasLocalAddress then
5   self.oclIsTypeOf (ResidentTaxPayer)
6 else
7   let hasLocalIncomes: Boolean = self.incomes->
8     select (i: Income | i.oclIsTypeOf (LocalIncome)) -> notEmpty () in
9   if hasLocalIncome then
10    self.oclIsTypeOf (NonResidentTaxPayer)
11   else
12     false
13   endif
14 endif
  
```

(b)

Figure 3.1. (a) Domain Model for a Legal Article, (b) Procedural Policy for the Article (Expressed in OCL)

Before a model such as the one in Fig. 3.1 can be used for automated analysis, it needs to be reviewed and validated by legal experts. To aid with validation, it is helpful to express procedural policies such as that in Fig. 3.1(b) in a visual manner.

In this chapter, we develop a visual and at the same time semantically-precise way to model procedural legal policies. Our approach follows the Domain-Specific Modeling (DSM) paradigm; but rather than building a new language, we use UML’s built-in customization mechanism, namely *profiles* [Object Management Group, 2011b], to adapt UML for use in our context. Using UML is motivated by its widespread use, commercial tool support, and the availability of standard extension mechanisms in the language.

Our work addresses a real need observed during our collaboration with our public service partner, CTIE (Centre des Technologies de l’Information de l’Etat). CTIE is Luxembourg’s national center for information technologies and responsible for developing eGovernment services for the state. CTIE already applies Model Driven Engineering (MDE), including UML and its extensions, for system development and is interested in enhancing its development methods with means for modeling legal policies. An important consideration for CTIE is for the models to be palatable to governmental stakeholders without IT background, but who have familiarity with simple conceptual models and business process models from earlier exposure and training.

The approach we propose in this chapter is not meant as a general solution for modeling all types of legal policies. In particular, we focus on prescriptive legal frameworks where legal policies are procedural. This situation is typical of highly-regulated domains such as taxation and social benefits. In general, however, many legal frameworks, e.g. privacy laws, are declarative, with rules defined

using deontic notions, i.e., permissions, obligations, and prohibitions [Ruiter, 1993]. Our current solution does not extend to declarative legal rules. In the rest of this chapter, we therefore take legal policy to mean “procedural” legal policies.

The starting point for our work is a field study, where we interacted with legal experts and analyzed several legal statutes, to identify both the information needs and the sources of complexity in the formalization of (procedural) legal policies (Section 3.1). Drawing on our field study, we define a UML-based methodology for modeling legal rules (Section 3.2). The core component of the methodology is a customization of UML Activity Diagrams, defined through a UML profile (Section 3.3). To use for analysis purposes the models resulting from our approach, we provide an algorithm for automatic transformation of the models into OCL (Section 3.4). We report on a case study, providing initial evidence for the feasibility and usefulness of our approach (Section 3.5). Finally, we compare our approach with related work and suggest avenues for future work (Sections 3.6–3.7).

3.1 Field Study of Legal policies

Our field study applies a Grounded Theory (GT) process [Corbin and Strauss, 2008], whereby observations and analysis of collected data are used for defining the problems to be addressed. In our context, we apply GT to define (1) what needs to be expressed in models of legal policies, i.e., the information requirements that such models should meet; and (2) factors that lead to complexity in models of legal policies and thus need special consideration in an approach targeted at building such models.

We began our field study with a series of meetings with legal experts, totaling ≈ 15 hours. The purpose of these meetings was (a) for the researchers to develop familiarity with legal concepts; (b) to define a suitable scope for the laws to consider; and (c) to identify representative legal policies for further investigation. Taxation was selected as the scope for the study, partly because of the priorities of the legal experts in our study, and partly because of the tax law’s large societal impact. Our field study resulted in several observations, outlined below.

Information Requirements. To identify the information needs in the specification of legal policies, we analyzed selected legal texts concerned with personal income taxes. While personal income taxes are only one facet of the tax law, the experts deemed the scope to be largely representative within the tax domain and the closely related domain of social benefits delivery. With help from legal experts, we identified, read, and interpreted the legal provisions relevant to personal income taxes. Our analysis covered a summary of direct taxes levied by the Government of Luxembourg, 16 articles from Luxembourg’s Income Tax Law (for brevity, referred to as *LITL* in the remainder of this chapter), three regulations, one tax scale, and several official web pages and circular letters.

While reading the above material, we applied a standard technique from qualitative data analysis [Corbin and Strauss, 2008] for analyzing text, and classifying, describing, and connecting the information presented in it. We annotated each important concept with a label denoting the nature of

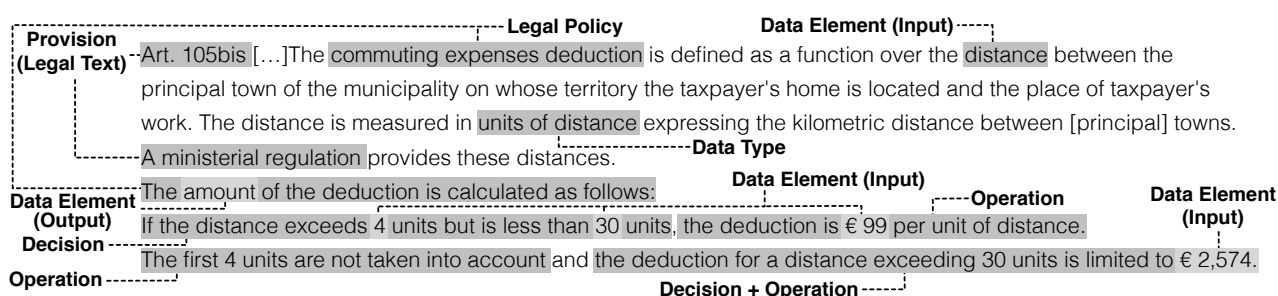


Figure 3.2. Excerpt of Article 105bis from *LITL* (Translated from French)

the concept, i.e., a meta-concept. Each time that a new meta-concept was encountered, we defined it in a glossary. As we proceeded through the text, we either created new labels or reused previous ones based on the definitions we had. We illustrate our analysis over an excerpt, shown in Fig. 3.2, of Art. 105bis of *LITL*. The excerpt, a simplified translation of the original French text, covers many of the information requirements identified by our study. The meta-concepts gleaned from the excerpt are shaded and labeled.

The (Legal) Policy the excerpt is concerned with is calculating the deduction a taxpayer is eligible for in relation to their commuting expenses. A policy may depend on several Provisions. It is important to maintain traceability from policies to the provisions they depend on. This is necessary both for reasoning about compliance and also for managing change in a predictable way. For the commuting expenses deduction, these provisions are: Art. 105bis of *LITL*, and an abstract reference to a ministerial regulation. *LITL* does not cite any regulation explicitly, as regulations may vary from year to year. The regulation that was in effect for commuting distances at the time our study was conducted is the ministerial regulation of February 6, 2012 (“règlement ministériel du 6 Février 2012”).

Each policy is made up of a set of Decisions and Operations, describing the (procedural) flow of the policy. An example decision from the excerpt is: “If the distance exceeds 4 units but is less than 30 units”; an example operation is setting “the deduction [amount to be] €99 per unit of distance”.

There are several Data Elements in the excerpt, denoting inputs to, outputs from, or intermediate values computed within the policy. For example, *distance* is an input to and *amount* is the output from the policy. The constants in the text, e.g., €2,574, are marked as input. This choice is motivated by the fact that constants may change over time and thus need to be treated explicitly.

Data elements are typed. The types are sometimes specified in the text, e.g., the excerpt states that *distance* is measured in certain units; but most often, the types are implicit, e.g., for monetary values and dates. One of the goals of our analysis was to identify and restrict the data types associated with the inputs and outputs of legal policies. Doing so is important for improving consistency. For example, all mathematical operations, e.g., summation and multiplication, over monetary values have to be consistent in how they round decimal values with precision points. A uniform treatment requires a specific data type to be defined for monetary values and used consistently in all legal policies.

For data elements that represent inputs, it is important to maintain traceability to the sources where the inputs come from. Some inputs are obtained directly from legal texts, e.g., the constants in the excerpt of Fig. 3.2. Alternatively, an input may be provided based on expert judgment by a legal agent. For example, to decide whether a company is eligible for certain deductions, a tax officer may need to determine whether the accounting performed by the company is adequate. Finally, an input may be derived from a physical or electronic data record, e.g., the *distance* input mentioned in the excerpt.

We distinguish different sources for inputs. The distinctions are important for better elaboration and validation of legal policies. For example, elaborating the inputs derived from electronic records often requires consultation with both legal experts and IT staff; whereas, inputs based on expert judgment or legal texts typically only concern legal experts.

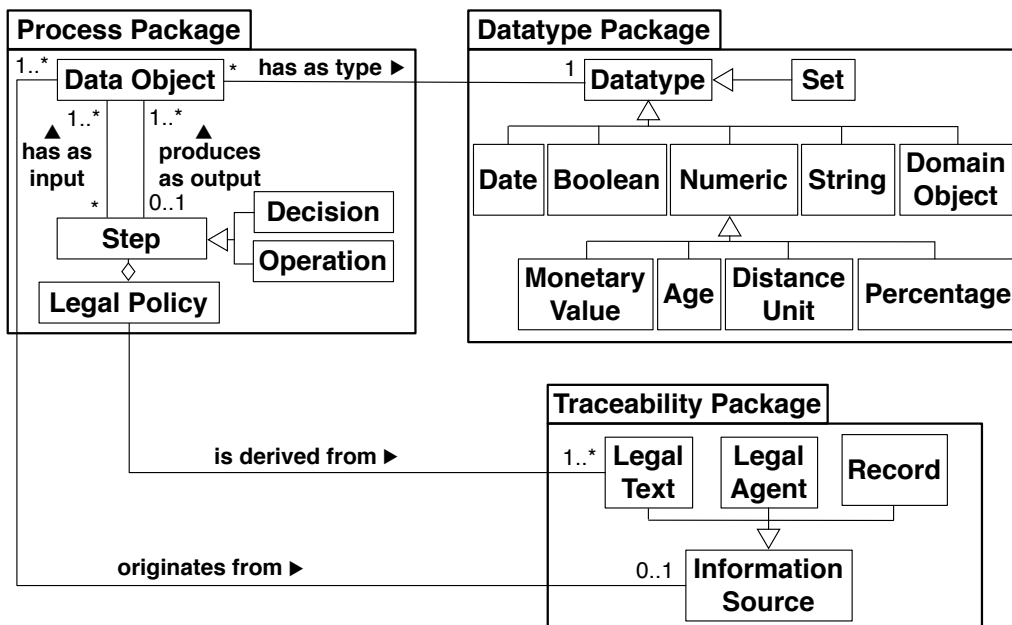


Figure 3.3. Information Model for Legal Policies

Based on our analysis above, we have developed an abstract information model, shown in Fig. 3.3, for legal policies. The model is organized into three packages, in line with the three main observations from the analysis, namely: (1) capturing legal policies as decisions and operations, (2) maintaining traceability, (3) restricting data types to what is essential.

The Process package in Fig. 3.3 defines the concepts related to the flow of a policy. Each Legal Policy is made up of a set of Steps, which can be either Decisions or Operations. Each step has Data Objects as input and output. The Traceability package groups the information sources to which traceability needs to be maintained from the elements in the Process package. Policies need to be traceable to Legal Texts. Inputs need to be linked to the Legal Text, Legal Agent, or Record where they originate from. The Datatype package contains a partial list of data types identified in our study. There is a special data type, named Domain

Object, to enable handling instances of domain concepts, e.g., `TaxPayer` (see Fig. 3.1(a)). In addition, the data types include a composite type, `Set`, to enable handling sets of objects. Note that the data types in Fig. 3.3 are specific to the tax law and may require tailoring if the approach is applied to other laws and regulations.

Complexity factors. We considered nine legal policies from *LITL* in our analysis of complexity factors. Six of these concern requirements on taxpayers' records (e.g, the taxpayer classification in Fig. 3.1). The rest concern the calculation of income tax credits. We captured these policies in OCL (total of 108 OCL lines, excluding comments and blanks). Our investigation of the resulting OCL constraints alongside our interactions with legal experts led to the following observations:

– *Navigation*: Navigation expressions in OCL tend to be lengthy for legal policies. For example, Art. 127 of *LITL* sets a cap on the costs a taxpayer can claim for the care of dependents. Calculating this cap requires identifying the dependents who live in the same household as the taxpayer but are not taxpayers themselves, and for whom the taxpayer receives some allowance. The corresponding OCL navigation expression (the OCL context being `TaxPayer`) is as follows:

```
self.taxPayerDependents→select(dependent:Person| not dependent.oclIsTypeOf(TaxPayer) and  
dependent.addresses→intersection(self.addresses)→notEmpty() and  
dependent.allowances.amount→sum()>0).
```

The complexity of navigation expressions is caused in part by the expressive (and thus long) labels of domain model elements in legal contexts, and in part by the richness of legal policies and the need for multiple navigation levels.

– *Branching*: Legal policies often have numerous decision branches, capturing the different cases where they apply and the corresponding actions to take. To illustrate, we recall the example of Fig. 3.1. Even for the simple task of classifying taxpayers into resident and non-resident, we need two if-then-else statements (or similarly complex propositional logic equivalents of if-then-else). This number rises to six or seven for more complex policies. Feedback from legal experts indicate that branching statements negatively impact comprehension.

– *Iteration*: OCL iterator operations (e.g., `select`, `exists`, `forAll`, `iterate`) are often inevitable in legal policies. For instance, in the navigation expression given earlier for identifying eligible dependents, one has to iterate over the dependents to determine which ones satisfy the desired criteria. Our interaction with legal experts suggests that iterations, specially nested ones, reduce comprehensibility.

Our approach, described next, take steps to address the observed information requirements and complexity factors.

3.2 Modeling Methodology

An overview of our modeling methodology is shown in Fig. 3.4. The legal texts and the specific provisions within them that are relevant to the legal policies of interest are provided as input by legal experts. The modeling step in the methodology includes two parallel but interrelated tasks: (1) modeling the domain and (2) modeling the legal policies. Both tasks require close interaction with legal experts to ensure a sound understanding of the underlying legal notions. The first task results in a domain model, providing a precise representation of the concepts and relationships in the input legal texts. As is common in object-oriented analysis, we use UML class diagrams for representing domain models [Larman, 2004]. A domain model excerpt for Article 2 of *LITL* was shown in Fig. 3.1(a). We follow standard practices for domain modeling [Larman, 2004] and thus do not elaborate this task further.

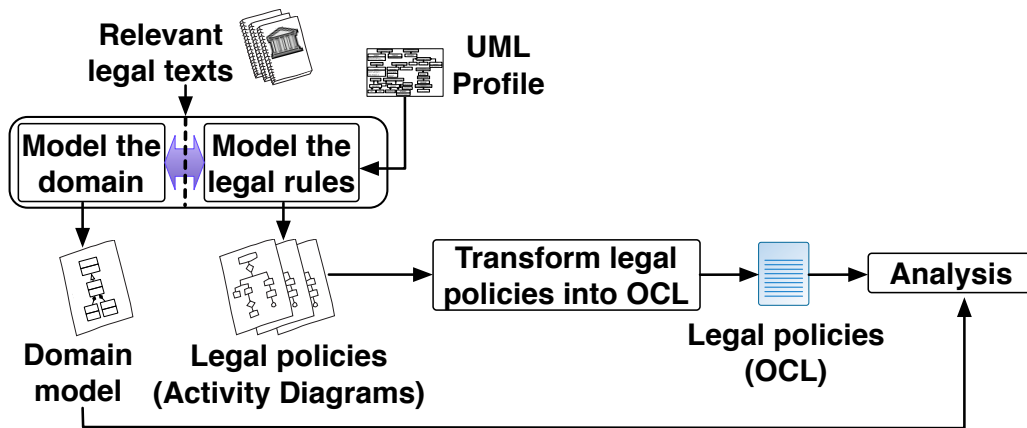


Figure 3.4. Methodology for Specifying Legal Policies

The second modeling task, i.e., modeling of the legal policies, is performed using a customization of UML Activity Diagrams (ADs). ADs have long been used for modeling procedural aspects of systems and organizations [Korherr and List, 2006]. The procedural nature of legal policies makes ADs a good match for our needs. Our customization of ADs is based on UML profiles [Object Management Group, 2011b], i.e., a standard mechanism to extend UML diagrams with domain-specific modeling concepts.

The domain model in our methodology is an instrument for elaborating the information that legal policies use as input. It is thus best to conduct tasks (1) and (2) *in tandem* and not sequentially. Doing these tasks in parallel ensures that the domain model is aligned with the legal policies in terms of data needs, and further narrows the scope of domain modeling to what is necessary for supporting the legal policies of interest. Once the legal policies have been modeled using our tailored AD notation, the models are automatically translated into OCL. The resulting OCL expressions along with the domain model can then be used for automated analysis using OCL evaluators [Eclipse Foundation, NAa] and OCL solvers [Cabot et al., 2008, Ali et al., 2013].

Our main technical goal in this chapter is to present the profile we have developed to customize ADs for expressing legal policies, and to describe how ADs built using our profile are transformed into OCL. The profile and the OCL transformation are respectively tackled in Sections 3.3 and 3.4.

3.3 UML Profile for Legal policies

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 [Object Management Group, 2011b]. The profile is shown in Fig. 3.5. The shaded elements in the figure represent UML metaclasses and the non-shaded elements represent the profile’s stereotypes (and enumerations). We provide definitions for the stereotypes in Table 3.1.

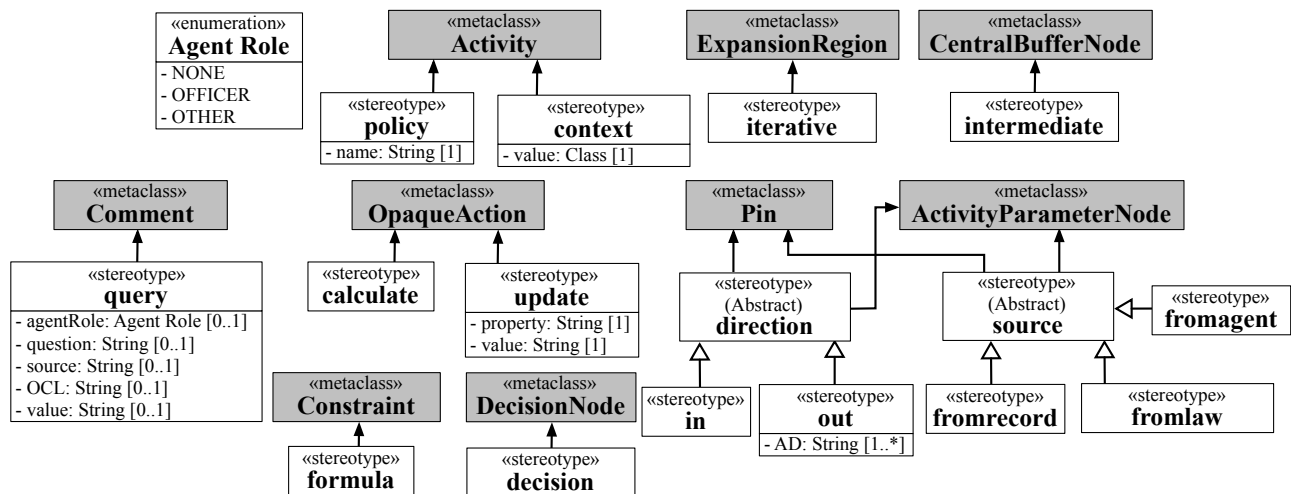


Figure 3.5. Profile Customizing UML Activity Diagrams for Legal Policy Models

Our profile’s stereotypes are shown in the first column of Table 3.1, followed by a description in the second column. The third column shows the UML metaclass(es) that each stereotype extends. We distinguish two kinds of stereotypes: (1) those that directly represent concepts from the information model of Fig.3.3, and (2) those that are auxiliary, providing additional information about model elements. The fourth column in Table 3.1 shows the mapping between the stereotypes and the concepts and packages of our information model. Auxiliary stereotypes are marked as *Auxiliary* in the column. The *Datatype* package of the information model is not represented through stereotypes. Instead, typing information is attached directly to the input and output nodes of ADs. The relevant fragment of the UML metamodel and our datatype library are provided in Appendix A.1 and Appendix A.2, respectively.

We illustrate our profile over the Commuting Expenses Deduction policy from the excerpt of Article 105bis given in Fig. 3.2. The French term for this deduction is “Frais de Déplacement”. We refer to this deduction as FD. In Fig. 3.6, we show how the FD policy is modeled using an AD. The «policy» stereotype applied to the AD in this figure indicates that the AD is a legal policy. The AD is further annotated with a «context» stereotype denoting the OCL context in which the AD is being

Table 3.1. Our Profile’s (Non-Abstract) Stereotypes

Stereotype	Description	Concept & Package	
«policy»	Defines an activity as a legal policy	Legal Policy	Process
«iterative»	Defines an iterative region		
«context»	Defines the OCL context in which a legal policy is being specified	Auxiliary	
«decision»	Defines a decision step	Decision	
«calculate»	Defines an operation that calculates a value	Operation	
«assert»	Defines an operation that checks an assertion		
«update»	Defines an operation that updates an object or the value of a parameter. This stereotype enables side-effect operations during simulation, e.g., storing simulation results. This stereotype is later discussed and exemplified in Section 4.2.	Auxiliary	
«in»	Defines an input to a legal policy	Data Object	
«out»	Defines an output from a region		
«intermediate»	Defines an intermediate value resulting from a calculation	Auxiliary	
«formula»	Defines the formula for a calculation	Auxiliary	
«statement»	Defines the logical expression for an assertion	Auxiliary	
«fromlaw»	Declares a (constant) input as originating from a legal text	Legal Text	Traceability
«fromagent»	Declares an input as being provided by a legal expert	Legal Agent	
«fromrecord»	Declares an input as being retrieved from a record (e.g., a database)	Record	
«query»	Defines the query for obtaining an input from its respective source	Auxiliary	

specified. The context is always an instance of a class from the underlying domain model. For the AD in Fig. 3.6, the context is an instance of the `TaxPayer` class from the domain model.

The core of the AD in Fig. 3.6 is a calculation procedure. The procedure yields a value of 0 (zero) when a taxpayer is deemed not eligible for FD, i.e., when `distance > minimal_distance` is false. For an eligible taxpayer, the procedure yields the result of multiplying three quantities: (1) a flat rate (constant) from the law, denoted `flat_rate`, (2) the distance between a taxpayer’s work and home addresses, denoted `distance`, and (3) a prorated ratio representing the full-time equivalent period during which the taxpayer has been employed over the course of the tax year, denoted `prorata_period`. The formula applies up to a maximum home-to-work distance threshold, denoted `maximal_distance` and specified in the law. Beyond this threshold, a nominal rate, denoted `maximal_flat_rate`, is applied irrespective of distance but prorated as discussed above.

As illustrated in Fig. 3.6, the decisions and calculations are marked respectively with the «decision» and «calculate» stereotypes. Each calculation has a «formula» constraint attached, providing

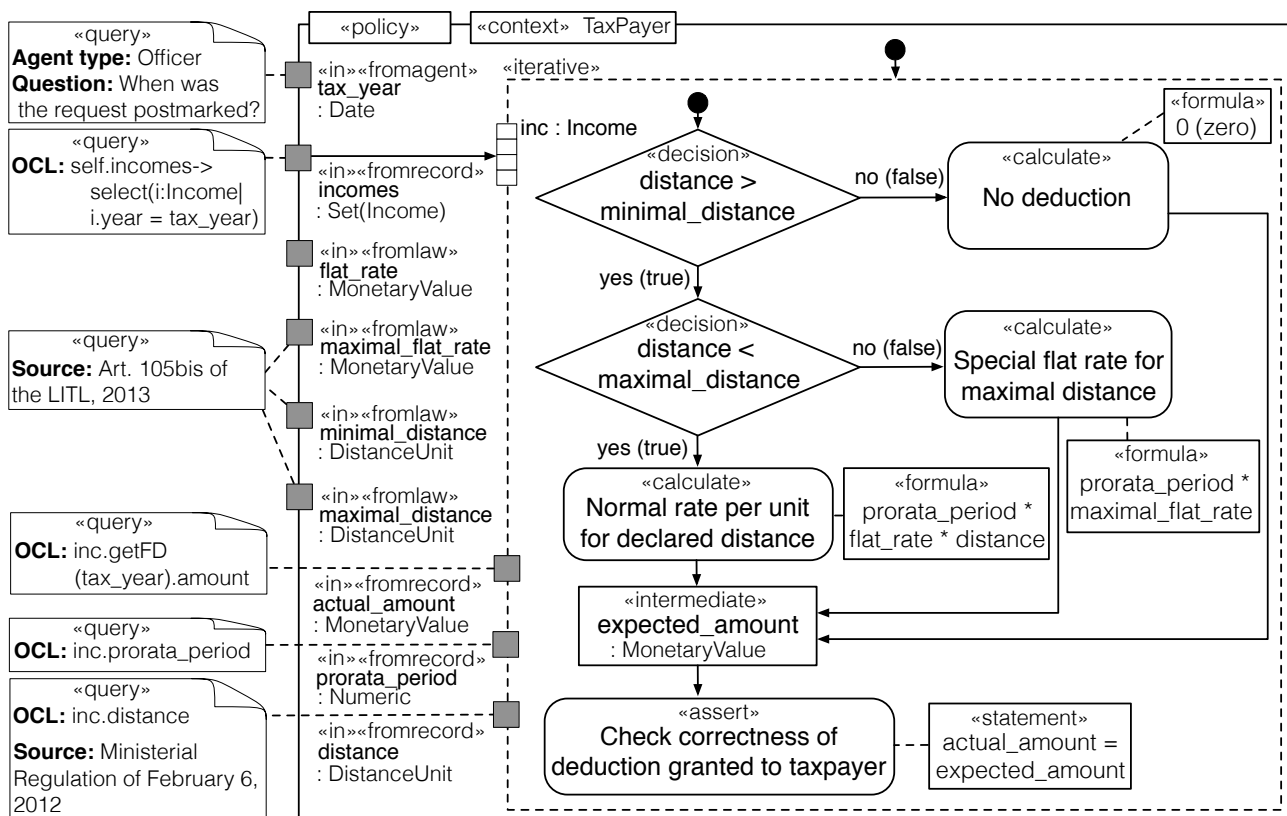


Figure 3.6. Activity Diagram for Commuting Expenses Deduction (FD)

the formula for the calculation. The result of a calculation is always stored in a (typed) intermediate variable marked by the «intermediate» stereotype, e.g., `expected_amount` in Fig. 3.6.

Each legal policy concludes with an assertion: an operation marked by the «assert» stereotype and providing an implicit Boolean output for the policy. Specifically, an assertion is used to ascertain that the outcome produced by a system or a human agent matches the outcome envisaged by the legal policy. Associated with an assertion is a constraint with the «statement» stereotype, defining the Boolean claim that needs to be checked. For example, the assertion in Fig. 3.6 checks whether the value of FD on a taxpayer’s file, denoted `actual_amount`, matches the value computed by the policy, denoted `expected_amount`.

Inputs to decisions and operations are represented by small rectangles with a gray shade and an «in» stereotype. The origin of each input is captured through one of the following stereotypes: «fromlaw», «fromrecord» or «fromagent». Each input has a query attached to it, represented as a comment with a «query» stereotype. A query provides details on how an input is obtained from its source. The «fromlaw» stereotype is used for inputs that are constants and specified in a legal text, e.g., `flat_rate`. For these inputs, the query provides a traceability link to the legal provision where the constant is defined. The «fromrecord» stereotype is used for inputs derived from a record, e.g., `incomes`. For these, the query is an OCL expression over the underlying domain model. Additional information may be provided along the OCL expression such as the legal text that describes the input, e.g., `distance`. Finally, «fromagent» stereotype is applied to inputs that originate from a

legal agent, e.g., `tax_year`. For these, the query provides information about the agent type (role) authorized to provide the input as well as the question that the agent needs to answer.

The legal policy in Fig. 3.6 takes into account the fact that a taxpayer may have multiple (simultaneous or sequential) employment activities, and thus multiple incomes and work addresses. To correctly compute the FD for a taxpayer, one needs to iterate over *all* incomes gained by the taxpayer and ensure that the computation of the FD portion for each income is consistent with the law. To capture this iterative behavior, we use expansion regions from the AD notation. An expansion region is an activity region that executes multiple times over the elements of an input collection [Object Management Group, 2015]. The legal policy of Fig. 3.6 has one expansion region with `incomes` as its input collection. UML provides three execution modes for expansion regions: *iterative*, *parallel*, and *stream* [Object Management Group, 2015]. Of these, our profile uses only the iterative mode, marked by the «iterative» stereotype. In this mode, executions are performed sequentially and according to the order of elements in the input collection. 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 can be used in the OCL expressions associated with the inputs of the expansion region, e.g., the OCL expression in the query attached to `prorata_period`.

The expansion region in the legal policy of Fig. 3.6 does not require an explicit output because the assert operation occurs within the expansion region. Our profile allows expansion regions to have an explicit output. This is useful for capturing complex iterative calculations, e.g., computing the total amount of benefits received by the dependents of a taxpayer. We use the «out» stereotype to denote the (explicit) output of an expansion region or a policy model, if one exists.

Consistency Constraints. To apply our profile in a sound manner, a number of consistency constraints need to hold. We provide a complete list of these constraints in Appendix A.3. The consistency constraints are aimed at enforcing the following: (1) Completeness of the information in models of legal rules, e.g., to ensure that the source for each input has been specified through the application of an appropriate stereotype and a query; (2) Mutually exclusive application of certain stereotypes, e.g., to ensure that each input has one and only one source stereotype («fromlaw», «fromrecord» or «fromagent») applied to it; and (3) Restrictions on the structure of ADs. Most notably, these structural restrictions ensure that the flows do not give rise to cyclic paths, and further that only the notational elements allowed by our methodology are being used. All consistency constraints can be enforced as the models are being built.

We next describe how ADs built using our profile are transformed into OCL.

3.4 Transforming Legal Policies into OCL

In this section, we provide an algorithm to automatically transform ADs to OCL, and illustrate this model-to-text transformation over the Commuting Expenses Deduction (FD) policy discussed in Section 3.3. The choice of OCL as the target language for the transformation is motivated by OCL being

Alg. 1: Activity Diagram to OCL (ADToOCL)

Inputs : (1) An Activity Diagram, \mathcal{AD} . (2) An element $\in \mathcal{AD}$.
Output: An OCL string, result.

```

1 if (element is NULL) then
2   | return ' ' /* Return empty string */
3 end if
4 Let  $P$  be the transformation pattern applicable to element.
5 Let  $input_1, \dots, input_n$  be the non-declared inputs required by  $P$ .
6 foreach  $input_i$  do
7   | result  $\leftarrow$  result + ADToOCL( $\mathcal{AD}$ ,  $input_i$ )
8 end foreach
9 if (element is not a DecisionNode) then
10  | Let  $st_1, st_2$  respectively be the opening and closing OCL fragments obtained from applying  $P$ .
11  | Let  $next$  be the next element to visit. /* ... chosen based on  $P$  and its outgoing flow */
12  | result  $\leftarrow$  result +  $st_1$  + ADToOCL( $\mathcal{AD}$ ,  $next$ ) +  $st_2$ 
13  | if (element is an ExpansionRegion with an output) then
14  |   | Let  $out$  denote the output element.
15  |   | result  $\leftarrow$  result + ADToOCL( $\mathcal{AD}$ ,  $out$ )
16  | end if
17 else
18  | Let  $f_1, \dots, f_m$  be the outgoing flows from element. /*  $m \geq 1$  */
19  | foreach  $f_i$  do
20  |   | if ( $i = 1$ ) then
21  |     | result  $\leftarrow$  result + 'if (' + element.name + ') = ' +  $f_i$ .name + ' then ' + ADToOCL( $\mathcal{AD}$ ,  $f_i$ .target)
22  |     | else
23  |       | result  $\leftarrow$  result + 'else if (' + element.name + ') = ' +  $f_i$ .name + ' then ' + ADToOCL( $\mathcal{AD}$ ,  $f_i$ .target)
24  |       | end if
25  |     end foreach
26  |     result  $\leftarrow$  result + 'else false' +  $\underbrace{\text{'endif' + \dots + 'endif'}}_{m \text{ times}}$ 
27  | end if
28  return result

```

Input declarations { 1-5
 Transforming non-decisions { 9-16
 Transforming decisions { 17-28

```

1. context TaxPayer inv FD:
2. let tax_year:Date = self.tax_year in
3. let incomes:Set(Income) = self.incomes->select(i:Income | i.year = tax_year) in
4. incomes->forall(inc:Income |
5. let distance:DistanceUnit = inc.distance in
6. let minimal_distance:DistanceUnit =
7. Constant::MINIMAL_DISTANCE.oclAsType(DistanceUnit) in
8. if (distance > minimal_distance) = true then
9. let maximal_distance:DistanceUnit =
10. Constant::MAXIMAL_DISTANCE.oclAsType(DistanceUnit) in
11. if (distance < maximal_distance) = true then
12. let flat_rate:MonetaryValue =
13. Constant::FLAT_RATE.oclAsType(MonetaryValue) in
14. let prorata_period:Numeric = inc.prorata_period in
15. let expected_amount:MonetaryValue = prorata_period * flat_rate * distance in
16. let actual_amount:MonetaryValue = inc.getFD(tax_year).amount in
17. actual_amount = expected_amount
18. else if (distance < maximal_distance) = false then
19. let maximal_flat_rate:MonetaryValue =
20. Constant::MAXIMAL_FLAT_RATE.oclAsType(MonetaryValue) in
21. let prorata_period:Numeric = inc.prorata_period in
22. let expected_amount:MonetaryValue = prorata_period * maximal_flat_rate in
23. let actual_amount:MonetaryValue = inc.getFD(tax_year).amount in
24. actual_amount = expected_amount
25. else false endif
26. endif
27. else if (distance > minimal_distance) = false then
28. let expected_amount:MonetaryValue = 0 in
29. let actual_amount:MonetaryValue = inc.getFD(tax_year).amount in
30. actual_amount = expected_amount
31. else false endif endif
32. )

```

Context Pattern
 Initial Node Pattern
 Expansion Region Without Output Pattern
 Initial Node Pattern
 Decision Node Pattern
 Intermediate Value Pattern
 Intermediate Value Pattern
 Intermediate Value Pattern
 Assert Pattern

Figure 3.7. Generated OCL Expression for the Example of Fig. 3.6 (FD)

part of the UML and further to benefit from existing testing and simulation frameworks, e.g., [Ali et al., 2013], that are built around OCL. This section does not cover all the implementation details of our transformation. See Appendix B.1 for full details.

The algorithm for the transformation, named ADToOCL and shown in Alg. 1, takes as input an Activity Diagram, \mathcal{AD} , and an element $\in \mathcal{AD}$. Specifically, \mathcal{AD} is an instantiation of the UML meta-model fragment for activity modeling, and element is an object within this instantiation. We assume that \mathcal{AD} satisfies the consistency constraints of our profile (Section 3.3). To ensure consistency between the semantics of activity diagrams and that of the OCL constraints generated from them, we further assume that \mathcal{AD} uses only deterministic decisions.

Initially, the algorithm is called over \mathcal{AD} with element pointing to the root `Activity` instance in \mathcal{AD} . In Fig. 3.7, we show the OCL constraint resulting from the application of Alg. 1 to the FD policy of Fig. 3.6. Note that when our approach is applied, analysts work over the ADs and are not exposed to such complex OCL constraints. The generated constraints are meant for use by OCL engines.

The transformation is based on a set of predefined patterns. These patterns are detailed in Appendix B.2. Each pattern is defined as a graph P . If P is matched to a subgraph of \mathcal{AD} rooted at element, the appropriate OCL fragment for P is generated. For example, consider the intermediate value `expected_amount` in FD. There is a pattern, *Intermediate Value Pattern*, that deals with such values. This pattern has the following shape: an action with the «calculate» stereotype connected by a flow to an intermediate value with the «intermediate» stereotype. The application of this pattern generates a **let** expression which defines an intermediate value based on a given calculation. This pattern is applied three times during the transformation of FD for the three calculations that lead to `expected_amount`. The OCL fragments for these three applications are shown on L. 15, 22, and 28 of the constraint in Fig. 3.7.

The transformation process is recursive and mimics a depth-first traversal of the underlying graph of \mathcal{AD} . There are three main parts to this process: (1) input declarations (Alg. 1, L. 4-8); (2) transformation of all elements other than decisions (Alg. 1, L. 10-16). Within this class of elements, additional processing is necessary for expansion regions to propagate their output if they have one (Alg. 1, L. 13-16); and (3) transformation of decision nodes (Alg. 1, L. 18-26).

The first part of the transformation process concerns identifying all inputs to be declared before transforming a given element (Alg. 1, L. 5). Each such input is transformed into a **let** expression (Alg. 1, L. 6-8). To illustrate, consider the decision `distance > minimal_distance` in FD. The inputs to this decision are transformed into L. 5-7 of the constraint in Fig. 3.7. This is performed before the transformation of the decision itself (Fig. 3.7, L. 8). An input may have dependencies to other inputs, e.g., `incomes` (Fig. 3.7, L. 3) depends on `tax_year` (Fig. 3.7, L. 2). Such dependencies are handled through the recursive call of L. 7 in Alg. 1.

The second part of the process handles non-decisions. This is where the initial call to Alg. 1 begins to unwind. The initial call is handled by the *Context Pattern*, which transforms the context information attached to an `Activity` instance via the «context» stereotype. There are no inputs

associated with the *Context Pattern*. Handling the pattern thus reduces to executing L. 10-12 of Alg. 1. The opening OCL fragment (st_1) resulting from the application of this pattern is L. 1 of Fig. 3.7; the closing fragment (st_2) is empty. Then, on L. 12 of Alg. 1 a recursive call is made with *next* set to the initial node of FD. The unwinding of this recursive call generates the remainder of the OCL constraint (Fig. 3.7, L. 2-32). On the left side of Fig. 3.7, we mark the scope of each recursive call and the respective pattern. To avoid clutter, calls that handle input declarations are not marked.

The third and final part of the process transforms decisions into if-then-else statements. This part is analogous to what we previously described.

We have implemented our transformation using Acceleo [Eclipse Foundation, 2006] – a model-to-text transformation tool for Eclipse. Our Acceleo implementation is closely aligned with the way we present the transformation in Alg. 1. While we targeted in this section only OCL as the target language for the transformation, it is possible to modify our text generation rules (patterns) to support other languages, e.g., Alloy [Jackson, 2012].

3.5 Evaluation

We report on an industrial case study where we apply our approach to *LITL*. The case study is an initial step towards answering the following Research Questions (RQs): **RQ1**. *Is the approach expressive enough to model complex legal policies?* **RQ2**. *Is the level of effort required by our approach reasonable?* And, **RQ3**. *Are the ADs built using our approach structurally less complex than OCL constraints written directly?* In the longer term, we plan to perform more extensive user studies to evaluate the approach in a more thorough manner.

Our case study builds on an initiative by the Government of Luxembourg to improve its eGovernment services in the area of taxation. One of the main objectives of the initiative is to ensure that these services remain *verifiably compliant* with the tax law as the law evolves. A key prerequisite for verification of compliance is to have analyzable models of the tax law. Our case study develops such models for a substantial fragment of the income tax law. The case study was conducted in collaboration with our public service partner, CTIE.

Study selection and execution. Our study concerns a set of legal policy from *LITL*. Luxembourg has two complementary schemes for income taxes: (1) withholding taxes from salaries, and (2) assessing taxes based on a declaration. Our study focuses on the former scheme. The basis for withholding is a *tax card*, detailing the tax deductions and credits that apply to an policies income. Deductions are expense items subtracted from the gross income before taxes. Credits are items applied either against the taxes due or paid to the taxpayer in cash. A tax card provides information about five deductions and three credits. The deductions are for commuting expenses (FD), miscellaneous expenses (FO), spousal expenses (AC), extraordinary expenses (CE), and special expenses (DS). CE is decomposed into three sub-categories and DS into six. The credits are for salaried workers (CIS), pensioners (CIP), and single parents (CIM).

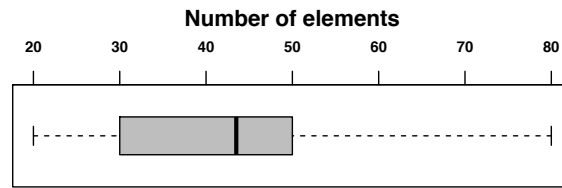


Figure 3.8. Number of AD Elements (Distribution)

The above deductions and credits give rise to 15 legal policies. We applied our methodology described in Section 3.2 for expressing these policies. This resulted in a domain model and 15 ADs built using our profile. The domain model has 7 packages, 61 classes, 15 enumerations, 106 attributes, and 24 operations. The distribution for the number of elements in the ADs is given in the box plot of Fig. 3.8. The element count for each AD is the sum of the number of inputs, outputs, decisions, actions, flows, intermediate variables, expansion regions, and constraint/comment boxes.

Discussion. We next discuss the RQs that motivated our study. The three tax credits in our study (CIS, CIP, and CIM) were used previously in our investigation of OCL complexity factors (Section 3.1) along with six other policies that are unrelated to the case study. Since we had a-priori knowledge about the tax credits, the AD models for the tax credits are uninteresting for RQ1. To mitigate learning effects, we further exclude these three models when discussing RQ2.

RQ1. Our profile provided enough expressiveness to conveniently capture the legal policies in our study. One of the factors we considered in our models was to avoid nested structures, particularly nested expansion regions. Although our profile and OCL transformation can handle nesting, models containing nested structures can be hard to comprehend. In our study, we could avoid nesting in all models by choosing a suitable OCL context for each of the legal policies.

RQ2. We are interested in measuring the level of effort as an indicator for whether the approach has a realistic chance of adoption in practice. The models were built following a half-day tutorial on personal income taxes by legal experts. The domain model was developed simultaneously with the ADs, as suggested in Section 3.2. Developing the 12 ADs for tax deductions took ≈ 40 person-hours (ph) including the effort spent on the domain model. This is an average of 3.3 ph per AD. The 3 ADs for tax credits took ≈ 7 ph to build, i.e., an average of 2.3 ph per AD. Only the tax deductions are representative in terms of effort, due to reasons discussed earlier. Once built, the ADs were presented to a group of six legal experts in a half-day training and walkthrough session. We received positive feedback from the legal experts involved in our study; however, we have not yet conducted a detailed user study to thoroughly assess our approach. We consider the overall effort to be worthwhile as the resulting models provide a complete characterization of the tax card, which applies to a large majority of the taxpayers.

RQ3. Our profile limits the use of OCL to the inputs, formulas, and statements of ADs. In this way, the profile to a large extent shields users from OCL and the structural complexity of OCL expressions. Reynoso et al. [Reynoso et al., 2004] argue that reductions in OCL structural complexity bring about reductions in cognitive complexity and improvements in understandability. The aim of RQ3 is to measure the value of our profile in terms of structural complexity reduction when compared to the

Table 3.2. Comparison of Complexity: Direct Use of OCL vs. OCL Fragments in ADs

		FD		FO		AC		CE1		CE2		CE3		DS1		DS2		DS3		DS4		DS5		DS6		CIS		CIP		CIM	
		M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A
Complexity metrics	C_1 (# navigations)	11	5	12	5	11	10	10	5	6	2	12	6	6	6	8	3	17	11	9	3	4	3	9	6	6	2	6	3	13	7
	C_2 (# if statements)	3	0	5	0	6	0	7	0	4	0	4	0	4	0	3	0	6	0	4	0	3	0	3	0	2	0	2	0	3	0
	C_3 (# collection_op's)	3	0	6	0	3	3	17	2	8	0	10	1	8	3	4	1	12	3	8	0	3	1	5	1	6	4	6	4	5	1
	C_4 (# iterative_op's)	3	1	5	1	8	6	9	2	5	1	7	1	6	4	4	2	7	6	5	1	3	2	4	2	2	1	2	1	5	4

situation where legal policies are directly written in OCL. This comparison provides preliminary insights as to whether our profile can result in more intuitive and understandable specifications of legal policies.

To answer RQ3, we manually wrote constraints for the tax deductions, in a similar manner to the tax credits (Section 3.1). We then compared these constraints to the OCL expressions used in the ADs, i.e., the OCL expressions to which the users of our profile are exposed. For the comparison, we selected a subset of the OCL structural complexity metrics proposed by Reynoso et al. [Reynoso et al., 2004]. Our selection was driven by what we deemed relevant to the complexity factors observed in our field study (Section 3.1). Specifically, we consider the following metrics: number of navigations (C_1), number of if-then-else statements (C_2), number of operations on collections, e.g., any, sum, excludes (C_3), and number of iterative operations, e.g., select, forAll (C_4). C_1 and C_2 respectively reflect the *navigation* and *branching* complexity factors; C_3 and C_4 both relate to the *iteration* complexity factor.

Table 3.2 shows the metrics, across all the deductions and credits, for manually-written OCL constraints (denoted, **M**) vs. the OCL fragments used in the ADs (denoted, **A**). As the table suggests, the ADs built using our approach lead to reductions in structural complexity. In particular, the AD’s reduce on average: C_1 by 45%, C_2 by 100%², C_3 by 72%, and C_4 53%. The structural complexity that carries over to the ADs is primarily caused by the OCL expressions that define the inputs to the ADs. To validate the ADs with non-software engineers, one can replace these expressions with intuitive descriptions without any impact on the ADs. Finally, we need to emphasize that the complexity reductions seen are only suggestive of benefits, but not definitive evidence for them. Further empirical validation remains essential to determine whether the complexity reductions indeed translate into improved understandability.

3.6 Related Work

In this section, we compare our approach with several areas of related work.

Modeling of legal policies. Modeling legal policies has been a subject of study in the Artificial Intelligence (AI) community for more than 50 years [Bench-Capon et al., 2012]. For example, Rissland and Skal [Rissland and Skalak, 1991] combine example-based and rule-based reasoning to mimic the reasoning of legal experts in tax court cases; Melz and Valente [Melz and Valente, 2004] build a

²The 100% reduction is due to the fact that the ADs in our study do not contain if-then-else statements, as all branching behaviors are captured using decision nodes.

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 [Breaux and Anton, 2008] develop a rule-based framework for modeling rights and obligations from the US Health Insurance Portability and Accountability Act (HIPAA). Breaux and Powers [Breaux and Powers, 2009] derive business process models from the clauses in HIPAA. Ghanavati et al. [Ghanavati et al., 2007] use a combination of goal models and use cases for specifying legal requirements in the healthcare domain. Islam et al. [Islam et al., 2011] use UMLsec [Jürjens, 2002] –a UML profile for security– for modeling security requirements imposed by regulations. Ingolfo et al. [Ingolfo et al., 2013] apply conceptual modeling for analyzing the compliance of healthcare systems with legal requirements. de la Vara and Panesar-Walawege [de la Vara and Panesar-Walawege, 2013], Emmerich et al. [Emmerich et al., 1999a], and Sannier and Baudry [Sannier and Baudry, 2014] propose metamodels for regulations and standards. van Engers et al. [van Engers et al., 2001] 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 automation.

Nevertheless, work from the software engineering community also share the same limitations observed for work from the artificial intelligence community. Unlike our *UML-based* modeling approach, these strands of work lack operationalization means to support the automated analysis perused in our context, i.e., simulation and testing. Further, the modeling methodology provides guidance on how to efficiently build policy formalization from scratch. Our modeling notation also manages traceability between the legal text and the detailed legislature that interpret the law. This enables us, for instance, to locate the policy models that are impacted by a change in the law.

Verification of legal compliance. Several methods were proposed to ensure *business processes* compliance with legal policies in domains such as healthcare [Ghanavati et al., 2007, Goedertier and Vanthienen, 2006] and finance [Hassan and Logrippo, 2008]. For instance, Ghanavati et al. [Ghanavati et al., 2007] tracks legal compliance by a framework based on combining goal, use case and privacy goal models. Hassan and Logrippo [Hassan and Logrippo, 2008] provide a semi-automatic method for checking compliance of enterprise requirements with respect to legal requirements using formal logic.

Few strands address compliance for *software systems*. Existing work on regulatory compliance for software is geared towards corporate governance, privacy, and security [Breaux and Anton, 2008, Islam et al., 2011, Ingolfo et al., 2013]. In contrast, our research targets specific aspects of compliance that are proper to procedural legal policies that are the scope of this project. A major concern in *prescriptive* laws is ensuring that the software systems are performing all calculations, e.g., tax deductions, only for eligible individuals and as specified by the underlying provisions.

Visualization of logical languages. Bottoni et al. [Bottoni et al., 2000] and Stein et al. [Stein et al., 2004] propose visualizations for OCL, and Amàlio et al. [Amàlio et al., 2010] – for the Z language [Smith, 2000]. These approaches are not tailored to legal policies and lack means for addressing the information requirements and complexity factors discussed in Section 3.1.

Model-to-OCL transformation. Cabot et al. [Cabot et al., 2010] construct OCL transformations of domain-specific language rules, and Milanović et al. [Milanovic et al., 2007] derive OCL constraints from integrity rule models. These approaches neither address legal policies nor tackle the transformation of activity diagrams, as done in our approach.

3.7 Conclusion

In this chapter, we presented a UML-based approach for modeling procedural legal policies. The key component of the approach is a profile for activity diagrams. To enable automated compliance analysis, we defined a transformation that produces OCL specifications from activity diagrams built using our profile. We presented a preliminary evaluation of our approach.

Our approach focuses on prescriptive legal frameworks. In the future, we would like to investigate how and to what extent our approach can accommodate declarative frameworks and notions such as permissions and obligations. Another topic for future work is to conduct more field studies and generalize our UML profile to a larger set of legal domains. Further, a more thorough evaluation of our approach is essential. In particular, the legal experts in our study underwent training before they were able to understand our models. Legal experts trained in other approaches, e.g., mathematical logic, may have done equally well. User studies are necessary to determine what advantages and disadvantages our approach offers compared to the direct use of logic. Finally, we plan to study how our models can support automated analysis tasks such as simulation (see the next chapter).

Chapter 4

Simulating Legal Policies

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 [Figari et al., 2015]. 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 [Figari et al., 2015], SYSIFF [Canova et al., 2009] and POLIMOD [Sutherland, 1995], 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 [Panko, 1998, Hermans et al., 2012, Hermans et al., 2015]. 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 chapter 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.

Our simulation framework leverages the UML-based modeling methodology for specifying procedural policies presented in Chapter 3. 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 presented in this chapter are as follows, with 2) and 3) being the main ones:

- 1) We augment our model-to-text transformation to support policy simulation.
- 2) We develop a UML profile [Object Management Group, 2011b] to capture the probabilistic characteristics of a population. This profile shares some common goals with MARTE [Object Management Group, 2011a] 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.

Structure. The remainder of the chapter is organized as follows. Section 4.1 provides an overview of our framework. Section 4.2 describes our policy modeling technique and introduces a running example for the chapter. Section 4.3 discusses how policy models are transformed into simulation code. Section 4.4 elaborates our profile for specifying the probabilistic characteristics of the simulation population. Section 4.5 presents our simulation data generation algorithm. Section 4.6 outlines tool support. Section 4.7 reports on the empirical evaluation of our simulation framework. Section 4.8 points out the limitations of our simulation framework and the threats to the validity of our empirical evaluation. Section 4.9 compares our work with related work. Finally, Section 4.10 concludes the chapter with a summary and directions for future work.

4.1 Simulation Framework Overview

Fig. 4.1 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 (discussed in Chapter 4). 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.

4.2 Background and Running Example

The first step of our simulation framework, as depicted earlier in Fig. 4.1, is modeling the legal policies. This step relies on the UML-based methodology and notation for specifying (procedural) legal policies discussed in Chapter 4. Our earlier work was motivated not by simulation but rather by

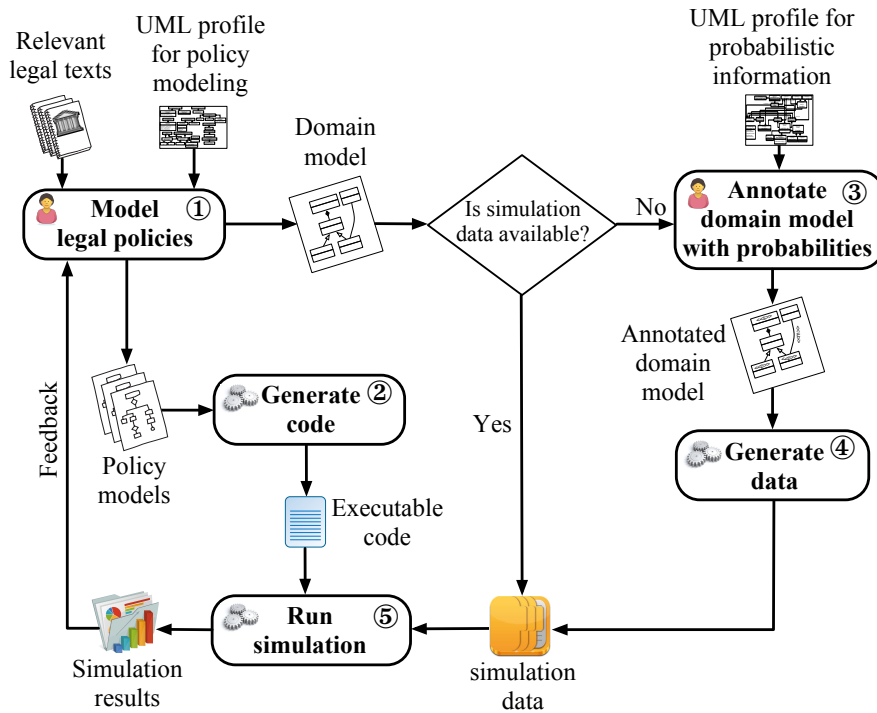


Figure 4.1. Simulation Framework Overview

model-based testing [Utting and Legard, 2007]. 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. This requirement is met via the «update» stereotype in Table 3.1 which allows policy models to perform operations with side effects such as updating any object in the simulation data.

We recall that legal policy modeling yields two types of models as illustrated in Fig. 4.1: (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 4.4, where we cover domain models alongside the probabilistic information that we attach to them for characterizing the simulation population. In the remainder of the section, we concentrate on the policy models (expressed as ADs).

We use as running example for this chapter the “invalidity” deduction legal policy, 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. 4.3. 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

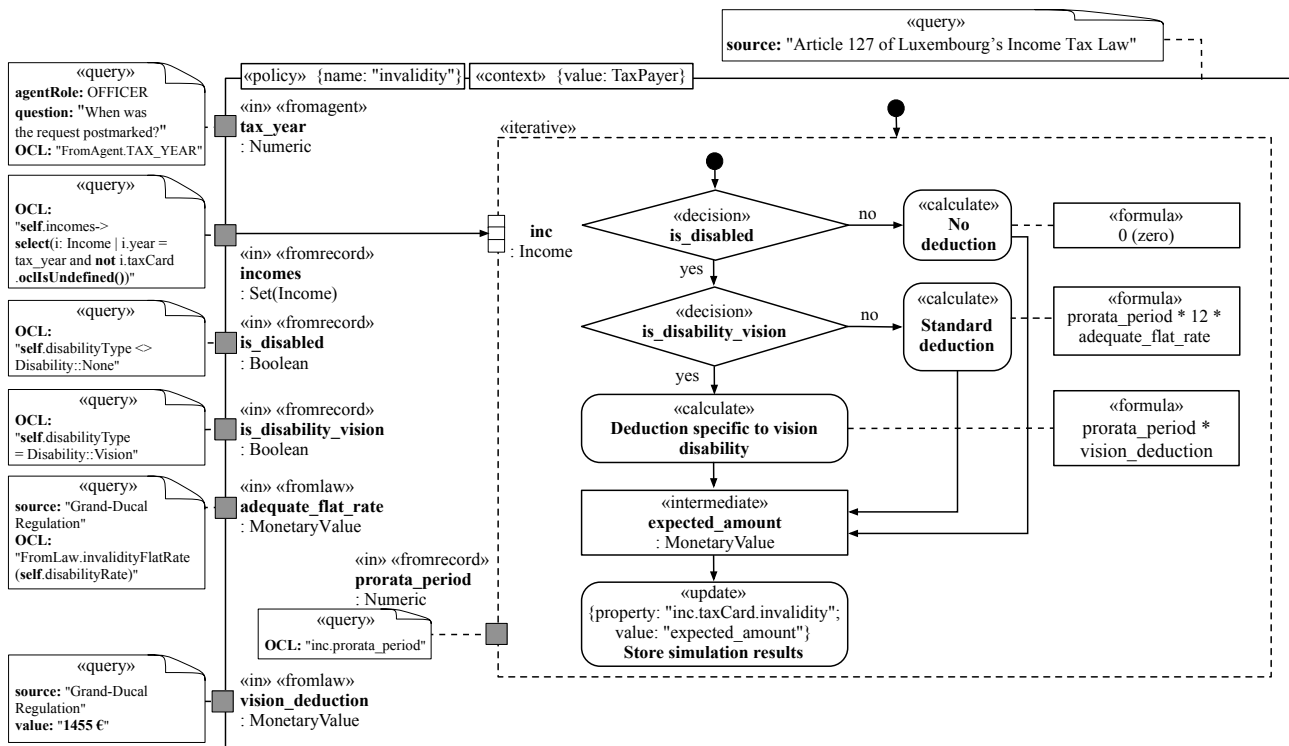


Figure 4.2. (Simplified) Policy Model for Calculating Invalidation Deduction

(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.

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.

Figure 4.3. (Simplified) Policy Description for Invalidation Deduction

In Fig. 4.2, we show how we model the above policy using our customized AD notation (see Fig. 3.5). For succinctness, we hereafter refer to the policy model of Fig. 4.2 as ID (Invalidity Deduction). Readers who are already aquatinted with our modeling notation can skip the remainder of the section and go directly to Section 4.3.

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. 4.6).

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. 4.3. 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. 4.3, 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 and as already discussed in Section 3.3, an expansion region is an activity region that executes multiple times over the elements of an input collection [Object Management Group, 2015]. UML defines three execution modes for expansion regions: *iterative*, *parallel*, and *stream* [Object Management Group, 2015]. 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 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.

4.3 Simulation Code Generation

To enable the simulation of a given set of policy models, we transform the models into code (Step 2 in Fig. 4.1). Below, we outline this transformation.

The transformation is an adaptation of the rule-based model-to-text transformation presented in Section 3.4. 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 (Appendix B.1) and rules (Appendix B.2) to the original ones, we do not present the technical details of the transformation in the main body of this section. The main difference between transforming ADs to OCL and transforming them to Java concerns the patterns that we apply to generate code. The Java patterns are presented in Appendix B.3.

Fig. 4.4 shows a fragment of the simulation code generated for the policy model of Fig. 4.2. 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).

```

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);

```

Figure 4.4. Fragment of Generated Code for the Model of Fig. 4.2

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

4.4 Expressing Population Characteristics

In this section, we present our UML profile [Object Management Group, 2011b] for capturing the probabilistic characteristics of a population. This profile is the basis for Step 3 of our simulation framework (Fig. 4.1). The profile, which extends UML class diagrams, is shown in Fig. 4.5. 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. 4.6. Rectangles with thicker borders in Fig. 4.6 are constraints (not to be confused with classes). References to Fig. 4.5 for the stereotypes and Fig. 4.6 for the examples are not repeated throughout the section.

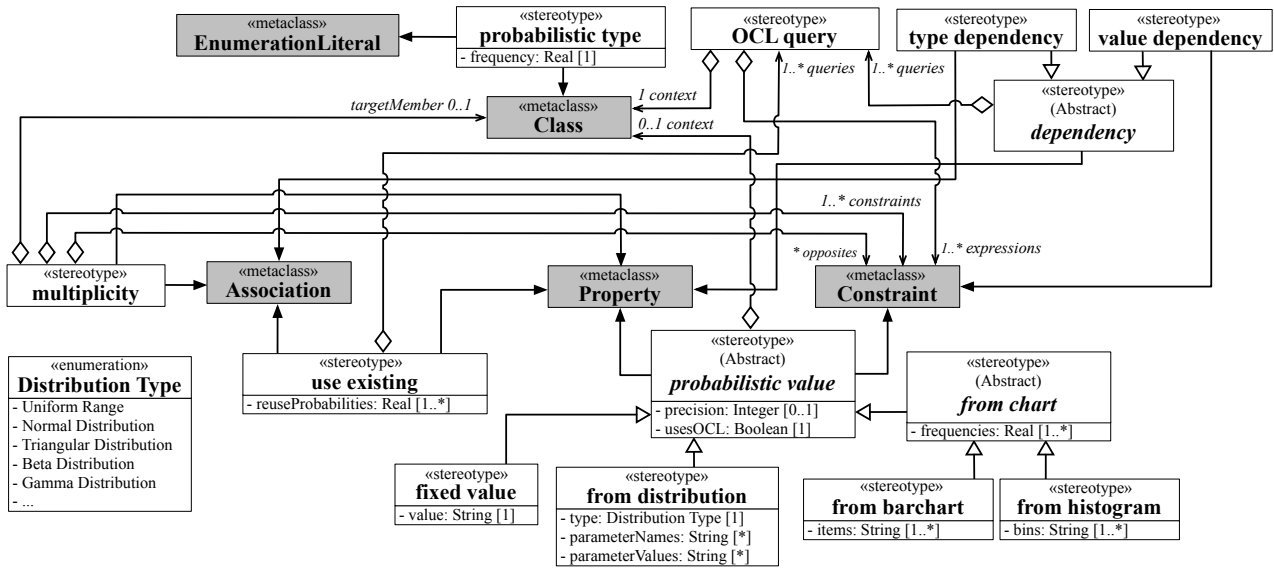


Figure 4.5. Profile for Expressing the Probabilistic Characteristics of a Population

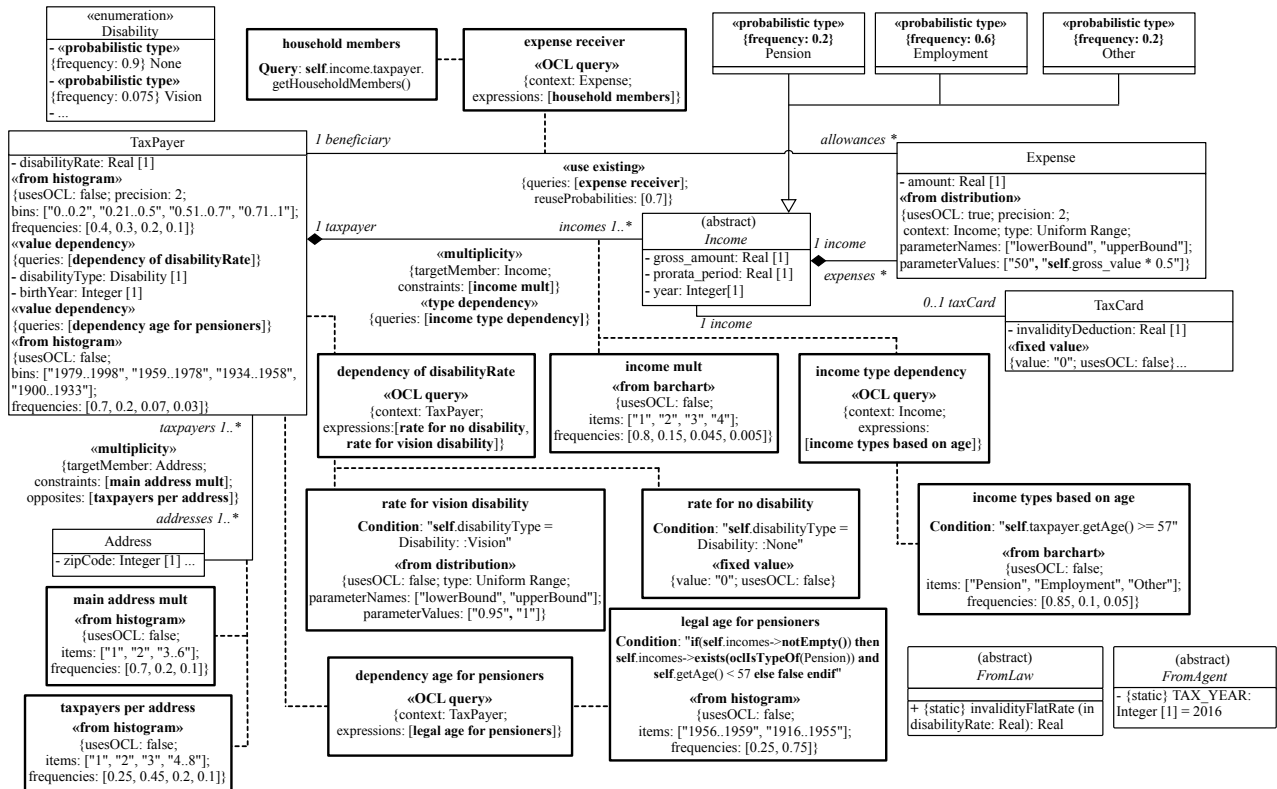


Figure 4.6. Partial Domain Model for Luxembourg's Income Tax Law Annotated with Probabilistic Information

- **«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, \dots, q_n of OCL queries (constraints annotated with «OCL query»), and (2) a collection p_1, \dots, 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 *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. 4.7 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 C.

```

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

```

Figure 4.7. Example Consistency Constraint for the Profile of Fig. 4.5

4.5 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. 4.1. An overview of this process is shown in Fig. 4.8. The inputs to the process are: a domain model annotated with the profile of Section 4.4, and the set of policy models to simulate. The process has four steps, detailed in Sections 4.5.1 through 4.5.4. In the remainder of this section, any reference to a particular “Step” concerns the steps of the process in Fig. 4.8, rather than the steps of our overall framework (Fig. 4.1).

4.5.1 Domain Model Slicing

In Step 1 of the process in Fig. 4.8, *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

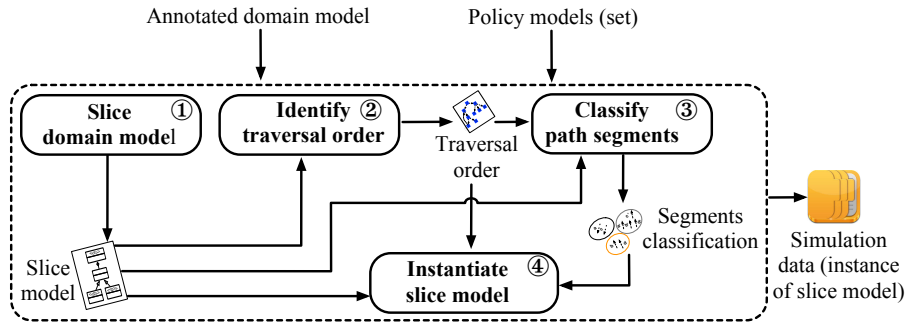


Figure 4.8. Overview of Simulation Data Generation

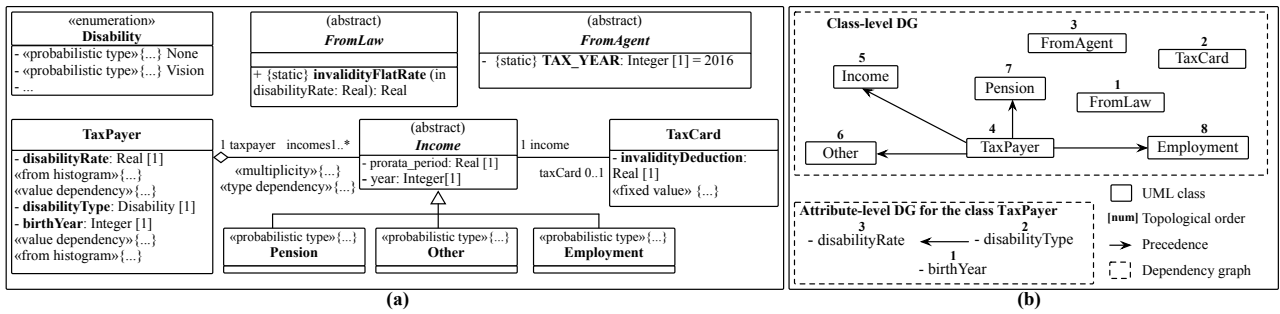


Figure 4.9. (a) Excerpt of Slice Model for Simulating the Policy Model of Fig. 4.2, (b) Topological Sorting of Elements in (a)

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. 4.9(a), we show an example slice model, obtained from the domain model of Fig. 4.6 specifically for the purpose of simulating the policy model in Fig. 4.2, named ID. Among other elements, the *Expense* class has been excluded from the slice because, to simulate the policy model of Fig. 4.2, we do not require instances of *Expense*. To avoid clutter in Fig. 4.9(a), we have not shown the constraints. All the constraints from the domain model of Fig. 4.6 except four are part of the slice model. The four constraints excluded from the slice model of Fig. 4.9(a) are: *main address mult*, *tax-payers per address*, *expense receiver*, and *household members*. These four constraints are not relevant for the simulation of ID.

The slice model of Fig. 4.9(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.

4.5.2 Identifying a Traversal Order

In Step 2 of the process in Fig. 4.8, *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* [Cormen et al., 2009] 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. 4.9(b), we illustrate DGs and topological sorting over the slice model of Fig. 4.9(a). The upper part of Fig. 4.9(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. 4.6), specifically by the OCL constraint named *income types based on age*. Since the instantiation of *TaxPayer* should precede that of *Income*, there are precedence edges from *TaxPayer* to all *Income* subclasses as well.

The edge in the attribute-level DG of Fig. 4.9(b) is induced by the «value dependency» stereotype of the attribute *disabilityRate* (Fig. 4.6), specifically by the OCL constraints *rate for vision disability* and *rate for no disability*. The numbers in the DGs of Fig. 4.9(b) denote one possible total ordering for the respective DGs. Computing these orderings is linear in the size of the DGs [Cormen et al., 2009] and thus inexpensive.

If the class-level or any of the attribute-level DGs are cyclic, topological sorting will fail, indicating that the stereotypes of the slice model are causing cyclic dependencies. In such situations, the cyclic dependencies are reported to the analyst. Such dependencies 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.

4.5.3 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. 4.8, *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. 4.2, the root would be *TaxPayer*. When simulation involves multiple policy models, we pick as root the context class from which all other context classes can be reached via containment associations (i.e., compositions or aggregations). For example, if the model of Fig. 4.2 is to be simulated alongside another policy model whose context is *Expense*, the root would still be *TaxPayer* as *Expense* is reachable from *TaxPayer* through 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. 4.9(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. 4.9(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. 4.9(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.

4.5.4 Instantiating the Slice Model

The last step of the process of Fig. 4.8, *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. 2, 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. 2, has to be instantiated. To do so, SDG selects (L. 6) and instantiates (L. 9) a *non-abstract* class from the following set: $\{C\} \cup \{\text{all descendants of } C\}$. SDG’s selection process (L. 6) is shown in Alg. 3. The selection is based on the «type dependency» and «probabilistic type» stereotypes attached to *C* (L. 1-6 in Alg. 3). As can be seen from Alg. 3, 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. 3).

To minimize potential deviations from the specified population characteristics, Alg. 3 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. 3). These adjustments are aimed at better handling conditional probabilities. To illustrate, consider the classes *Employment*, *Pension*, and *Other* in Fig. 4.6. 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. 4.6. 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 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. 4.6 would yield the following: -20% for *Employment*, 5% for *Pension*,

Alg. 2: Simulation Data Generator (SDG)

Inputs : (1) a slice model \mathcal{S} ; (2) a class $C \in \mathcal{S}$ to instantiate; (3) the orderings, \mathcal{O} , from Step 2; (4) path segment classifications, \mathcal{P} , from Step 3; (5) the last traversed segment or attribute, $source \in \mathcal{S}$ (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  $\leftarrow$  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  $\leftarrow$  CS( $\mathcal{S}$ ,  $C$ , source)
7 if (chosen is null) then return null
8 else
9   | res  $\leftarrow$  Instantiate (chosen)
10  | Let attributes be the set of chosen's attributes
11  | attributes  $\leftarrow$  SortAttributesByOrder (attributes,  $\mathcal{O}$ )
12  | atts_now  $\leftarrow$  RemoveUnreadyStereotypes (attributes, res)
13  | atts_after  $\leftarrow$  RemoveReadyStereotypes (attributes, res)
14  | foreach (att  $\in$  atts_now) do
15  |   | AVG( $\mathcal{S}$ , att,  $\mathcal{O}$ ,  $\mathcal{P}$ , unsat)
16  | Let paths be the Safe and PotentiallyUnsafe outgoing segments from chosen
17  | foreach (seg  $\in$  SortSegmentsByOrder (paths,  $\mathcal{O}$ )) do
18  |   | nextC  $\leftarrow$  target class of seg
19  |   | mult  $\leftarrow$  Attempt «multiplicity» of seg
20  |   | if (mult is null) then
21  |     | mult  $\leftarrow$  random number from multiplicity range of seg
22  |     | Let objects be an (initially empty) set of instances
23  |     | for (i  $\leftarrow$  0; i < mult) do
24  |       | Let obj be an instance (initially empty)
25  |       |  $\mathcal{P}' \leftarrow \mathcal{P}$ 
26  |       | if (seg is PotentiallyUnsafe in  $\mathcal{P}$ ) then
27  |         | obj  $\leftarrow$  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  $\mathcal{P}'$ 
31  |         | if (obj is not null) then
32  |           | objects.add(obj)
33  |         | else
34  |           | if ((unsat.get(seg) \ res.seg) =  $\emptyset$ ) then
35  |             | objects.add(SDG( $\mathcal{S}$ , nextC,  $\mathcal{O}$ ,  $\mathcal{P}'$ , seg, unsat))
36  |           | else
37  |             | obj  $\leftarrow$  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  $\leftarrow$  Attempt «multiplicity» of seg's opposite
44  |           | if (op_mult is null) then
45  |             | op_mult  $\leftarrow$  random number from multiplicity range of seg's opposite
46  |             | for (j  $\leftarrow$  0; j < (op_mult - 1)) do
47  |               | add objects.last() to the list of instances in unsat for the key seg
48  |       | foreach (att  $\in$  atts_after) do
49  |         | if (att has at least one stereotype) then
50  |           | AVG( $\mathcal{S}$ , att,  $\mathcal{O}$ ,  $\mathcal{P}$ , unsat)
51  | return res

```

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*,

Alg. 3: 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  $res \leftarrow null$ 
2 if ( $source$  has «type dependency») then
3    $res \leftarrow$  Attempt «type dependency» of  $source$ 
4 if ( $res$  is null) then
5   if ( $C$ 's immediate subclasses have «probabilistic type») then
6      $res \leftarrow$  Attempt «probabilistic type» from  $C$ 
7   else
8      $res \leftarrow$  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  $res$ 

```

Alg. 4: 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  $mult \leftarrow$  Attempt «multiplicity» of  $att$ 
2 if ( $mult$  is null) then
3    $mult \leftarrow$  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  $\mathcal{S}$ ) then
6   for ( $i \leftarrow 0$ ;  $i < mult$ ) do
7      $value \leftarrow null$ 
8      $value \leftarrow$  Attempt «value dependency» of  $att$ 
9     if ( $value$  is null) then
10     $value \leftarrow$  Attempt «probabilistic value» of  $att$ 
11    if ( $value$  is null) then
12       $value \leftarrow$  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   $att \leftarrow att\_values$ 
17 else
18  Let  $att\_objects$  be an (initially empty) set of instances
19  for ( $i \leftarrow 0$ ;  $i < mult$ ) do
20     $att\_objects.add(SDG(\mathcal{S}, typeOf(att), \mathcal{O}, \mathcal{P}, att, unsat))$ 
21   $att \leftarrow att\_objects$ 

```

and $15\% + 80\% \times \frac{15}{5+15} = 75\%$ for *Other*. To facilitate understanding, we summarize in Table 4.1 the calculations that we described above.

Table 4.1. Example of Adaptive Adjustment of Frequencies

	Frequency		
	<i>Employment</i>	<i>Pension</i>	<i>Other</i>
User-specified (Fig. 4.6)	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. 4 (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. 4 determines the required number of values to assign to a given attribute based on the attached «multiplicity» stereotype (L. 1 in Alg. 4). 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. 4). 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. 4). 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. 4). Similar to Alg. 3, the frequencies in Alg. 4 are dynamically adjusted when necessary (L. 14-15 in Alg. 4). 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. 4).

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. 4.9(a). After creating an instance of *TaxPayer*, a value will be immediately assigned to *disabilityRate* by applying the «from histogram» and «value dependency» stereotypes (shown in Fig. 4.6). 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. 3 and Alg. 4.

(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. 4.9(a) also includes the *Address* class from Fig. 4.6 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*.

4.6 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. 4.10 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 4.1, 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/).

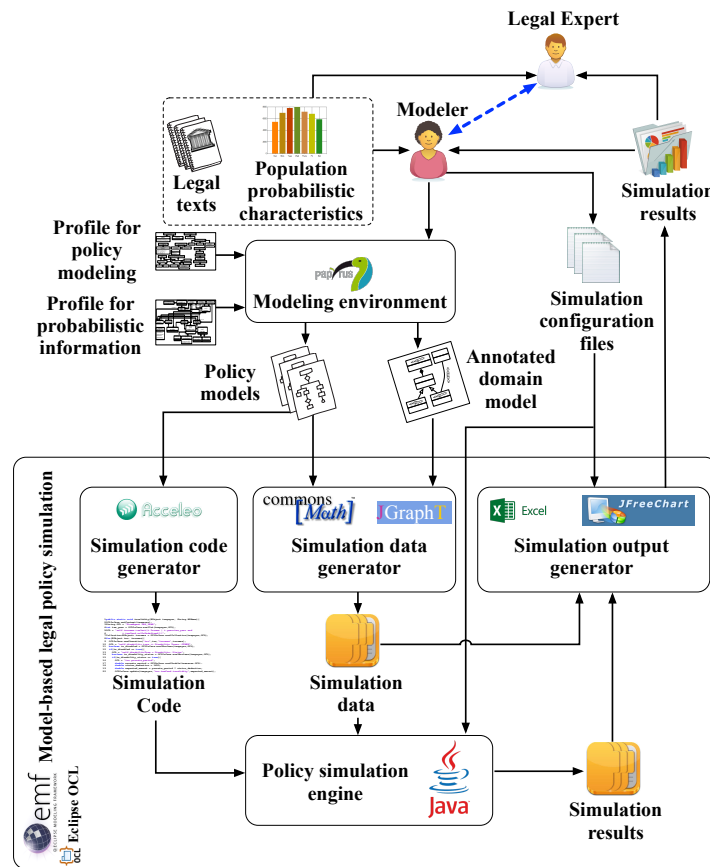


Figure 4.10. PoliSim Architecture

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 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.

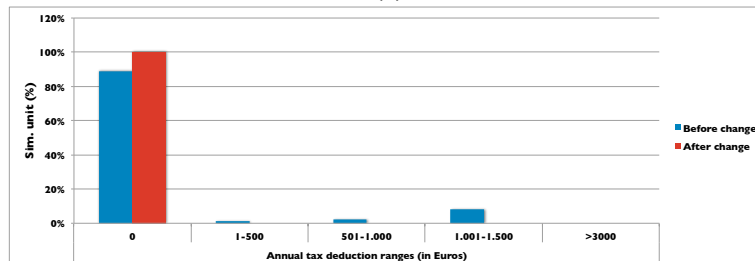
The core components of PoliSim are shown in the rectangle labeled *Model-based legal policy simulation* in Fig. 4.10. All of these components rely on: (1) the Eclipse Modeling Framework (EMF) for manipulating the input models and the simulation data, and (2) EclipseOCL for parsing and evaluating OCL expressions (eclipse.org/ocl/).

The *Simulation code generator* component uses Acceleo (eclipse.org/acceleo/) for transforming policy models into simulation code (discussed in Section 4.3). 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 data generator* implements the process discussed in Section 4.5. This component relies mainly on: (1) JGraphT (jgrapht.org) for graph analysis, including topological

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	Other	Other	50958.84	45300	511	511	0
	Income 2	invalidityDeduction (TaxCard)	0	0	One_A	Pension	20651.4	20650	0	0	0

(a)



(b)

Figure 4.11. Excerpt of Simulation Results Presented as a (a) Spreadsheet and (b) Chart

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 4.5).

The *Simulation output generator* creates the final output from the tool. Fig. 4.11 presents the output obtained for a (hypothetical) scenario, where the invalidity policy presented in Section 4.2 has been abolished. As shown by the figure, the output comes in two different formats: spreadsheets and charts. The spreadsheets 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. 4.11(a) shows the difference in revenue before and after the invalidity policy abolishment scenario mentioned above. The chart of Fig. 4.11(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.

4.7 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 execution 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 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. 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. 4.2. Initial versions of these six policy models and the domain model supporting these policies (as well as other policies not considered here) were built for the conducting the evaluation of Chapter 3.

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. 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 models above were then validated with (already-trained) legal experts in a series of meetings totaling ≈ 12 hours. Subsequently, we annotated the (now 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. 4.6 are based on the extracted information, noting that the actual numerical values were rounded up or down to avoid cluttering the figure with long decimal-point values.

To answer the RQs, we ran the simulator (automatically derived from the six policy models) over simulation data (automatically generated by Alg. 2). We discuss the results below. All the results were obtained on a computer with a 3.0GHz dual-core processor and 16GB of memory.

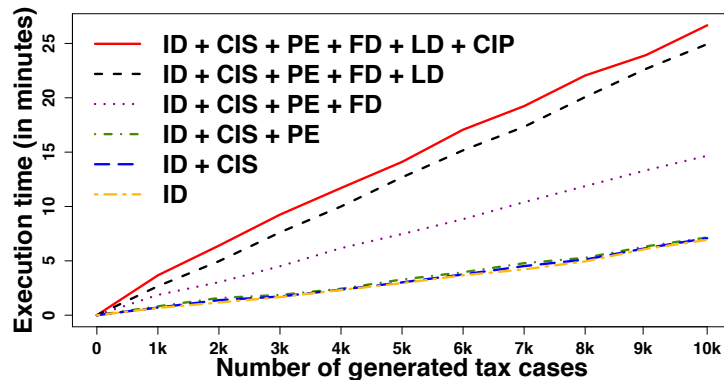


Figure 4.12. Execution Times for Data Generation

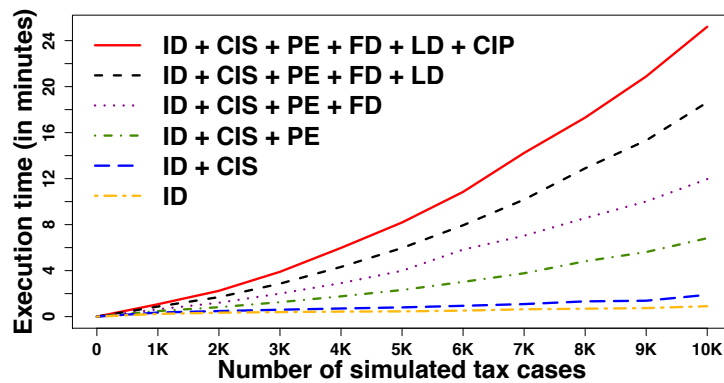


Figure 4.13. Execution Times for Simulation

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 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. 4.12 and 4.13, we show the execution times (average of the five runs) for the data generator and for the simulator, respectively.

As suggested by Fig. 4.12, 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. 4.12. Specifically, the slice models for the six policy sets used in our evaluation, i.e., (1) ID, (2) ID + CIS, ..., (6) ID + CIS + PE + FD + LD + CIP, covered approximately 4%, 5%, 7%, 13%, 20%, and 22% of the domain model, respectively. We note that as more policies are included, the slice model will eventually saturate, as the largest possible slice model is the full domain model.

With regards to simulation, the execution times partly depend on the complexity of the workflows in the underlying policies (e.g., the nesting of loops), and partly on the OCL queries that supply the input parameters to the policies. The latter factor deserves attention when simulation is run over a large instance model. Particularly, OCL queries containing iterative operations may take longer to run as the instance model grows. The non-linear complexity seen in the fifth and sixth curves (from the bottom) in Fig. 4.13 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. 4.12 and 4.13, 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.

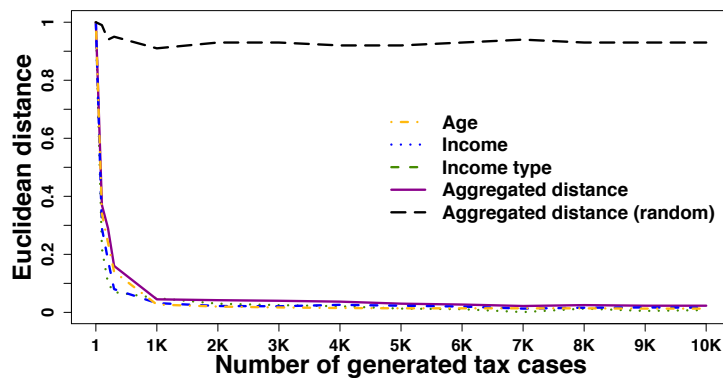


Figure 4.14. Euclidean Distances between Generated Data & Real Population Characteristics

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 [Cha, 2007]. Fig. 4.14 presents Euclidean distances for the age, income, and income type histograms as well as the Euclidean distance for the normalized aggregation of the three. As indicated by the figure, the Euclidean distance for the aggregation 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.

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 [Corder and Foreman, 2014], 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 4.2 shows the results, with P_1, \dots, 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 [Corder and Foreman, 2014]. In our evaluation, we did not employ additional tests given the clear results obtained with the KS test ($p > 0.8$ everywhere).

The results in Table 4.2 provide confidence that our data generator yields consistent data across different runs.

Table 4.2. Pairwise Kolmogorov-Smirnov (KS) Test Applied to Five Samples (P_1, \dots, 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

4.8 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 4.7.

4.8.1 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 4.4 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 [Jackson, 2012] and UML2CSP [Cabot et al., 2014], 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 4.7, our data generation strategy meets the above scalability requirement. However, the strategy has limitations: (1) As noted in Section 4.5.2, 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 4.7). 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.¹ 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 4.7 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 4.5.4, 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. 4.2, 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. 4.3). 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. 4.6). 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

¹We note that the traversal strategy of Alg. 2, discussed in Section 4.5.2, 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.

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

4.8.2 Threats to Validity

Construct and external validity are the most relevant validity considerations for our evaluation of Section 4.7. 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 4.7), 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.

4.9 Related Work

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

Simulation of legal policies. Several policy simulation tools exist in the field of applied economics. For example, SYSIFF [Bourguignon et al., 1988, Canova et al., 2009] in France, SPSS/M [Statistics Canada, 2015] in Canada, and POLIMOD [Sutherland, 1995] in the UK have been used in tax-benefit simulation for several years. Most recently and at a European level, EUROMOD [Figari et al., 2015] 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 [SAS Institute, 1996a] have been customized for use in policy simulation and prediction [SAS Institute, 1996b].

All the above tools use a combination of spreadsheets, hard-coded formulae, and programming languages for implementing legal policies. As we argued in the introduction of this chapter, 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 [Object Management Group, 2013], e.g., as used by Mijatov et al. [Mijatov et al., 2015] 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 [IBM, 2003a], IBM Rhapsody Simulation Toolkit [IBM, 2003b], Papyrus' Moka [Papyrus, 2009], and MagicDraw's Cameo [No Magic, 2005]. 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 [Iqbal et al., 2013] and system configuration [Behjati et al., 2014]. Several instance generation approaches are based on exhaustive search, using tools such as Alloy [Jackson, 2012] and UML2CSP [Cabot et al., 2014]. 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 4.8, do not scale well in our application context.

Another class of instance generation approaches rely on non-exhaustive techniques, e.g., graph-based rules [Gogolla et al., 2005], metaheuristic search [Ali et al., 2013], mutation analysis [Di Nardo et al., 2015], and model cloning [Bousse et al., 2014]. 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 Mougnot et al. [Mougnot et al., 2009] and Hartmann et al. [Hartmann et al., 2014]. Mougnot 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, Mougnot 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.

4.10 Conclusion

In this chapter, 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.

Chapter 5

Simulating the Impact of Changes in Legal Policies (Case Study)

Software systems are increasingly subject to laws and regulations. To develop a legally compliant system, software engineers need to interpret, typically in collaboration with legal experts, the relevant legal texts and derive from these texts legal requirements that the system under development must fulfill. A common way for representing legal requirements is through *modeling*. Models offer intuitive means for communication between software engineers and legal experts, thus facilitating building a shared understanding of the legal requirements. Models further provide a useful basis for systematic and automated analysis of legal compliance.

Several strands of work employ models for elaborating legal requirements and assessing whether and to what extent these requirements are met by a given system. These strands include the large body of research concerned with the application of goal models to laws and regulations, e.g., [Ghanavati et al., 2014, Ingolfo et al., 2014], as well as a number of conceptual modeling techniques aimed at representing the semantic metadata of legal texts, such as key legal abstractions and modalities, e.g., [Zeni et al., 2015, Breaux, 2009a], and the structural makeup of legal texts, e.g., [Emmerich et al., 1999b, Breaux, 2009b, Sannier et al., 2017].

In recent years, we have been exploring the application of modeling for automated analysis of compliance to the tax law. The tax law, along with several other categories of law such as social benefits and customs duty laws, are highly operationalized. These laws typically structure the compliance requirements into detailed *policies*. Public administration IT systems need to implement and comply with these policies.

In Chapter 3, we proposed a UML-based approach for expressing legal policies. An observation that we made in this context is that the resulting policy models have added analytical value beyond what we initially had in mind, which was to use the models for compliance testing of government IT systems. An interesting additional application of the policy models is for *simulation*. The goal of simulation is to predict the impact of a set of proposed changes to laws and regulations, and ensure

that the proposed changes will bring about the desired outcomes without unwanted side effects. This new application of policy models prompted us to develop a model-based simulator (see Chapter 5), drawing on the same modeling approach developed previously for legal compliance testing.

In this chapter, we report on a case study where we apply, for the first time, our simulator for analyzing the impact of a *real* legal change proposal. The change proposal concerns the “joint taxation” policy in Luxembourg’s Income Tax Law. This policy, which is also known as “income splitting” in some other countries, enables spouses to attribute, under certain conditions, some of the income of the higher-earning spouse to the lower- (or non-)earning spouse. The policy often leads to a reduced overall tax obligation for the spouses. The Government of Luxembourg is currently considering a proposal to abolish (repeal) joint taxation. Our case study employs simulation in order to examine how this potential reform is likely to impact personal income taxes. The case study is motivated by the following Research Questions (RQs):

RQ1: Can we bring together and model all the information necessary for performing a real-world simulation scenario? While the feasibility of building individual policy models was examined in Chapters 4 and 5, we did not previously investigate whether one can build a seamless set of models that need to be considered together in a real-world simulation scenario. Furthermore, our simulator includes a probabilistic data generator to create artificial data in situations where access to real simulation data is restricted (the situation we have to deal with in our case study), or where real data is unavailable, e.g., when we are simulating a new policy for which no historical data exists. RQ1 aims to study the feasibility of building a coherent set of policy models for joint taxation and other closely-related policies. This RQ further looks into whether we can instrument our data generator with sufficient probabilistic guidance to generate data for simulating joint taxation.

RQ2: Are the simulation results credible? RQ2 is aimed at comparing the results produced by our simulator against publicly-available statistics on tax contributions from taxpayers in different income brackets. Indirectly, RQ2 develops confidence about two important factors: First, is the level of abstraction at which we express the policy models a good fit for our analytical purpose? And second, does the artificial data generated by our simulator serve as a good replacement for real data, i.e., taxpayer records, to which we have no access? Our answer to RQ2 will be based on comparing the results of simulating the status quo, i.e., the situation where joint taxation is enforced, against *current* tax contribution statistics. The analysis that we performed in reality for our public service partners was to compare the status quo against a potential future where joint taxation has been abolished [Soltana et al., 2016b]. Using our simulator for enacting this scenario does not add new conceptual element to our framework; to do so, we simply provided a current (i.e., with joint taxation) and a future (i.e., without joint taxation) set of policies, subjected the two sets to the same simulation data, and quantified the difference between the simulation results. Since we naturally do not know at this stage how accurate our predictions are, we rely on the status quo for analyzing the credibility of the simulation results.

In addition to addressing the above two RQs, we summarize in this chapter the lessons that we learned from our experience. The lessons cover a number of important considerations in relation to

traceability between models and legal texts, and making policy models easier to understand for legal experts.

We believe that our work in this chapter is useful to the Requirements Engineering community in three related ways: First, we demonstrate how models of legal requirements built for enabling compliance analysis can further be exploited, with minor adaptations, for another important analytical purpose: simulation. Modeling in general, and requirements modeling in particular, require upfront investment when adopted in industrial settings. Being able to use the same models for multiple purposes makes modeling more cost-effective, in turn contributing to the wider adoption of models in industry. Second, one can use simulation as a vehicle for validating legal requirements models by comparing the results of simulation against expectations. Since, in the context of our work, compliance analysis and simulation build on the same models, validation via simulation contributes to building more accurate models for compliance analysis. Finally, while our RQs and lessons learned are naturally oriented around simulation due to the nature of our case study, we anticipate our conclusions and observations, e.g., about modeling effort and the accuracy of analysis results, to generalize to a large extent to compliance analysis.

Structure. Section 5.1 outlines our simulation framework and tool support. Section 5.2 describes the design of our case study. Section 5.3 presents the case study results and answers our motivating RQs, stated earlier. Section 5.4 reflects on the lessons learned. Section 5.5 compares with related work and Section 5.6 concludes the chapter.

5.1 Approach

Fig. 5.1 summarizes our simulation approach that we discussed in Chapter 4. The approach takes as input a set of policy models that need to be simulated. Policy models are interpretations of the legal texts and provisions that are relevant to the intended simulation activity. For the taxation domain, the policies are concerned primarily with the calculation of quantities such as credits, deductions and taxes. We exemplify in Section 5.1.1 one of the policy models that we use in our case study. Although not shown in Fig. 5.1, all policy models need to be validated by legal experts before simulation. The

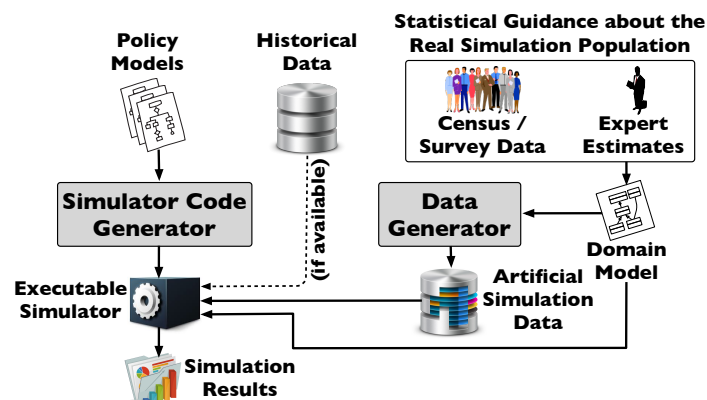


Figure 5.1. Overview of our Simulation Approach

goal of the validation is to ensure that the models are a faithful representation of the relevant legal requirements. We describe our validation process in Section 5.2.2.

The input policy models are automatically transformed into an executable simulator. This process is transparent to the user, thus eliminating the need to manually review or manipulate any software code. The data to be processed by the simulator can be provided through two alternate means: If historical data, e.g., a database of tax records, is available, this data can be fed directly to the simulator for processing. Alternatively, if historical data is unavailable, inaccessible, or incomplete, our approach automatically generates artificial data for simulation.

We recall that the basis for automatic data generation is a domain model. The main purpose of a domain model in our approach is to enable guiding the data generator towards creating realistic data, i.e., data whose characteristics are as close as possible to those of the population being simulated. Specifically, we use the domain model as a container for providing statistical guidance to the data generator (see Section 4.4). This guidance is typically derived from existing census and survey statistics. Expert estimates may be further utilized for exploring future (uncertain) contingencies, or for addressing possible gaps in the statistics.

Once the simulator has processed the simulation data (either historical or artificial), the results are aggregated into spreadsheets and charts and provided to the user. Our approach further supports simulation result differencing. This means that the user can provide an original and a revised set of policy models, simulate both sets, and compare the simulation results to quantify the impact.

In this chapter, we consider our simulation approach exclusively from the perspective of users. We therefore address only the inputs to and the outputs from our simulation approach, rather than the technical machinery behind the approach. A detailed treatment of the technical components of the approach are presented in Chapter 4. Below, we present the main policy model that is directly impacted by the reform of this case study and exemplify the use of probabilistic notations to guide data generation in Sections 5.2 and 5.1.2, respectively. Note that all policy models and an excerpt of the underlying domain model are available in [Soltana et al., 2016b].

5.1.1 Policy Model Example

In our context, policy models capture the workflow for realizing a given policy using a customized form of UML activity diagrams. To illustrate, Fig. 5.2 shows a policy model, named Tax Class Categorization (TCC), whose function is to assign a tax class to a taxpayer, as per the provisions of Luxembourg’s Income Tax Law (*LITL*). This policy is one of the policies involved in our case study (Section 5.2). *LITL* defines three tax classes: tax class 1, tax class 1.a, and tax class 2. The tax class determines the taxation rate and formula to apply for calculating taxes due. For example, the lowest tax rates are used for taxpayers belonging to tax class 2. A taxpayer is assigned a tax class based on their personal situation, including among other factors, family and residence status.

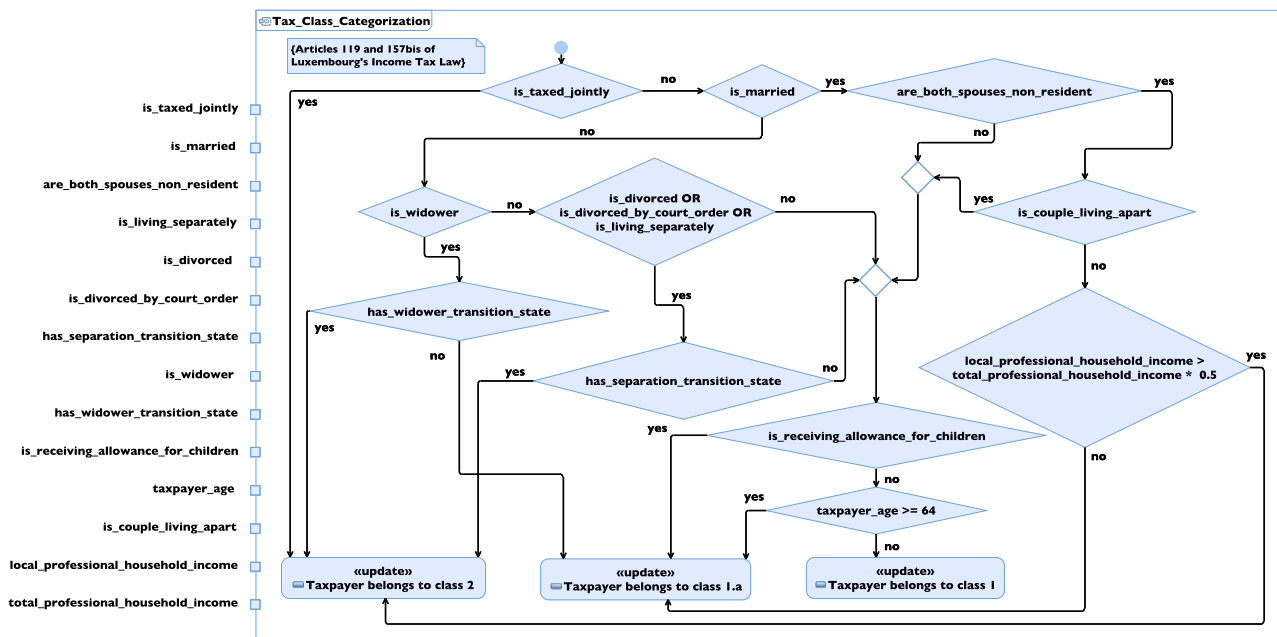


Figure 5.2. Policy Model for Tax Class Categorization (TCC)

To read a policy model, one can first explore the inputs associated with the policy. The inputs are listed on a policy model’s left side. For example, the input *is_married* in the policy model of Fig. 5.2 is “yes” for a given taxpayer if and only if the taxpayer is married. We formalize the inputs to a policy model using expressions written in the Object Constraint Language (OCL) [Object Management Group, 2004]. For example, the OCL expression for *is_married* is: `self.isMarried(Constants.TAX_YEAR)`, where `self` refers to the taxpayer being simulated and `Constants.TAX_YEAR` is the current taxation year, represented as a static attribute of a class named `Constants`. The OCL expressions for the inputs are not shown in the policy model of Fig. 5.2.

As we mentioned earlier, all policy models, including their inputs, have to be validated by legal experts. However, we have observed that legal experts often find it difficult to work directly with formal languages such as OCL (see Section 3.1). To tackle this issue, we complement the OCL definitions of the inputs by a glossary. The glossary contains, for each input, an intuitive description written in natural language. Table 5.1 shows the glossary describing the inputs to the policy model of Fig. 5.2.

Once familiar with the inputs to a policy model, one can proceed to explore the workflow of a policy from the initial node, denoted by a circle, and follow the different paths leading to the update function(s), denoted by an «update» stereotype. Typically, an update function is an action performed within the workflow of a policy in order to record the outcome resulting from the simulation of that policy.

To illustrate, we describe the workflow of the TCC policy model in Fig. 5.2. The first criterion to consider when assigning a tax class to a taxpayer is whether the taxpayer is taxed jointly with a spouse. This criterion is captured by the diamond-shaped decision node. Depending on the value of the input *is_taxed_jointly*, a suitable path (yes or no) is taken out of the decision node. A taxpayer who

Table 5.1. Glossary for the Inputs to the Policy Model of Fig. 5.2

Input name	Description
<i>is_taxed_jointly</i>	Yes if a given taxpayer is taxed jointly with another taxpayer; otherwise, no. The value of this input is determined via the application of another policy model: joint taxation.
<i>is_married</i>	Yes if a given taxpayer is married; otherwise, no.
<i>are_both_spouses_non_resident</i>	Yes if a given couple are both non-residents; otherwise, no.
<i>is_living_separately</i>	Yes if a given taxpayer is “de facto” separated (<i>séparation de fait</i> in French); otherwise, no.
<i>is_divorced</i>	Yes if a given taxpayer is divorced by mutual agreement; otherwise, no.
<i>is_divorced_by_court_order</i>	Yes if a given taxpayer is divorced by court order; otherwise, no.
<i>has_separation_transition_state</i>	Yes if a given taxpayer is in a transition state after separation; otherwise, no. The transition state is granted when a taxpayer’s date of separation is within the past three years.
<i>is_widower</i>	Yes if a given taxpayer is a widower; otherwise, no.
<i>has_widower_transition_state</i>	Yes if a given taxpayer is in a transition state due to having been widowed within the past three years; otherwise, no.
<i>is_receiving_allowance_for_children</i>	Yes if a given taxpayer is receiving some benefit or allowance for their children; otherwise, no. Examples of allowances include childbirth benefit and child allowance.
<i>taxpayer_age</i>	Age of a given taxpayer.
<i>is_couple_living_apart</i>	Yes if a given couple do not live at the same address; otherwise, no.
<i>local_professional_household_income</i>	Sum, for a given household, of the professional incomes taxed in Luxembourg. Categories of professional income are defined by <i>Articles 14, 61, 91 and 95 of LITL</i> .
<i>total_professional_household_income</i>	Sum, for a given household, of all professional incomes.

is taxed jointly is assigned tax class 2. If a taxpayer is not taxed jointly, then other criteria, including marital status, will determine tax class 2 eligibility. For instance, a widower taxpayer belongs to tax class 2 if they have lost their spouse in the past three years (*has_widower_transition_state*). Similarly, a taxpayer who has had a divorce in the past three years belongs to tax class 2. Finally, married non-resident taxpayers who are living at the same address belong to tax class 2, provided that they realize more than half of their professional income in Luxembourg.

Taxpayers who do not belong to tax class 2 might belong to tax class 1.a. Specifically, tax class 1.a covers: (1) taxpayers who are receiving some child allowance, (2) taxpayers who are aged at least 64, (3) widowers who have lost their spouse prior to the last three years, and (4) non-resident married taxpayers who are not taxed jointly, but who are living at the same address and realizing less than half of their professional income in Luxembourg. Taxpayers who do not belong to either tax class 2 or tax class 1.a are assigned tax class 1.

5.1.2 Example of Statistical Guidance for Generating data

We capture a domain model using UML class diagrams. As we noted earlier, the main role of a domain model in our approach is as a vehicle for providing statistical guidance to our data generator. To express statistical guidance over a domain model, we extend UML class diagrams with explicit probabilistic notions, including relative frequencies, distributions, and conditional probabilities (see Chapter 4).

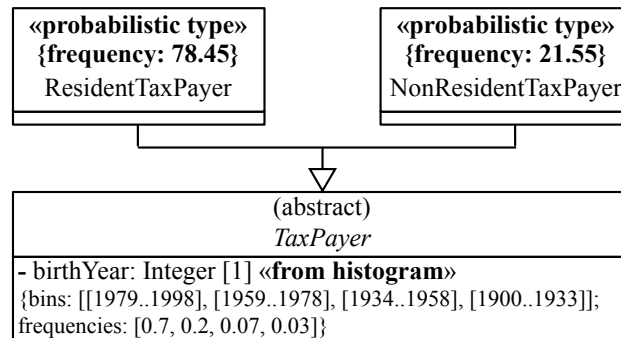


Figure 5.3. Domain Model Fragment Extended with Statistical Guidance

Fig. 5.3 illustrates the application of some of the above probabilistic notions on a small fragment of the domain model built as part of our case study. The «probabilistic type» stereotypes applied to the specializations of the *TaxPayer* class state that $\approx 78\%$ of the taxpayers are resident and the remainder are non-resident. The «from histogram» stereotype attached to the *birthYear* attribute provides, via a histogram, the birth year (age) distribution of the taxpayers.

5.2 Case Study Design

In this section, we report on a case study where we apply the simulation approach presented in Section 5.1 for analyzing the impact of the (potential) abolishment of the joint taxation policy in Luxembourg. Joint taxation is done by considering spouses who fulfill certain eligibility criteria as a single tax unit. The effect is that the difference between the income of the higher-earning spouse and the lower- (or non-)earning spouse is split between the two, thus potentially giving the higher-earner a lower tax rate and the lower-earner a higher tax rate. In many circumstances, the policy yields a lower overall tax obligation for the spouses. The case study was conducted in collaboration with the Government of Luxembourg.

5.2.1 Case Selection

Our main criterion for case selection was to work on a real and practically-relevant simulation scenario. As long as this criterion was met, our case selection strategy was *opportunistic*. Once the abolishment of joint taxation was suggested to us as a possible case study, we conducted a preliminary investigation to ensure that the case study had a realistic chance of being brought to completion.

To this end, we considered three factors: (1) whether the (at this stage, ballpark) level of modeling effort is reasonable given the research team’s resource constraints, (2) whether we have access to legal experts who can validate the models built throughout the case study, and (3) knowing a priori that we will not have access to real tax data, whether there are publicly-available sources where we are likely to find relevant statistical information for guiding the data generation process. We started conducting the case study after initial positive indications about the factors above.

5.2.2 Data Collection Procedure

Data collection was targeted at: (1) building policy models and a domain model for the case study, and (2) gathering the relevant probabilistic information and incorporating this information into the domain model.

Policy and domain model construction. The legal experts in our case study directed us to the legal provisions (segments of legal texts) relevant to the intended simulation activity and informally described the interpretation of these provisions to us. Subsequently, we followed our proposed modeling methodology (see Chapter 3) for modeling the policies and the domain.

To validate the developed models, we held regular walkthrough sessions with the legal experts. The experts had already received training on how to read and understand simple activity and class diagrams. For each policy model, the legal experts would first review the glossary definitions for the inputs to that model (see Section 5.1.1). The experts would then go through the workflow of the policy model and provide feedback. The validation of the domain model was intertwined with that of the policy models. Specifically, for each policy model, we discussed with the experts the pertinent classes and associations of the domain model and obtained feedback. The validation process was iterative. We kept refining and (re-)validating the models until the legal experts deemed the models to be complete and precise representations of the underlying legal provisions.

Extending the domain model with statistical information. Once the domain model had sufficiently stabilized, we started annotating its elements (classes, attributes, associations) with statistical information. Specifically, for each element in the domain model, we did the following: First, we examined the objective data sources that were available to us, notably public census data and statistics published by the Government, for a suitable distribution. If a suitable distribution could not be found, we asked the experts for estimates and suggestions. In addition, we captured in our domain model any dependencies we knew of between the elements, either from the objective sources or from the experts’ domain knowledge. An example dependency is the following: “If a taxpayer has a pension income, the taxpayer must be older than 57” (dependency between income type and age). We capture such dependencies in our domain model as conditional probabilities. Conditional probabilities provide more flexibility than deterministic rules for dealing with dependencies. For the dependency above, we could, for example, state using conditional probabilities that “If a taxpayer has a pension income, then the taxpayer’s age should be between 57 and 65 with a probability of 90% and should be greater than 65 with a probability of 10%”. A complete list of the statistical information in our case study is provided in Section 5.3.

5.2.3 Analysis Procedure

Analyzing the feasibility of modeling. To analyze whether our simulation approach has a realistic chance of being used in practice, we kept track of the level of modeling effort required by our approach. We further monitored for potential situations where our modeling notation did not provide adequate expressive power. Finally, we examined whether we could enrich our domain model with sufficient statistical information for guiding simulation data generation.

Analyzing the quality of simulation input data. Before using an automatically-generated data sample for simulation, we needed to ensure that the sample is of reasonable quality. We did so by performing sanity checks that compared the statistical distributions of the sample against those of the real taxpayers population, i.e., the distributions attached to the domain model. An example sanity check is to verify whether the relative distribution of resident versus non-resident taxpayers in a generated sample indeed matches what we know about the real population in Luxembourg.

All the statistical distributions in our case study are histograms. For sanity checking of the generated data samples, we thus needed a metric for comparing histograms. We chose the (normalized) Euclidean distance [Cha, 2007] for this purpose. This metric, whose value is between 0 and 1, measures how far two histograms are from one another. For a given pair of histograms, a Euclidean distance close to 0 means that the histograms are well-aligned; whereas a Euclidean distance close to 1 means that the histograms differ significantly.

For example, suppose that the distribution of taxpayers' residential status in the generated simulation data is: 90% resident versus 10% non-resident. As we showed earlier in the domain model fragment of Fig. 5.3, the actual proportions for the real population are $\approx 78\%$ resident versus $\approx 22\%$ non-resident. When these two distributions are represented as histograms, the Euclidean distance between them is ≈ 0.2 .

We deem the sanity check for a specific distribution derived from an automatically-generated data sample as being successful if the Euclidean distance between that distribution and the corresponding actual distribution (attached to the domain model) is sufficiently small. We use a failure threshold of 0.1 to decide whether a sanity check succeeds or fails. Checks that yield a distance below the threshold succeed; checks that yield a distance above the threshold fail. If a generated sample failed any of the sanity checks, we repeated data generation with a larger sample size. We increased the sample size until all sanity checks passed.

Analyzing the credibility of simulation results. After completing a simulation run, we verified the credibility of the simulation results by comparing the results against the aggregate information we had about the tax contributions of households in different income brackets. Similar to the sanity checks performed on the simulation input data, we used the Euclidean distance metric for determining how close the simulation results were to reality. As we noted earlier in chapter when presenting RQ2, the basis for assessing simulation credibility is the status quo. In other words, we do not assess the credibility of our prediction about the impact of abolishing joint taxation, since, naturally, no data exists

about the impact of this potential future change. This said, achieving reasonable accuracy in simulating the status quo is an indication of the adequacy of our modeling methodology and automated data generation process. This in turns provides confidence about the credibility of the predictions made using our approach.

5.3 Results and Discussion

The simulation of joint taxation involved the construction of three policy models and a domain model. The policy models are: (1) *Joint Taxation (JT)* to determine whether a taxpayer is eligible for joint taxation, (2) *Tax Class Categorization (TCC)*, shown in Fig. 5.2, to determine a taxpayer's tax class, and (3) *Extra-Professional Deduction (EPD)*, to grant, under certain conditions, a special deduction to a taxpayer who is taxed jointly. JT, TCC and EPD respectively have, 111, 93, and 73 elements, where an element can be an input, output, decision, action, flow, intermediate variable, expansion region, or a constraint. The domain model has 16 classes, 4 enumerations, 6 associations, 11 generalizations, and 24 attributes.

In addition to the models above, our case study used a pre-existing implementation of the formulas for calculating taxes due. These formulas could have been captured as policy models. Nevertheless, since our case study did not involve any modifications to these formulas, we elected to use the available implementation. This decision was motivated by avoiding the need to re-validate the formulas, which had been already extensively tested.

5.3.1 RQ1: Can we bring together and model all the information necessary for performing a real-world simulation scenario?

We consider three aspects for answering RQ1: (1) whether the modeling effort is practical, (2) whether our modeling notation is expressive enough, and (3) whether we could successfully provide the input (statistical information) necessary for the generation of simulation data.

With regard to the modeling effort, and as noted in Section 5.2.3, we kept track of the effort spent on constructing the models (from scratch) and validating them. Table 5.2 presents the effort for model construction and validation. The effort for constructing the models is inclusive of the time spent on reading the relevant legal texts and preparing glossaries (exemplified in Table 5.1) for the policy model inputs. Modeling the policies relevant to joint taxation (JT, TCC, EPD) took 15 person-hours (ph). Building the domain model took 7 ph, including the effort for annotating the domain model with the statistical guidance for data generation (discussed later). The validation of the models with legal experts took 5 ph in total, of which 3 ph was spent on the policy models and the remaining 2 ph on the domain model. The above results suggest that the modeling effort for our case study was practical.

We note that to simulate the impact of a set of proposed changes to laws and regulations, we need to capture both the original and the modified sets of policies. In our case study, we spent negligible effort on creating the modified set of policies because of the nature of the underlying legal change.

Table 5.2. Effort for Building and Validating the Models

Activity	Model			
	JT	TCC	EPD	Domain Model
Model construction (person hours)	7	5	3	7
Model validation (person hours)	1.5	1	0.5	2

Specifically, abolishing joint taxation entailed: (1) dropping the JT and EPD policy models from the modified set (thus leaving only TCC in the modified set); and (2) a slight modification to TCC in order to link the initial node to the *is_married* decision thus bypassing the *is_taxed_jointly* decision in the original model (see Fig. 5.2). When changes to laws and regulations involve more extensive alterations, e.g., adding new policies, the effort for building the modified set of policy models will be proportional to the extent of the alterations.

The second aspect of RQ1 has to do with the expressiveness of our modeling notation. In our case study, we faced only one situation where an extension to our notation was necessary. More precisely, our original notation defines three possible sources from which an input to a policy model can originate: (1) *records*, e.g., for taxpayers' incomes and expenses; (2) *legal texts*, e.g., for monetary values explicitly written into the text of legal provisions, and (3) *agents*, for any input that needs to be supplied by a human, e.g., an eligibility criterion that should be ascertained by a tax officer.

The above three sources do not consider the situation where an input to a given policy model may correspond to the output from another policy model. For example, the *is_taxed_jointly* input of TCC in Fig. 5.2 is in fact the output of JT. If such relationships are not explicitly modeled, then the simulator cannot determine a correct order for executing the policy models. To address this problem, without having to require users to define an execution order for the policy models, we added a fourth input source, namely *policy*, to state that the value of a certain input to a policy model is determined through the execution of another policy model. This additional construct enables the simulator to automatically find an appropriate execution order for the policy model and warn the users if any cyclic relationships between the models are detected. Aside from this minor change to our policy modeling notation, we found the notation to be expressive enough for our case study.

The third and final aspect of RQ1 concerns whether we could find the statistical information necessary for running our data generator and instantiating our domain model. We consulted two information sources for statistics about the population of taxpayers in Luxembourg: (1) census data from STATEC (statec.lu/), and (2) a recent report published by Luxembourg's Ministry of Finance [Luxembourg's Ministry of Finance, 2015]. We extracted from these sources statistics about 15 quantities relevant to our simulation scenario. These statistics are described in Table 5.3.

Rows 1 to 10 in Table 5.3 are general statistics about the entire population, and Rows 11 to 15 are specialized statistics that replace some of the general ones under certain circumstances. For example, pension incomes require special treatment and are handled differently than, say, employment incomes. This is because pensions are subject to certain regulations that constrain their amounts to a predefined range. During simulation data generation, if a taxpayer has, for example, an employment income, then the general *Income amounts* statistic is used for (probabilistically) generating an income amount;

Table 5.3. Statistics from Public Census Data and Governmental Sources

	Statistic	Description
1	<i>Residence status</i>	Relative distribution of resident versus non-resident taxpayers (see Fig. 5.3).
2	<i>Age</i>	Distribution of the population by age (see Fig. 5.3).
3	<i>Household size</i>	Distribution of the size of households, e.g., 63% of households are composed by more than two persons.
4	<i>Types of civil union</i>	Relative distribution of civil unions, e.g., 95.6% of the unions are marriages.
5	<i>Income types</i>	Relative distribution of different income types, e.g., 1.4% of incomes are of type agriculture.
6	<i>Income amounts</i>	Distribution of gross annual income for households.
7	<i>Workers per household</i>	Distributions of households based on the number of active workers in the household.
8	<i>Divorce rate</i>	The percentage (by year) of individuals whose marital status has changed from married to divorced.
9	<i>Divorce types</i>	Relative distribution of divorce types, e.g., by mutual agreement.
10	<i>Widower rate</i>	Percentage of the population that has been widowed in a given year (past 10 years).
11	<i>Income for pensioners</i>	Specialization of income distribution for pensioners.
12	<i>Income for traders</i>	Specialization of income distribution for traders.
13	<i>Age of pensioners</i>	Specialization of age distribution for pensioners.
14	<i>Foreign income types</i>	Specialization of the distribution of income types for non-resident taxpayers.
15	<i>Residence status based on spouse's residence status</i>	Specialization of the distribution of resident versus non-resident taxpayers based on the residence status of their spouses.

whereas, if a taxpayer is receiving a pension, the specialized statistic for pensions, i.e., *Income for pensioners*, is used. All the statistics in Table 5.3 are provided using distributions represented as histograms.

There are six quantities for which we could not find suitable statistical information in the two sources mentioned earlier. These quantities are: (1) the probability that household members are living apart; (2) the probability that taxpayers are assisted by their spouses for realizing an income; (3) the distribution of taxpayers over countries of residence (noting that Luxembourg has a significant population of non-resident taxpayers); (4) the distribution of gross annual incomes for taxpayers (as opposed to those for households given by Row 6 of Table 5.3); (5) the detailed breakdown of the > €1 million gross annual income bracket; and (6) the distribution of child allowances over taxpayers within individual households.

We addressed these gaps in our statistical information using feedback from legal experts. Specifically, for quantities (1) through (4), we defined heuristics for value assignment. For example, for quantity (3), we set the country of residence to Luxembourg for resident taxpayers; for non-resident

taxpayers, we randomly picked one of the neighboring countries (France, Germany, or Belgium) as the country of residence.

With regard to quantity (5) and to avoid generating unrealistically large incomes, we put an upper bound on incomes larger than €1 million. Dealing with quantity (6) was slightly more involved. This is because taxpayers have some degree of control over who receives a child allowance within a household. This can affect the simulation results (see *is_receiving_allowance_for_children* input in Fig. 5.2). We tried to make quantity (6) as realistic as possible through optimization. In particular, we distributed child allowances over the taxpayers within the same household in a way that would minimize the household’s overall tax obligation.

In total, we attached 57 annotations (stereotypes) to our domain model for guiding the data generation process. About 70% of the annotations came from public sources. The remaining 30% were based on feedback from experts and common sense, e.g., the optimization for quantity (6) above. In summary, we could successfully compensate for the lack of access to real data in our case study. Whether the data generated based on our annotations leads to credible simulation results is discussed in RQ2.

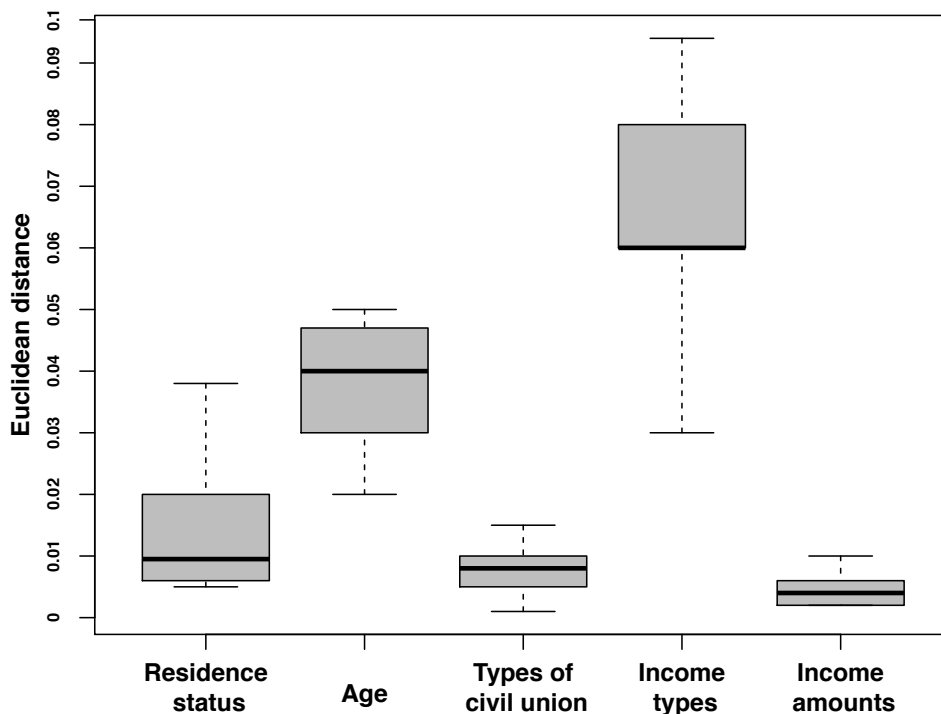


Figure 5.4. Euclidean Distances between (Selected) Distributions of the Actual Simulation Population and Distributions of the Generated Data Samples

5.3.2 RQ2: Are the simulation results credible?

We used the statistics discussed in RQ1 to guide our data generator. We ran the data generator 10 times, creating 10 samples, each containing 10,000 taxpayers. The execution time for a single run of

the data generator was ≈ 30 minutes. Once the samples were generated, we used the sanity checking procedure discussed in Section 5.2.3 to ensure that the samples were aligned with the known distributions for the statistics of Table 5.3. Specifically, for each statistic in this table, we calculated 10 Euclidean distances, one per generated sample, measuring how close that particular sample was to (the distribution of) the statistic. In Fig. 5.4, we provide box plots for the Euclidean distances calculated for selected statistics from Table 5.3. As shown by the figure, the maximum values in the box plots are < 0.1 , suggesting that all the generated samples were aligned with the actual simulation population. Similar results were observed for the other statistics of Table 5.3 not covered in Fig. 5.4.

After ensuring the quality of the generated data samples, we ran our simulator (automatically derived from our policy models) over the samples. The simulation took approximately 80 minutes to process each sample. As noted in earlier sections, we answer RQ2 by simulating the status quo and measuring how aligned the simulation results are with the current tax contribution statistics. The current statistics pertain to the most recent tax year that has been assessed (2014) [Luxembourg’s Ministry of Finance, 2015]. In more precise terms, we address RQ2 based on the contributions of households in different income brackets to the overall income tax revenue collected by the Government. In the simulation report that we prepared for our public service partners [Soltana et al., 2016b], we additionally considered the distribution of taxpayers over tax classes, the distribution of households by tax brackets, the evolution of income taxes by households, and the evolution of the overall revenue. Since we do not have any statistics for these dimensions to compare our simulation results against, we do not use them in RQ2.

Fig. 5.5 shows the results from the 10 runs of our simulator alongside the current statistics from 2014. The x -axis of the chart in Fig. 5.5 represents the income brackets; the y -axis represents the contributions of households to the revenue. For example, the chart indicates that, in 2014, households having a gross annual income between €50K and €60k contributed $\approx 7\%$ of the total tax revenue.

We observe from the chart of Fig. 5.5 that the simulation results are closely in line with the current statistics. To quantify how accurate our results are, we computed for each simulation run the Euclidean distance between the obtained curve and the current statistics (represented by the curve marked with large circles in Fig. 5.5).

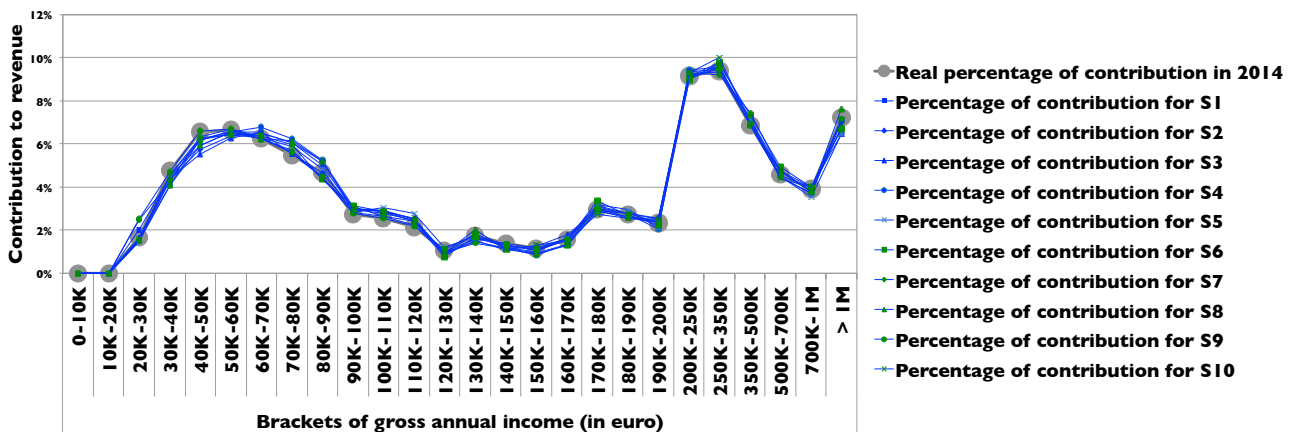


Figure 5.5. Simulated and Actual Contributions of Households to Revenue

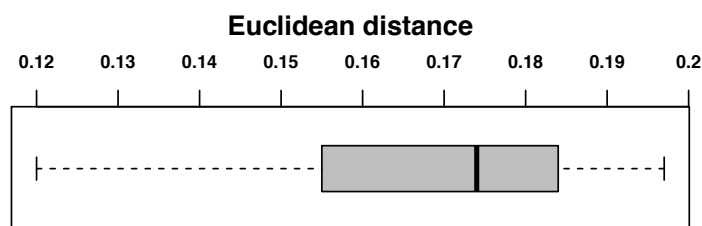


Figure 5.6. Euclidean Distance Box Plot for Simulation Results

In Fig. 5.6, we present a box plot for the Euclidean distances calculated over the simulation results. The figure shows that our simulation results deviate from the current statistics by a distance ranging from 0.12 to 0.197 with a median of 0.174. We believe that these results, given the restrictions we had in terms of access to real data, are very promising. The positive results from RQ2 provide confidence that we modeled the legal policies at the right level of abstraction for our analysis, and further that our data generator, while not a substitute for real data, can still produce meaningful data for simulation.

5.3.3 Threats to Validity

The most relevant aspects of validity for our case study are internal and external validity, as we discuss below.

Internal validity. The statistical distributions that we used for data generation pertain to different taxation years. This raises the potential problem that some of the distributions may no longer be valid for the taxation year addressed in our case study (2014). To mitigate this problem, we used statistics only from the last ten years, our assumption being that the properties of the taxpayers population would not change drastically over a short span of time. The close alignment seen in the chart of Fig. 5.5 between our simulation results and the actual contributions of households to the revenue for the addressed taxation year suggests that using old distributions (from the past ten years) has not had a major negative impact on accuracy.

External validity. To date, we have applied our simulation approach in only one legal domain, i.e., taxation. Further case studies in other legal domains are thus essential for improving external validity. Our modeling notation is geared towards laws that provide step-by-step instructions for performing legal administrative processes and achieving compliance. This makes us optimistic that our approach will generalize with ease to laws other than taxation that have the above characteristic, e.g., social security laws. Whether our approach can be extended to a broader spectrum of laws, e.g., those having a more declarative nature and relying heavily on deontic modalities such as permissions and obligations, needs a thorough investigation.

5.4 Lessons Learned

In this section, we reflect on the lessons that we learned from the case study described in Sections 5.2 and 5.3.

Align the models with the reasoning of legal experts. For a given policy, one can often come up with alternative but semantically-equivalent model representations. To illustrate, consider the policy model of Fig. 5.2. Here, one can merge the decisions concerning the age ($taxpayer_age \geq 64$) and the allowances ($is_receiving_allowance_for_children$) using a logical disjunction. Alternatively, one may swap the order in which these two decisions occur. In either case, we obtain a model that is semantically-equivalent to that in Fig. 5.2. An observation that we made during policy model validation is that legal experts usually have a preconceived mental image of the workflow of a policy. When a policy model deviated from this mental image, the experts proposed modifications to avoid logical formulas that were unnatural to them and to bring the workflow of the models in line with the way they reasoned about the policies. To improve communication with legal experts and increase understandability, it is therefore important not to view a policy model merely as a logical formula. Special attention needs to be paid to how the model is represented and ensuring that the representation is aligned with how legal experts think.

Maintain traceability to legal texts. Another observation from model validation is that, during the validation process, the experts frequently needed to consult the legal provisions underlying the policies. Being able to do so effectively requires traceability between the policy models and the relevant legal texts. In particular, the policy models and their inputs need to be traced to the specific legal provisions pertaining to them, e.g., articles in legislative texts. For example, the policy model of Fig. 5.2 is linked to articles 119 and 157bis of *LITL*, as shown on the top of the model.

A further remark about traceability concerns the relationship between the changes made to legal texts and the updates made to the policy models in response to these changes. (Proposed) changes to legal texts are typically provided as amendments. During our validation, we observed that legal experts could not readily follow how a change stated in an amendment related to the updates made to the policy models in order to materialize that change. This prompted us to develop the simple guidelines shown in Table 5.4 to clarify how different types of amendments can impact the policy models. Subsequently, when we modified a policy model in response to an amendment, we explained to the legal experts the *rationale* behind the modifications by tracing the amendment to the affected model elements (including deleted ones). We received positive feedback from the experts about our clarifications and how we traced the amendments to model changes.

Keep the modeling notation simple and lean. Once provided with adequate training, legal experts were able to grasp with relative ease class and activity diagrams. In contrast, OCL, which as noted in Section 5.1.1 is used for expressing the inputs to our policy models, posed a challenge to the legal experts, despite (basic) training and our attempts to explain the meaning of the OCL expressions. While we did not try to use any other formal language as a replacement for OCL and examine whether understandability would improve, we anticipate that using formal languages in general causes a communication barrier in the context of legal requirements. To validate our policy models with legal experts, we removed from these models all OCL expressions. To compensate, we wrote precise natural-language descriptions for the model inputs and used these descriptions as the basis for valida-

Table 5.4. Impact of Amendments on Policy Models

Amendment type	Description	Effects on policy models
<i>Addition</i>	New provisions, for example, new articles or paragraphs, are being added to the law.	Based on the nature of the new provisions, either new policy models are created or the existing ones are extended.
<i>Deletion</i>	Some existing provisions are being repealed and removed from the law.	Some existing policy models may be removed in their entirety or in part. No new policy specification is normally needed in response to a deletion amendment.
<i>Replacement</i>	Some phrases in existing provisions are being replaced.	The effect is that of combining additions and deletions described above. The effect of replacements is often limited to making changes in the policy models.
<i>Redesignation</i>	The numbers or titles of some existing provision are being changed.	No changes are required to the inputs or the workflows of the existing policy models. The traceability information that links the models to the text of law needs to be updated to account for the new numbers or titles.

tion. The policy model of Fig. 5.2 alongside the glossary of Table 5.1 provide an actual example of the artifacts we used during validation.

A further observation is that although the legal experts in our case study could understand and review class and activity diagrams, they could not be expected to keep the notational details in their working memory for long. In the early validation rounds, we would start a session with a brief “refresher” on the modeling notation, e.g., reminding the experts that the diamond shapes represent decisions, the «update» actions store simulation results, and so on. In later rounds, we prepared a *notation legend* and added it at the bottom of the models being validated. We found this strategy to be very effective for facilitating communication with legal experts. An important consideration with regard to legends is for them to be as succinct as possible, implying that there is an advantage in restricting the notational elements being used to what is absolutely essential for expressing the models.

5.5 Related Work

Model-based techniques are increasingly being used for expressing legal requirements. For example, van Engers et al. [van Engers et al., 2001] use UML for modeling the Dutch tax legislation. Ghanavati et al. [Ghanavati et al., 2014, Ghanavati et al., 2007] employ a combination of goal models and

use cases for capturing legal requirements and supporting the comparison of multiple regulations. Ingolfo et al. [Ingolfo et al., 2014, Ingolfo et al., 2013] develop a goal-oriented modeling framework for arguing about regulatory compliance. Breaux and Powers [Breaux and Powers, 2009] specify legal requirements via business process models and use these models for compliance checking. Zeni et al. [Zeni et al., 2015] and Breaux [Breaux, 2009a] develop conceptual models for characterizing key abstractions in legal texts, and exploit these models for ambiguity reduction and various types of automation. These earlier work strands are not specifically targeted at simulation. Our work complements the above strands by proposing an approach for modeling legal requirements as executable policies and simulating these policies.

Several tools exist in the field of applied economics for policy simulation, e.g., SYSIFF [Canova et al., 2009], POLIMOD [Sutherland, 1995], and EUROMOD [Figari et al., 2015]. These tools use a combination of spreadsheets and software code for specifying legal policies. This strategy can complicate the validation of the resulting policy specifications, as legal experts often lack the software engineering expertise required to understand complex spreadsheets and software code. In contrast, our approach uses models to raise the level of abstraction at which legal policies are specified. This helps improve the understandability of policy specifications by legal experts. Furthermore, the above tools assume that the input data for simulation is available a priori. This assumption does not always hold as noted at the beginning of this chapter. Our approach is equipped with a built-in data generator that can produce artificial but representative input data for simulation when real data is missing or incomplete.

5.6 Conclusion

In this chapter, we reported on a real-world case study of our model-based approach for simulating legal policies. Through our case study, we evaluated the feasibility and usefulness of our approach in practice to assess changes to policies, demonstrating that the approach can be applied with reasonable effort, and that it yields credible results. We further discussed the lessons we learned from the case study, particularly in relation to making modeling more palatable to legal experts. An important characteristic of our simulation approach is that it builds on the same models that are used for legal compliance analysis. This makes the experience gained from our case study relevant not only to simulation but also to legal compliance.

In the future, we plan to use our simulation approach to conduct case studies beyond the taxation domain. For example, we would like to apply the approach to social security and customs laws where simulation is commonly used for assessing the risks and consequences of legal reforms. We would further like to extend our approach to account for situations where the simulation input data needs to be dynamically updated, e.g., when taxpayers change their behavior in response to changes in the laws and regulations.

Chapter 6

Synthetic Data Generation for Statistical Testing

Usage-based statistical testing, or statistical testing for short, is concerned with detecting faults that cause the most frequent failures (thus affecting reliability the most), and with estimating reliability via statistical models [Runeson and Wohlin, 1995]. In contrast to testing techniques that focus on system verification (fault detection), e.g., testing driven by code coverage, statistical testing focuses on system validation from the perspective of users. Statistical testing typically requires a *usage profile* of the system under test. This profile characterizes, often through a probabilistic formalism, the population of the system’s usage scenarios [Musa, 1993].

Existing work on usage profiles has focused on state- and event-based systems, with the majority of the work being based on Markov chains [Whittaker and Poore, 1993, Poore and Trammell, 1999, Kallepalli and Tian, 2001, Tonella and Ricca, 2004, Guen et al., 2004, Herbold et al., 2017]. For many systems, which we refer to as *data-centric* and concentrate on in this chapter, system behaviors are driven primarily by data, rather than being triggered by stimuli. For example, consider a public administration system that calculates social benefits for citizens. How such a system behaves is determined mainly by complex and interdependent data such as citizens’ employment, household makeup, residence status, disabilities, and so on. The system’s scenarios of use are thus intimately related to the data that is processed by the system. Consequently, the usage profile of such a system is governed by the statistical characteristics of the system’s input data, or stated otherwise, by the system’s *data profile*. Given our focus on data-centric systems and the explanation above, we equate, “usage profile” and “data profile”, and use the latter term hereafter.

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

Further, and perhaps more importantly, synthetic data (and hence a data profile) are indispensable when access to actual data is restricted. Notably, under most privacy regulations, e.g., EU’s General Data Protection Regulation [General Data Protection Regulation, 2016], “repurposing” of personal data is prohibited without explicit consent. This complicates sharing of any actual personal data with third-parties who are responsible for software development and testing. Anonymization offers a partial solution to this problem; however, doing so often comes at the cost of reduced data quality [De Capitani di Vimercati et al., 2010] and proneness to deanonymization attacks [Al-Azizy et al., 2015].

Data profiles have received little attention in the literature on software testing. This is despite the fact that many data-centric systems, e.g., public administration and financial systems, are subject to reliability requirements and thus statistical testing. We have already presented in Chapter 4 a statistical data profile and a heuristic algorithm for generating representative synthetic data. Although motivated by microeconomic simulation [Figari et al., 2015] rather than software testing, our previous data generation approach provides a useful basis for generating data that can be used for statistical testing. However, the approach suffers from an important limitation: while the approach generates synthetic data that is aligned with a desired set of statistical distributions and has shown to be good enough for running financial simulations as demonstrated in Chapter 5, the approach cannot guarantee the satisfaction of *logical constraints* that need to be enforced over the generated data.

To illustrate, we note three among several other logical anomalies that we observed when using our previous approach (presented in Chapter 4) for generating test cases based on a data profile of citizens’ records: children who were older than their parents, individuals who were married before being born, and individuals who were classified as widower without ever having been married. Without the ability to enforce logical constraints to avoid such anomalies, the generated data is unsuitable for statistical testing and estimating reliability. This is because such anomalies may result in exceptions or system behaviors that are not meaningful. In either case, targeted system behaviors will not be exercised.

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

Contributions. The contributions presented in this chapter are as follows:

1) We develop a model-based test data generator that can simultaneously satisfy statistical representativeness and logical validity requirements over complex, interdependent data. The desired statistical characteristics are expressed using our previously-developed probabilistic UML annotations (presented in Section 3.5). Validity constraints are expressed using UML’s constraint language, OCL [Object Management Group, 2004]. Our data generator incorporates two collaborating components: (a) a search-based OCL constraint solver which enhances previous work by Ali et al. [Ali et al., 2016],

and (b) a mechanism that guides the solver toward satisfying the statistical characteristics that the generated data (test suite) must exhibit.

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

Structure. Section 6.1 discusses background. Section 6.2 outlines our overall approach. Section 6.3 elaborates our data generator. Section 6.4 presents our evaluation. Section 6.5 compares with related work. Section 6.6 concludes the chapter.

6.1 Background

To make this chapter self-contained, we briefly describe the existing components (from Chapter 4) that we re-use in this chapter. Specifically, we re-use (1) our profile for expressing the desired statistical characteristics of data (Section 4.4), and (2) our previous data generator (Section 4.5) which we process further to achieve not only representativeness but logical validity as well. Readers acquainted with our previous technical contributions might wish to skip this chapter.

For specifying statistical characteristics, we defined a set of annotations (stereotypes) which can be attached to a data schema expressed as a UML Class Diagram. Fig. 6.1 illustrates some of these annotations on a small excerpt of a data schema for a tax administration system.

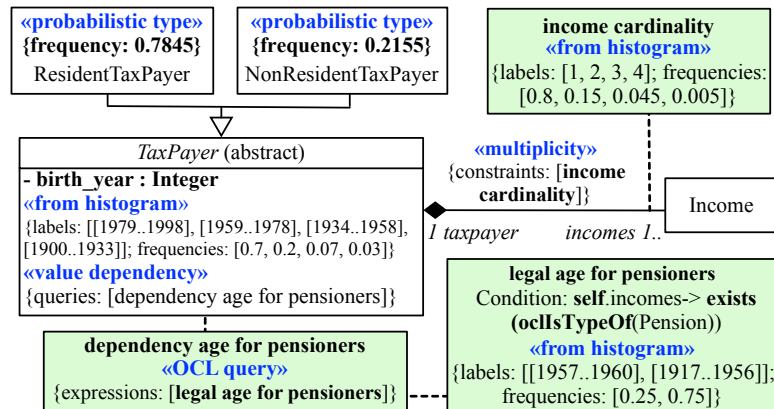


Figure 6.1. Data Schema (Excerpt) Annotated with Statistical Characteristics

The «probabilistic type» stereotypes applied to the specializations of the *TaxPayer* class state that $\approx 78\%$ of the taxpayers should be resident and the remainder should be non-resident.

The «from histogram» stereotype attached to the *birth_year* attribute provides, via a histogram, the birth year distribution for taxpayers. The attribute *birth_year* is further annotated with a conditional probability specified via the «value dependency» stereotype. The details of this conditional probability are provided by the *legal age for pensioners* constraint. The information in the constraint

reads as follows: 25% of pensioners have their birth year between 1957 and 1960, i.e., are between 57 and 60 years old; the remaining 75% are older than 60.

The «multiplicity» stereotype attached to the association between *TaxPayer* and *Income* classes describes, via the *income cardinality* histogram, the distribution of the number of incomes per taxpayer. As shown in Fig. 6.1, 80% of the taxpayers have one income, 15% have two, and so on.

For generating a data sample, we previously presented in Section 4.5 a heuristic technique that is aimed exclusively at representativeness. This technique traverses the elements of the data schema and instantiates them according to the prescribed probabilities. The technique attempts to satisfy multiplicity constraints but satisfaction is not guaranteed. More complex logical constraints are not supported.

In this chapter, we use as a starting point the data generated by our previous data generator, and alter this data to make it valid without compromising representativeness. Indeed, as we show in Section 6.4, our new approach not only results in logically valid data but also outperforms our previous approach in terms of representativeness.

6.2 Approach Overview

Fig. 6.2 presents an overview of our approach for generating representative and valid test data. Steps 1–3 are manual and Step 4 is automatic. In Step 1, *Define data schema*, we define using a UML Class Diagram (CD) [Object Management Group, 2015] the schema of the data to generate. This diagram, illustrated earlier in Fig. 6.1, is the basis for: (a) capturing the desired statistical characteristics of data (Step 2), and (b) generating synthetic data (Step 4).

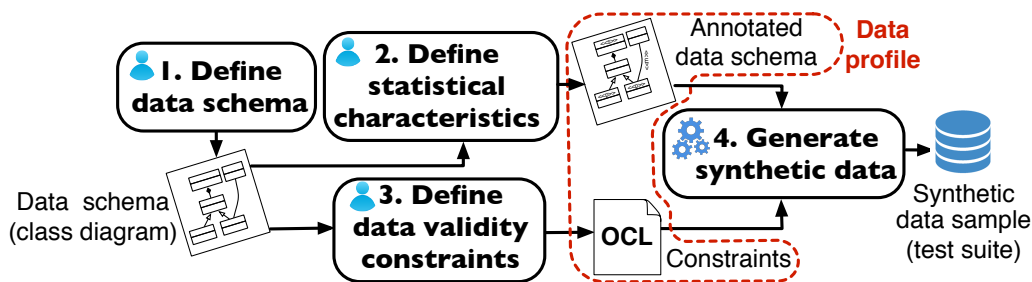


Figure 6.2. Approach for Generating Valid and Representative Synthetic Data

In Step 2, *Define statistical characteristics*, the CD from Step 1 is enriched with probabilistic annotations (see Section 6.1) to express the representativeness requirements that should be met during data generation in Step 4.

In Step 3, *Define data validity constraints*, users express via the Object Constraint Language (OCL) [Object Management Group, 2004] the logical constraints that the generated data must satisfy. For example, the following OCL constraint states that children must be born at least 16 years after their parents: `self.children->forAll(c| c.birth_year > self.birth_year + 16)`. Here, `self` refers to a person.

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

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

In the next section, we elaborate Step 4, which is the main technical contribution covered in this chapter.

6.3 Generating Synthetic Data

In this section, we describe our synthetic data generator. Fig. 6.3 shows the strategy employed by the data generator. Initially and as mentioned in Section 6.1, a potentially invalid collection of instance models is created using our previous heuristic data generator (see Section 4.5). We refer to this initial collection as the *seed sample*. Our data generator then transforms the seed sample into a collection of valid instance models. This is achieved using a customized OCL constraint solver, presented in Section 6.3.1.

The solver attempts to repair the invalid instance models in the seed sample. To do so, the solver considers the constraint specified in Step 3 of our overall approach (Fig. 6.2) alongside the multiplicity constraints of the underlying data schema and the constraints implied by the probabilistic annotations from Step 2 of the approach. The rationale for feeding the solver with instance models from the seed sample, rather than having the solver build instance models from scratch, is based on the following

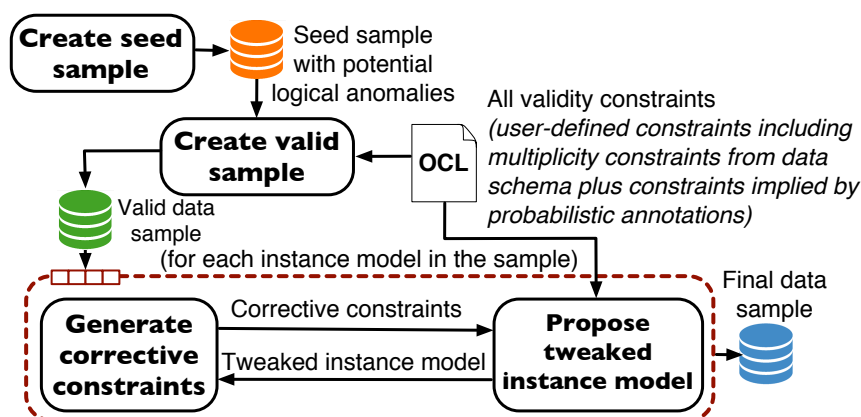


Figure 6.3. Overview of our Data Generation Strategy

intuitions: (1) By starting from the seed sample, the solver is more likely to be able to reach valid instance models, and (2) The valid sample built by the solver will not end up too far away from being representative, in turn making it easier to fix deviations from representativeness, as we discuss later.

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

Once we have a valid data sample that has the requested number of instance models in it, our data generator attempts to realign the sample back with the desired statistical characteristics. This is done through an iterative process, delineated in Fig. 6.3 with a dashed boundary. We elaborate the details of this iterative process in Section 6.3.2.

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

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

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

In the above scenario, we illustrated the iterative process using a single corrective constraint. In practice, the process may generate multiple such constraints, since the data sample at hand may be

deviating from multiple representativeness requirements. We treat corrective constraints as being *soft*. This means that if after the maximum number of iterations, the solver manages to solve some of the corrective constraints but not all, the process will give the tweaked instance model a chance to replace the original one *as long as* the tweaked instance model still satisfies all the validity constraints.

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

6.3.1 Solving OCL Constraints

A number of techniques exist for solving OCL constraints, notably using Alloy [Anastasakis et al., 2010, Cunha et al., 2015], constraint programming [Cabot et al., 2014], and (metaheuristic) search [Ali et al., 2013, Ali et al., 2016]. Alloy often fails to solve constraints that involve large numbers [Krieger and Knapp, 2008]. We observed via experience that this limitation could be detrimental in our context. For example, our case study in Section 6.4 has several constrained quantities, e.g., incomes and allowances, that are large numbers. As for constraint programming, to our knowledge, the only publicly-available tool is UML2CSP [Cabot et al., 2014]. We observed that this tool did not scale for our purposes. In particular, given a time budget of 2 hours, UML2CSP did not produce any valid instance model in our case study. This is not practical for statistical testing where we need a representative sample with many (*hundreds* or more) valid instance models.

Search, as we demonstrate in Section 6.4, is more promising in our context. Although search-based techniques cannot prove (un)satisfiability, they are efficient at exploring large search spaces. In our work, we adopt with two customizations the search-based OCL solver of Ali et al.'s [Ali et al., 2016], hereafter referred to as the *baseline solver*. The customizations are: (1) a feature for setting a specific instance model as the starting point for search, and (2) a strategy to avoid premature narrowing of the search space. The former customization, which is straightforward and not discussed here, is necessary for realizing the process in Fig. 6.3. The latter customization is discussed next.

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

The above heuristic in the baseline solver poses no problem as long as the goal is to find *some* valid solution. If search fails from one starting point, the solver (pseudo-)randomly picks another and starts over. In our context however, starting over from an arbitrary point is not an option. For the final data sample to have a chance of being aligned with the desired statistical characteristics, the

solver needs to use as starting point instance models from a statistically representative seed sample (see Fig. 6.2). If the solver fails at making valid an instance model from the seed sample, that instance model has to be discarded. This negatively affects performance, since the solver will need to start all over on a replacement instance model supplied by the seed data generator, as noted earlier.

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

To address the above problem, we customize the baseline solver as follows: Rather than working directly on the original constraints, the customized solver works on the constraints' *prime implicants* (PI). An implicant is prime (minimal) if violating any of its literals results in the violation of the underlying constraint. To derive all the PIs for a given OCL constraint, we first transform the constraint into a logical expression with only ANDs and ORs, negation, and OCL operations. We next transform this expression into Disjunctive Normal Form (DNF) by applying De Morgan's law [Hurley, 2014]. Each clause of the DNF expression is a PI. For instance, constraint $C1$ yields three PIs: $(x=2 \text{ and } y=5)$, $(x <> 2 \text{ and } x=3 \text{ and } y=4)$, and $(x <> 2 \text{ and } x <> 3 \text{ and } y=0)$. Note that we use the term PI slightly differently than what is standard in logic. Our literals are not necessarily independent logically. For example, in the second PI above, $x <> 2$ is redundant because $x=3$ implies $x <> 2$. Such redundancies pose no problem and are ignored.

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

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

Second, the DNF transformation can result in exponentially large DNF representations when there are many literals [Miltersen et al., 2005]. Such exponential explosion is unlikely to arise in our

context: Manually-written logical constraints for data models typically include only a handful of literals. For the corrective constraints that are generated automatically (through Alg. 6 described later), the number of literals is at most as many as the number of ranges (or categories) in the bar graphs that capture the desired statistical distributions. Again, these numbers are seldom very large. In our case study of Section 6.4, the DNF transformations took negligible time (milliseconds).

To summarize, using PIs instead of the original constraints helps avoid dead-ends when solution search has to start from a specific point in the search space. In Section 6.4 (RQ1), we examine how customizing the baseline solver in the manner described in this section influences performance.

6.3.2 Generating Valid and Representative Data

This section presents the technical details of our data generation strategy, depicted in Fig. 6.3 and outlined earlier on. We already discussed the creation of the seed sample (in Section 4.5) and how we make this sample valid using a customized OCL solver (Section 6.3.1). Below, we focus on the iterative process in Fig. 6.3, i.e., the region delineated by dashed lines, and present the algorithms behind this process. A reader interested primarily in the practical utility of our work may wish to move on directly to Section 6.4.

We start with some remarks about how we represent statistical distributions. The instruments we use to this end are *barcharts* (for categorical quantities) and *histograms* (for ordinal and interval quantities). Without loss of generality, and while we support both notions, we talk exclusively about histograms in the text. A histogram is a set of *bins*. Each bin is defined by a label (value or range), and a *relative frequency* denoting the proportional abundance of the bin’s label. We do not directly handle continuous distributions, e.g., the normal distribution. Continuous distributions are discretized into histograms. Doing so is routine [Hammond and Bickel, 2015] and not explained here. We note however that the discretization should not be too fine-grained, e.g., resulting in more than 100 bins. This is because the corrective constraints in our approach will get complex, in turn posing scalability issues for the OCL solver, e.g., with respect to the DNF transformation discussed in Section 6.3.1.

As a convention, we use calligraphic math symbols, e.g., \mathcal{S} , to represent sets and maps in the remainder of this section.

The PIM algorithm. Alg. 5, *Process Instance Model (PIM)*, presents the procedure for one iteration of the iterative process (region within the dashed boundary) in Fig. 6.3. PIM takes the following as input: (1) a valid data sample, (2) a specific instance model from the sample to process, (3) a set of validity constraints, (4) the desired statistical characteristics defined as histograms, (5) a parameter specifying how many attempts the algorithm should make to generate tweaked instance models, and (6) a parameter specifying how sensitive the algorithm is to differences in relative frequencies. Essentially, if the difference between two relative frequencies is below the sensitivity parameter, the two frequencies are considered equal.

Alg. 5: Process Instance Model (PIM)

Inputs : (1) a set \mathcal{S} of valid instance models; (2) an instance model $\text{inst} \in \mathcal{S}$ to process; (3) a set \mathcal{V} of validity constraints; (4) a set $\mathcal{H}_{\text{desired}}$ of desired statistical characteristics (expressed as histograms); (5) a parameter nb_attempts denoting the number of times that the solver will be invoked over inst to create tweaked instance models; (6) a parameter $\text{freq_sensitivity} \in [0..1]$ denoting the margin beyond which two relative frequencies are deemed far apart. */* freq_sensitivity is used only for invoking GCC (Alg. 6). */*

Output : Either inst or a tweaked instance model, whichever leads to a more representative data sample.

Fun. calls: **GCC**: generates corrective constraints (Alg. 6);
solve: invokes the customized solver (see Section 6.3.1).

```

1  $CC \leftarrow \text{GCC}(\mathcal{S}, \text{inst}, \mathcal{H}_{\text{desired}}, \text{freq\_sensitivity})$ 
  /* CC is the set of corrective constraints returned by Alg. 6. */
2 if ( $CC = \emptyset$ ) then
3   return  $\text{inst}$ 
4  $\mathcal{T} \leftarrow \emptyset$  /* T will store potential replacements for inst. */
5  $i \leftarrow 0$  /* i is the number of times the solver has been invoked so far. */
6 while ( $i < \text{nb\_attempts}$ ) do
7    $\text{inst\_tweaked} \leftarrow \text{solve}(\text{inst}, \mathcal{V} \cup CC)$ 
8    $i \leftarrow i + 1$ 
9   if ( $\text{inst\_tweaked}$  satisfies the constraints in  $\mathcal{V}$ ) then
10     $\mathcal{T} \leftarrow \mathcal{T} \cup \{\text{inst\_tweaked}\}$ 
11  $\text{inst\_best} \leftarrow \text{inst}$ 
12  $\mathcal{S}_{\text{best}} \leftarrow \mathcal{S}$ 
13 foreach  $\text{candidate} \in \mathcal{T}$  do
14    $\mathcal{S}' \leftarrow (\mathcal{S} \setminus \{\text{inst}\}) \cup \{\text{candidate}\}$ 
15   if ( $\mathcal{S}'$  is better aligned with  $\mathcal{H}_{\text{desired}}$  than  $\mathcal{S}_{\text{best}}$ ) then
16      $\text{inst\_best} \leftarrow \text{candidate}$ 
17      $\mathcal{S}_{\text{best}} \leftarrow \mathcal{S}'$ 
18 return  $\text{inst\_best}$ 

```

The algorithm works in three stages as we describe next.

1) *Generate corrective constraints (L. 1-3 of Alg. 5):* In this stage, PIM calls another algorithm GCC (Alg. 6, described later). GCC generates corrective constraints for the instance model being processed (L. 1). For example, assume that the instance model is a pensioner, and that pensioners are currently over-represented in the data sample. In response, GCC will generate the following corrective constraint, named *CCI*: **self.incomes->forAll(not oclIsTypeOf(Pension))**. If GCC does not yield any corrective constraints, then the original instance model will be retained in the sample (L. 2-3).

2) *Build tweaked instance models (L. 4-10 of Alg. 5):* In this stage, PIM attempts to produce a set of tweaked instance models based on the corrective constraints generated previously. These constraints are fed to the solver alongside the validity constraints (L. 7). To illustrate, consider the example corrective constraint *CCI* generated at the first stage. This constraint instructs the solver to tweak the instance model at hand so that the income type will no longer be pension. PIM tries building tweaked instance models multiple times (L. 6). This is intended at coming up with multiple candidates (ideally more than one) for replacing the original instance model in the sample. As noted earlier, we treat corrective constraints as soft and try to satisfy them on a best-effort basis. Therefore, any tweaked instance model returned by the solver will be included in the set of candidate replacements as long as the validity constraints hold (L. 9-10).

3) *Select best replacement (L. 11-18 of Alg. 5):* In this stage, PIM chooses to either retain the original instance model or replace it with one of the tweaked instance models computed in the second stage.

The criterion applied for the decision is which instance model, once incorporated into the sample, will result in the most statistically representative sample.

The metric we use for measuring statistical representativeness is *Euclidean distance* [Cha, 2007]. This metric measures how far two histograms are from one another. The closer the distance between two histograms is to zero, the better aligned the histograms are. For example, suppose that the data sample is composed of 40% resident versus 60% non-resident taxpayers. As showed in the data schema excerpt in Fig. 6.1, the desired distribution is $\approx 78\%$ resident versus $\approx 22\%$ non-resident. The Euclidean distance between the data sample and the desired distribution is ≈ 0.55 , indicating that the sample is not representative. Since PIM needs to take into account several distributions simultaneously, it uses the average of the Euclidean distances computed for all the histograms.

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

The GCC algorithm. Given an instance model *inst* within a data sample \mathcal{S} , Alg 2., titled *Generate Corrective Constraints (GCC)*, provides suggestions (in the form of constraints) as to how *inst* can be tweaked so that \mathcal{S} will become a more representative sample. The input to GCC was described previously as part of PIM’s input. GCC works in three stages as explained below. Throughout the explanation, we will be referring to Table 6.1, Table 6.2 and Fig. 6.4 for illustration.

1) *Generate OCL literals (L. 1-18 of Alg. 6):* In this stage, GCC groups histograms that annotate the same data schema element, i.e., class, attribute or association, as illustrated in Fig. 6.1 (L. 4-11). For each group, sets \mathcal{O} and \mathcal{U} will be built (L. 12-18). These two sets contain OCL literals for *Over-represented* and *Under-represented* histogram bins, respectively. These literals will later be assembled into intermediate constraints (see second stage below).

The literals in \mathcal{O} and \mathcal{U} are derived as follows: We compare in a pairwise manner the relative frequencies of the actual characteristics of \mathcal{S} against the desired characteristics in $\mathcal{H}_{\text{desired}}$ (L. 13-15). To illustrate, consider rows 2 and 3 of Table 6.1. The relative frequencies to compare are $F1$ against $D1$, $F2$ against $D2$, and so on. If for an index i , $|Fi - Di| > \text{freq_sensitivity}$ (L. 15), the algorithm will generate a literal. Whether an exclusion or inclusion literal is generated depends on whether the underlying bin is over- or under-represented (L. 16-18). For example, in Table 6.1, the difference between $F1$ and $D1$ is $|0.9 - 0.7| = 0.2$, which is larger than the (user-provided) *freq_sensitivity* value on row 4 of Table 6.1. Since $F1$ is over-represented, the following literal is added to \mathcal{O} in order to exclude $L1$: `TaxPayer.allInstances()->select(id = 1)->forall(not (birth_year >= 1979 and birth_year <= 1998))`. Note that the generated literal targets the specific instance model being processed, since ultimately, the literal is intended at tweaking that particular instance model. The case for under-representation is dual and not illustrated.

2) *Combine literals (L. 19-21 of Alg. 6):* In the second stage, the algorithm combines the literals in \mathcal{O} and \mathcal{U} into what we call an *intermediate* constraint. We use the term “intermediate” to distinguish

Alg. 6: Generate Corrective Constraints (GCC)

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

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

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

```

1  $\mathcal{CC} \leftarrow \emptyset$ 
2  $\mathcal{H}_{\text{current}} \leftarrow$  Statistical characteristics of  $\mathcal{S}$ 
3  $\mathcal{M} \leftarrow \{H \in \mathcal{H}_{\text{current}} \mid H \mapsto ""\}$  /*  $\mathcal{M}$  maps each histogram in  $\mathcal{H}_{\text{current}}$  onto an ‘‘intermediate’’ constraint (explained in the text).
   All histograms are initially mapped onto an empty expression.
4  $\mathcal{P} \leftarrow \emptyset$  /*  $\mathcal{P}$  will store histograms (from  $\mathcal{H}_{\text{current}}$ ) which have been already processed. */
5 foreach  $H \in \mathcal{H}_{\text{current}}$  do
6   if ( $H \in \mathcal{P}$ ) then
7     continue /* We have already processed  $H$  and thus skip the loop. */
8   Let  $e$  be the data schema element to which  $H$  has been attached
9   Let  $\mathcal{L}$  be the set of all histograms in  $\mathcal{H}_{\text{current}}$  that annotate  $e$ 
10  foreach  $L \in \mathcal{L}$  do
11     $\mathcal{P} \leftarrow \mathcal{P} \cup \{L\}$  /* Histogram  $L$  is marked as processed. */
12    Let  $\mathcal{O}$  and  $\mathcal{U}$  be initially empty sets of OCL literals
13    /*  $\mathcal{U}$  stores literals generated for Under-represented bins;
14     *  $\mathcal{O}$  stores literals generated for Over-represented bins. */
15    foreach relative frequency  $F \in L$  do
16      Let  $D$  be the relative frequency in  $\mathcal{H}_{\text{desired}}$  corresponding to  $F$ 
17      if ( $|F - D| > \text{freq\_sensitivity}$ ) then
18        if ( $F > D$ ) then
19           $\mathcal{O} \leftarrow \mathcal{O} \cup \{\text{excludeBin}(\text{inst}, F)\};$ 
20        else  $\mathcal{U} \leftarrow \mathcal{U} \cup \{\text{includeBin}(\text{inst}, F)\};$ 
21      if ( $\mathcal{O} \neq \emptyset$  or  $\mathcal{U} \neq \emptyset$ ) then
22         $\text{OCL}_{\text{intermediate}} \leftarrow \left( \bigwedge_{j=1}^{j=|\mathcal{O}|} \mathcal{O}_j \wedge \bigvee_{j=1}^{j=|\mathcal{U}|} \mathcal{U}_j \right)$  /* See Fig. 6.4. */
23         $\mathcal{M} \leftarrow \mathcal{M} \cup \{L \mapsto \text{OCL}_{\text{intermediate}}\}$ 
24       $\mathcal{A} \leftarrow \{A \in \mathcal{M} \mid \mathcal{M}(A) \neq ""\}$  /*  $\mathcal{A}$  is the set of all histograms in  $\mathcal{M}$  with a non-empty intermediate constraint */
25      if ( $\mathcal{A} \neq \emptyset$ ) then
26        if ( $|\mathcal{A}| = 1$  and  $\mathcal{M}$ (single histogram in  $\mathcal{A}$ ) is unconditional) then
27           $\text{OCL}_{\text{final}} \leftarrow \mathcal{M}$ (single histogram in  $\mathcal{A}$ ) /* Row 1 of Table 6.2 */
28        else
29           $\text{OCL}_{\text{else}} \leftarrow \text{‘‘true’’}$  /*  $\text{OCL}_{\text{else}}$  will store the ‘‘catch all’’ else rule when all of  $\mathcal{A}$ ’s histograms are conditional (Row 3 of
30           Table 6.2) */
31          foreach  $A \in \mathcal{A}$  do
32             $\text{condition}_A \leftarrow \text{‘‘true’’}$  /*  $\text{condition}_A$  will store the OCL condition for histogram  $A$ ’s intermediate constraint. */
33            if ( $A$  is conditional) then
34               $\text{condition}_A \leftarrow$  condition of  $A$ 
35               $\text{OCL}_{\text{else}} \leftarrow \text{OCL}_{\text{else}} \wedge (\neg \text{condition}_A)$ 
36              foreach  $B \in (\mathcal{A} \setminus \{A\})$  do
37                /* Now, complete  $A$ ’s condition based on other histograms in  $\mathcal{A}$ . */
38                if ( $B$  is conditional) then
39                   $\text{condition}_A \leftarrow \text{condition}_A \wedge (\neg \text{condition}_B)$ 
40               $\text{OCL}_{\text{final}} \leftarrow \text{OCL}_{\text{final}} \vee (\text{condition}_A \wedge \mathcal{M}(A))$ 
41            if (all histograms in  $\mathcal{A}$  are conditional) then
42               $\text{OCL}_{\text{final}} \leftarrow \text{OCL}_{\text{final}} \vee \text{OCL}_{\text{else}}$ 
43           $\mathcal{CC} \leftarrow \mathcal{CC} \cup \{\text{OCL}_{\text{final}}\}$  /* Store  $\text{OCL}_{\text{final}}$  in  $\mathcal{CC}$ . */
44       $\mathcal{M} \leftarrow \{H \in \mathcal{H}_{\text{current}} \mid H \mapsto ""\}$  /* Reset  $\mathcal{M}$ . */
45 return  $\mathcal{CC}$ 

```

Table 6.1. Illustrative Example for Alg. 6

	Construct	Value
1	Excerpt of the instance model to process.	
2	Desired statistical characteristics ($\mathcal{H}_{\text{desired}}$): For simplicity, we limit our illustration to the histograms attached to the <i>birth_year</i> attribute of <i>TaxPayer</i> in Fig. 6.1.	The first histogram, $H1$, attached to <i>birth_year</i> : - Bin labels: $\{L1=[1979..1998], L2=[1959..1978], L3=[1934..1958], L4=[1900..1933]\}$ - Relative Frequencies: $\{D1=0.7, D2=0.2, D3=0.07, D4=0.03\}$ - Condition: true (none)
		The second histogram, $H2$, attached to <i>birth_year</i> : - Bin labels: $\{L5=[1957..1960], L6=[1917..1956]\}$ - Relative Frequencies: $\{D5=0.25, D6=0.75\}$ - Condition: self.incomes->exists (ocIsTypeOf(Pension))
3	Statistical characteristics of the current sample ($\mathcal{H}_{\text{current}}$ computed on L. 2 of Alg. 6). $\mathcal{H}_{\text{current}}$ differs from $\mathcal{H}_{\text{desired}}$ only in the relative frequencies.	Histogram $H1'$ for the sample (differs from $H1$ on row 2 above only in relative frequencies): - Relative Frequencies for $H1'$: $\{F1=0.9, F2=0.05, F3=0.05, F4=0\}$
		Histogram $H2'$ for the sample (differs from $H2$ on row 2 only in relative frequencies): - Relative Frequencies for $H2'$: $\{F5=0.5, F6=0.5\}$
4	freq_sensitivity.	0.03

From \mathcal{O} $\left\{ \begin{array}{l} ((\text{TaxPayer.allInstances}() \rightarrow \text{select}(\text{id} = 1) \rightarrow \\ \text{forall}(\text{not}(\text{birth_year} \geq 1979 \text{ and } \text{birth_year} \leq 1998))) \\ \text{and} \\ (\text{TaxPayer.allInstances}() \rightarrow \text{select}(\text{id} = 1) \rightarrow \\ \text{forall}(\text{not}(\text{birth_year} \geq 1959 \text{ and } \text{birth_year} \leq 1978)))) \\ \text{and} \end{array} \right.$

From \mathcal{U} $\left\{ \begin{array}{l} (\text{TaxPayer.allInstances}() \rightarrow \text{select}(\text{id} = 1) \rightarrow \\ \text{forall}(\text{birth_year} \geq 1900 \text{ and } \text{birth_year} \leq 1933)) \end{array} \right.$

Figure 6.4. Intermediate OCL Constraint for Distribution $H1'$ in Table 6.1

the output of this stage from the final corrective constraint built in the next (third) stage of the algorithm, described later. In particular, in the final corrective constraint, we have to account for the fact that some histograms apply only under certain conditions. For example, histogram $H2$ on row 2 of Table 6.1 applies to pensioners only. This detail is not captured by the literals in \mathcal{O} and \mathcal{U} .

The construction of the intermediate constraint is straightforward, noting that we take the *conjunction* of the literals in \mathcal{O} which prescribe *exclusions*, and the *disjunction* of the literals in \mathcal{U} which prescribe *inclusions* (L. 20). In Fig. 6.4, we provide an example of an intermediate constraint for histogram $H1'$, shown on row 3 of Table 6.1.

3) *Generate final constraints (L. 22-41 of Alg. 6)*: In the third (and final) stage, the algorithm (1) adds to the intermediate constraints conditions that describe under what circumstances these constraints apply (L. 27-36), and (2) combines the intermediate constraints, now complemented with conditions, into corrective constraints (L. 25, 37 and 39). Fig. 6.5 shows the final corrective constraint for the example of Table 6.1. Note that other examples of corrective constraints, where the domain element is not an attribute, can be found in Appendix D.2.

Instead of describing the final corrective constraint of Fig. 6.5, we show in Table 6.2 all possi-

Intermediate OCL constraint for distribution $H1'$

$$\left\{ \begin{array}{l} ((\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{not}(\text{birth_year} \geq 1979 \text{ and } \text{birth_year} \leq 1998))) \text{ and} \\ (\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{not}(\text{birth_year} \geq 1959 \text{ and } \text{birth_year} \leq 1978)))) \text{ and} \\ (\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \text{forAll}(\text{birth_year} \geq 1900 \\ \text{and } \text{birth_year} \leq 1933))) \\ \text{or} \\ ((\text{if}(\text{self.incomes} \rightarrow \text{notEmpty}()) \text{ then } \text{self.incomes} \rightarrow \text{exists}(\text{oclIsTypeOf}(\text{Pensions_and_Annuities_Income})) \text{ else } \text{false} \text{ endif}) \\ \text{and} \\ ((\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{birth_year} \geq 1957 \text{ and } \text{birth_year} \leq 1960)) \text{ and} \\ (\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{not}(\text{birth_year} \geq 1917 \text{ and } \text{birth_year} \leq 1956)))) \end{array} \right.$$

Condition of distribution $H2'$

$$\left\{ \begin{array}{l} ((\text{if}(\text{self.incomes} \rightarrow \text{notEmpty}()) \text{ then } \text{self.incomes} \rightarrow \text{exists}(\text{oclIsTypeOf}(\text{Pensions_and_Annuities_Income})) \text{ else } \text{false} \text{ endif}) \\ \text{and} \\ ((\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{birth_year} \geq 1957 \text{ and } \text{birth_year} \leq 1960)) \text{ and} \\ (\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{not}(\text{birth_year} \geq 1917 \text{ and } \text{birth_year} \leq 1956)))) \end{array} \right.$$

Intermediate OCL constraint for distribution $H2'$

$$\left\{ \begin{array}{l} ((\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{birth_year} \geq 1957 \text{ and } \text{birth_year} \leq 1960)) \text{ and} \\ (\text{Tax_Payer.allInstances}() \rightarrow \text{select}(id = 1) \rightarrow \\ \text{forAll}(\text{not}(\text{birth_year} \geq 1917 \text{ and } \text{birth_year} \leq 1956)))) \end{array} \right.$$

Figure 6.5. Final Corrective OCL Constraint for the Example of Table 6.1

ble scenarios for composing a corrective constraint from the set of histograms that annotate a given data schema element. In the first scenario (row 1 of Table 6.2), there is no condition involved. The corrective constraint is thus the same as the intermediate constraint built for the unconditional histogram (L. 25 of Alg. 6). In the second scenario (row 2 of Table 6.2), the algorithm first complements with conditions the intermediate constraints of the conditional histograms. The condition of one (conditional) histogram is naturally exclusive of the conditions of others (L. 33-36). This has been illustrated in the second column of Table 6.2. The constraint in Fig. 6.5 was generated according to the second scenario. The third scenario (row 3 of Table 6.2) is similar to the second scenario. The only difference is that, since there is no unconditional histogram, we need an extra clause to deal with the situation where none of the conditional histograms apply (L. 38-39). This “catch all” clause ensures that the final corrective constraint will not impact an instance model to which none of the conditional histograms should apply.

Table 6.2. Scenarios for Composing Corrective Constraints

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

6.4 Evaluation

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

6.4.1 Research Questions (RQs)

Our evaluation aims to answer the following RQs:

RQ1: *How does the customized OCL solver fare against the baseline OCL solver?* As discussed in Section 6.3.1, we customize a baseline OCL solver [Ali et al., 2016]. RQ1 compares the customized solver against the baseline across two dimensions: (a) execution time, and (b) success rate, i.e., how often each solver succeeds in constructing a logically valid instance model.

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

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

The experimental setup for answering these RQs is elaborated in Section 6.4.4 alongside our results and discussion.

6.4.2 Implementation

The manual steps of our approach (Steps 1 and 2 in Fig. 6.2) can be done using any UML modeling environment, e.g., Papyrus [Eclipse Foundation, NAB]. For writing OCL constraints (Step 3), we recommend Eclipse’s completeOCL editor which provides syntax validation and auto-completion assistance. To evaluate and parse the OCL constraints, we use EclipseOCL [Eclipse Foundation, NAa]. Our data generator delegates several of its tasks, e.g., calculating Euclidean distances, to the Apache Commons Mathematics Library [Apache Foundation, 2016]. Our data generator, which is available at people.svv.lu/tools/SDG/, has been implemented in Java using the Eclipse Modeling Framework (EMF) [Eclipse Foundation, 2002]. Excluding comments and third-party libraries, our data generator is approximately 39K lines of code.

6.4.3 Case Study Description

Our case study is motivated by an anticipated difficulty that acceptance testing of a public administration IT system in Luxembourg will pose, once the development of the system is completed. For this system, many of the software development and testing activities have been commissioned to third-parties. Since the actual data that the system will manipulate is sensitive and of a personal nature, sharing the data with third-parties poses complications. Further, there are gaps in the actual data as well as structural mismatches between the data schema used by the system under development and

Table 6.3. Comparison against the Baseline Solver (RQ1)

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

the data schema in which the historical records have been archived. Due to these issues, our collaborating partners have concluded that the most practical way to ascertain reliability is through testing the system using synthetic test data.

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

The validity constraints over the data are expressed using 68 OCL invariants available in Appendix D.1. Of these, 26 target avoiding logical anomalies (e.g., children being older than their parents). Of the remaining 42 constraints, 30 are implied by the ranges (upper and lower bounds) of the probabilistic annotations, and the final 12 are multiplicity constraints from the data schema. The constraints include 10 nested if-then-else expressions, 7 occurrences of OCL quantifiers, 23 variable declarations, 107 references to predefined OCL operations, and 212 logical operators.

6.4.4 Results and Discussion

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

RQ1: To answer RQ1, we attempted to generate 100 valid instance models with both the customized and the baseline solver. In this experiment, we considered only the validity constraints of our case study, without taking representativeness into account. We recall that in contrast to the baseline solver which starts from a randomly-generated instance model, the customized solver is seeded with the output of the data generator presented in Section 6.1. Further, the two solvers differ in their strategy for exploring the search space as discussed in Section 6.3.1. In Table 6.3, we report the execution time and success rate of the two solvers in the 100 attempts made. We note that different runs of the customized solver were seeded with different and randomly-selected initial instance models. None of these initial instance models were valid.

As shown in Table 6.3, the customized solver is on average ≈ 3 times faster than the baseline solver. More importantly, the customized solver is on average ≈ 4 times more likely to succeed in reaching a valid instance model. Stated otherwise, the customized solver produces a valid instance

model in much fewer runs, thus significantly decreasing wasted time and CPU usage when compared to the baseline. The observed improvements are explained mainly by two factors: First, the customized solver has a better starting point (initial instance model) which is easier to make valid. And second, the customized solver has a strategy (explained in Section 6.3.1) for avoiding entrapment in regions of the search space that do not contain any valid solutions.

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

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

For this experiment, we configured our data generator as follows: (a) The number of times the solver is invoked over a given instance model in order to create tweaked instance models (parameter `nb_attempts` of Alg. 5) is set to two, and (b) the margin for comparing relative frequencies (parameter `freq_sensitivity` of Alg. 5) is set to 0.01. This means that a difference of 1% between a relative frequency in the data sample and the corresponding frequency in the desired characteristics will prompt our data generator to take corrective action.

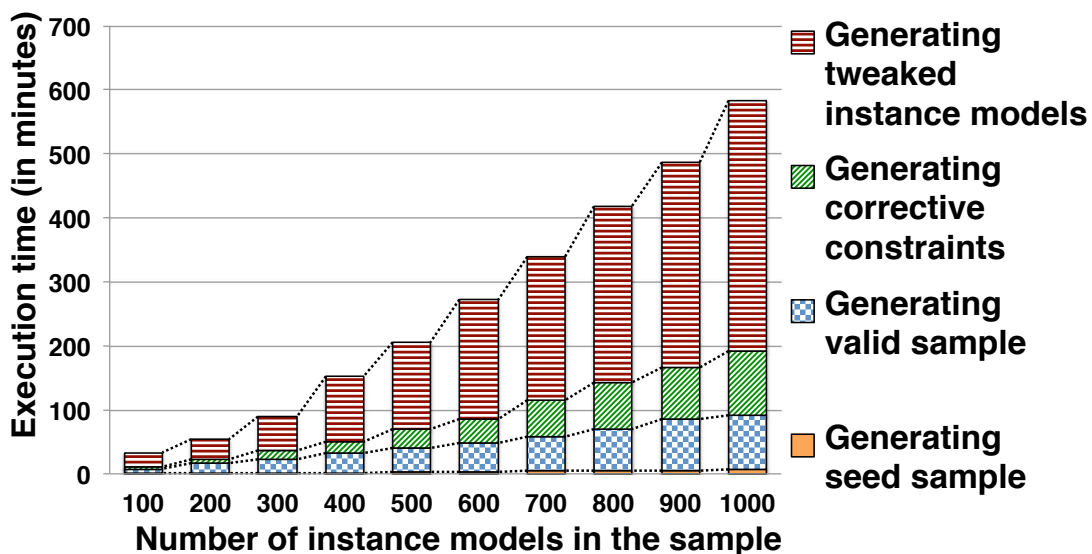


Figure 6.6. Execution Times for Generating Valid Data Samples of Different Sizes

Average execution times for different sample sizes are shown in Fig. 6.6. For example, the average execution time (across five runs) for producing a data sample with 500 (valid) instance models is ≈ 200 minutes. Overall, we generated $(100 + 200 + \dots + 1000) \times 5 = 27500$ (valid) instance models. On average, an instance model from this cumulative population has 40 objects, 276 attribute values, and 37 object links. Average instance model size depends on the specific data profile of the system under test.

Fig. 6.6 further provides a breakdown of the execution times over the different steps of our data generator. The breakdown indicates: First, the time required for creating an initial seed sample is negligible. Second, the generation of corrective constraints (by Alg. 6) is highly scalable with its execution time showing a linear growth trend. Finally, the most computationally-intensive steps are those involving constraint solving (i.e., generating valid sample and generating tweaked instance models in Fig. 6.6). Constraint solving accounts on average for 85% of the execution time. Despite its complexity, our data generator could produce in less than ten hours a data sample with 1000 instance models (i.e., 1000 test cases). This execution time is practical in our context because, in the worst case, the data can be generated overnight. Indeed, since data profiles are often stable, one can imagine that the test data can be generated early on and well before the testing phase. For systems with more complex data schemas, parallelization can be considered, noting that the solver technology underlying our approach is search-based and easily parallelizable [Ali et al., 2016].

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

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

As noted in Section 6.3.2, we use the Euclidean distance metric for guiding data generation. Euclidean distance is nevertheless not the only metric that one can use for quantifying representativeness. To gain more thorough insights about the representativeness of the data samples generated by our approach, we employ two additional distance metrics, namely Manhattan and Canberra [Cha, 2007]. These additional metrics were selected on the basis of the following criteria: (1) they, alongside Euclidean distance, are among the most commonly-used distance metrics for comparing distributions [Cha, 2007], and (2) robust implementations of the metrics were readily available [Apache Foundation, 2016]. These two new distance metrics are interpreted in the same way as Euclidean distance: the closer the distance is to zero, the better aligned a given pair of distributions are. Using these additional metrics in our evaluation helps ensure that our results are not strongly biased toward the specific notion of representativeness induced by Euclidean distance.

Figs. 6.7(a)–(c) respectively show the representativeness results computed by the Euclidean, Manhattan, and Canberra distance metrics. For each sample size, distances are computed for: (1) the seed (potentially invalid) data sample (2) the initial valid data sample built based on the seed sample, and (3) the final sample returned by our data generator. These distances are denoted d_1 , d_2 and d_3 as shown in Fig. 6.7. The difference between d_2 and d_1 results from fixing the logical anomalies in the seed sample. The difference between d_3 and d_2 is the improvement induced by the corrective con-

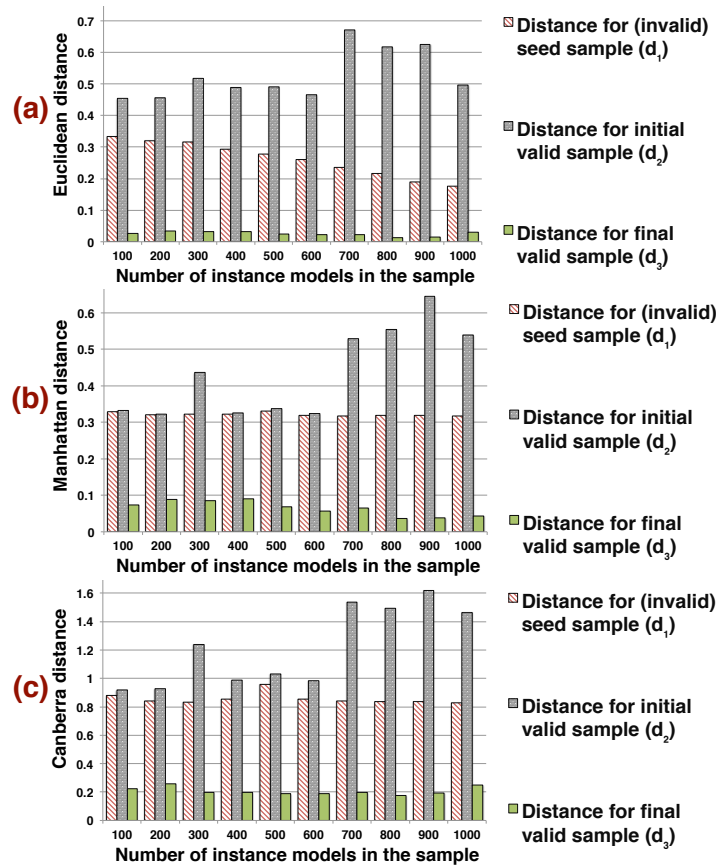


Figure 6.7. Distance between Generated Sample and Desired Distributions: (a) Euclidean Distance, (b) Manhattan Distance, and (c) Canberra Distance

straints. The difference between d_3 and d_1 indicates the improvement in representativeness brought about by our data generator when compared to the representativeness of the seed sample.

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

When considering the final data samples, the largest standard deviation observed in distances across the five runs made for each sample size was ≈ 0.004 (not shown in Fig. 6.7). This provides confidence that random variation has little influence over the representativeness of the final data samples.

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

6.4.5 Threats to Validity

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

Conclusion validity. As stated in Section 6.4.3, our case study was prompted by a foreseen difficulty in the acceptance testing of a system that is still under development. The unavailability of the final system prevented us from using the data generated by our approach for system-level testing. This leaves the possibility that the system may require test data beyond what was generated. To mitigate this threat, we ensured that the data schema was validated by domain experts. Further, since the system is an operationalization of procedures described in taxation and social security laws, we were able to check our data schema against legal provisions and make sure that no important concepts were overlooked. We thus believe that the likelihood of major omissions in our data schema is low.

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

6.5 Related Work

Usage profiles. We have already (in the beginning of this chapter) compared our work with the existing literature on usage profiles. Without repeating what was already said, we make some additional remarks. Existing usage profiles mainly target embedded and web-based systems. The behaviors of these systems typically lend themselves to being modeled as states and transitions (for web-based systems, web pages represent states, and clicks on links and buttons represent transitions [Tonella and Ricca, 2004]). State-machine-like notations such as Markov chains therefore provide a convenient way to build usage profiles for these systems. Our work in contrast focuses on systems whose behavior is driven by data that is interdependent and subject to complex logical constraints. A data schema enhanced with probabilistic information and constraints is a more natural choice for encoding usage profiles in our application context.

Synthetic data generation. The ability to create synthetic data is an integral part of automated test case generation. Since, in practice, it is often infeasible to cover all possible test scenarios, test case generation (and thus the underlying data generation strategy) is typically targeted at optimizing some notion of coverage, e.g., state or path coverage [Ali et al., 2010]. Metaheuristic search is widely used for generating data to support coverage-based testing [Ali et al., 2010]. Our data generation strategy relies on search, but rather than attempting to maximize some coverage criterion, we try to achieve statistical representativeness.

In the context of model-based development, data generation has been considered from many angles, including model verification and model-based testing. The most notable tool to mention here is Alloy [Jackson, 2012], which provides a specification language based on first-order logic and a SAT-based model finder. Another interesting work strand is UML2CSP [Cabot et al., 2014], where constraint programming is employed for generating instance models that satisfy a given set of OCL constraints. In theory, we could have employed in our approach either Alloy or UML2CSP for constraint solving. Nevertheless, due to the technical limitations already discussed in Section 6.3.1, most importantly scalability, we opted for a search-based solution.

Aside from the above work, a number of heuristic techniques exist for generating large synthetic data. Notably, Hartmann et al. [Hartmann et al., 2014] propose a rule-based technique for generating realistic smart grid instances according to the grid’s known topological characteristics. And, Mougnot et al. [Mougnot et al., 2009] adopt random sampling for generating large models in linear time. These techniques cannot enforce complex validity constraints over data. The techniques, on their own, are therefore not sufficient for achieving our goals in this chapter.

Whole test suite generation. Whole test suite generation builds an entire test suite by simultaneously optimizing multiple fitness functions (e.g., for multiple coverage criteria) [Fraser and Arcuri, 2013, Rojas et al., 2017]. In principle and with appropriate fitness functions defined for validity and representativeness, the problem addressed in this chapter can be formulated as whole test suite generation. The realization is however impractical: Whole test suite generation has been applied mainly to unit testing, where the test cases are small. In our context, test cases are much larger and are composed of complex and interdependent data elements. A whole test suite would therefore be prohibitively large for being manipulated by search. Further, our goal is not to optimize validity, but rather to guarantee it while optimizing representativeness. Our approach therefore takes a different route than whole test suite generation. We achieve validity and representativeness separately. Specifically, we start with a representative but invalid test suite. We make this test suite valid, but in the process, reduce its representativeness. At the end, we optimize representativeness without affecting validity.

6.6 Conclusion

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

Chapter 7

Conclusion

This chapter summarizes the research contributions of this dissertation and discusses potential areas for future work.

7.1 Summary

In this dissertation, we proposed a model-based framework for facilitating a number of laborious tasks in modeling, simulation, and testing of legal policies stipulated by laws and regulations. The main peculiarity of our framework is that it raises the level of abstraction at which legal policies are specified by relying on intuitive, simplified and tailored UML models. Our models can be viewed as a common repository of legal knowledge that bridges (to a large extent) an observed communication gap between software engineers and legal experts, and yet enables automated simulation of legal policies and system compliance checking. Broadly, models are used to derive the necessary simulation infrastructure for (1) deriving test oracles that enables the verification of the system's legal compliance, and (2) for anticipating the impact of changes in legal policies on the real world, thus enabling the analysis of legal requirements changes before they are implemented. Furthermore, our models are the basis for generating complex test cases that are designed for testing the reliability of data intensive systems such as public administration IT systems. We have empirically evaluated our framework using a sizeable case study conducted in collaboration with public administrations. The results were promising and showed that the technologies we employ are: scalable, usable in real settings, able to produce accurate results, and suitable for supporting both simulation and statistical usage-based testing even when no operational data is available.

In short, this dissertation made the following contributions:

Chapter 3 presented our UML-based methodology for modeling legal policies and our model-to-text transformation for converting policy models into different executable languages for the purpose of simulation and testing. Our experience in modeling Luxembourg's Income Tax Law provides evidence that our models requires a reasonable level of effort to be built and are expressive enough for adequately capturing the subtleties of prescriptive laws. Most importantly, by relying on our

models instead of raw legal text, we could successfully engage with legal experts and, with adequate training, the experts could independently understand and review our models.

Chapter 4 described our model-based simulation approach for generating test oracles (legal compliance) and for assessing the impact of envisaged legal changes (decision-making). The approach is supported by a prototype tool named *PoliSim*. One of the main advantages of our approach is that it is capable of producing artificial but yet realistic simulation data, i.e., data that exhibit the probabilistic characteristics of the simulation population. Our evaluation indicates that our approach is scalable, effective in creating simulation data, and consistent over several runs, i.e., random variation has little impact on simulation results. Remarkably, our simulation approach was able to generate then simulate up to 10,000 tax cases (for six legal policies) in ≈ 50 minutes.

Chapter 5 reported on a real-world case study where we used our model-based simulation framework to investigate how a potential legal reform is likely to impact personal income taxes in Luxembourg. From an industry viewpoint, the case study was a vehicle to provide the government with a complete diagnosis about the economic implications of the envisaged reform (see [Soltana et al., 2016b]). From a research viewpoint, this case study evaluated the feasibility and usefulness of our framework when used in practice. In particular, our evaluation demonstrated that the framework could yield credible and accurate simulation results while relying on artificial simulation data.

Chapter 6 presented our approach for building synthetic test data for reliability testing. The approach relies on an (enhanced) OCL solver that tweaks initially invalid synthetic data in order to produce statistically representative and logically valid test case data. The approach is supported by a prototype tool available at people.svv.lu/tools/SDG/. The empirical evaluation of the approach suggests that our approach can generate within practical time high-quality test data. In particular, the underlying data generator could produce test data samples with up to 1000 instance models in less than ten hours, surpassing the state-of-the-art in terms of representativeness and execution time.

7.2 Future Work

In this dissertation, we focused on modeling and analyzing prescriptive legal frameworks where legal policies represent some procedural rules that need to be applied for compliance. In the future, we would like to investigate how the solutions presented in this thesis can be extended to accommodate declarative frameworks where compliance is defined through notions such as prohibitions, permissions, and obligations [Boella and van der Torre, 2003], e.g., civil code laws.

Currently, our simulator only supports basic simulation scenarios such as result differencing, i.e., when an original and a modified set of policies are executed over the same simulation data to quantify the impact of given change in the law. In the future, we would like to investigate search-based techniques to support more advanced simulation scenarios. One advanced simulation scenario is the identification of undesirable situations, e.g., the identification of exploitable loop-holes by legal means, that might result from a given change in the law. Further, we would like to extend our current

simulator to incorporate time. Such extension would make it possible to answer questions like how the revenue is likely to evolve over the next ten years.

Finally, we would like to further improve the synthetic data generator presented in Chapter 6 to ultimately transform it into a general purpose (vs. specific to statistical testing), robust and professionally built tool. Such tool could be for example integrated into the Papyrus modeling environment [Eclipse Foundation, NAb] to support the instantiation of class diagrams under certain (OCL) constraints. A key challenge here is to further improve the performance of the solver used by the generator in terms of execution time and success rate (in particular for extremely complex constraints and large class diagrams). In particular, we want to explore using a combination of search and Satisfiability Modulo Theories (SMT) as a novel hybrid approach for solving complex OCL constraints.

List of Papers

Published papers included in this dissertation:

1. Ghanem Soltana, Elizabeta Fourneret, Morayo Adedjouma, Mehrdad Sabetzadeh, and Lionel C. Briand. “**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)*, Valencia, Spain, September 28 - October 3, pp. 450-466, 2014.
2. Ghanem Soltana. “**A Model-Based Framework for Legal Policy Simulation and Legal Compliance Checking**”. In *Proceedings of Doctoral Symposium at the 18th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS’15)*, Ottawa, Canada, September 29, <http://ceur-ws.org/Vol-1531/paper1.pdf>, 2015.
3. Ghanem Soltana, Nicolas Sannier, Mehrdad Sabetzadeh, and Lionel C. Briand. “**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)*, Ottawa, Canada, September 30 - October 2, pp. 70-79, 2015.
4. Ghanem Soltana, Nicolas Sannier, Mehrdad Sabetzadeh, and Lionel C. Briand. “**Model-Based Simulation of Legal Policies: Framework, Tool Support, and Validation**”. In *Software & Systems Modeling (SoSyM)*, Springer, (in press).
5. Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. “**Model-Based Simulation of Legal Requirements: Experience from Tax Policy Simulation**”. In *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE’16)*, Beijing, China, September 12-16, pp. 303-312, 2016.
6. Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. “**Synthetic Data Generation for Statistical Testing**”. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE’17)*, Illinois, USA, 2017, (to appear).

Bibliography

- [Al-Azizy et al., 2015] Al-Azizy, D., Millard, D., Symeonidis, I., O’Hara, K., and Shadbolt, N. (2015). A literature survey and classifications on data deanonymisation. In *Proceedings of 10th International Conference on Risks and Security of Internet and Systems (CRiSIS’10)*, pages 36–51. Springer.
- [Ali et al., 2010] Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762.
- [Ali et al., 2013] Ali, S., Iqbal, M. Z., Arcuri, A., and Briand, L. C. (2013). Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402.
- [Ali et al., 2016] Ali, S., Iqbal, M. Z., Khalid, M., and Arcuri, A. (2016). Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach. *Empirical Software Engineering*, 21(6):2459–2502.
- [Amàlio et al., 2010] Amàlio, N., Kelsen, P., Ma, Q., and Glodt, C. (2010). Using VCL as an Aspect-Oriented Approach to Requirements Modelling. *Transactions on Aspect-Oriented Software Development*, 7:151–199.
- [Anastasakis et al., 2010] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2010). On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, 9(1):69–86.
- [Apache Foundation, 2016] Apache Foundation (2003-2016). Apache commons mathematics library. <http://commons.apache.org/proper/commons-math/>, last accessed: May 2017.
- [Behjati et al., 2014] Behjati, R., Nejati, S., and Briand, L. C. (2014). Architecture-level configuration of large-scale embedded software systems. *ACM Transactions on Software Engineering and Methodology*, 23(3):25:1–25:43.
- [Bench-Capon et al., 2012] Bench-Capon et al., T. (2012). 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.

- [Boella and van der Torre, 2003] Boella, G. and van der Torre, L. (2003). Permissions and obligations in hierarchical normative systems. In *Proceedings of 9th International Conference on Artificial Intelligence and Law (ICAIL'03)*, pages 109–118. ACM.
- [Bottoni et al., 2000] Bottoni, P., Koch, M., Parisi-Presicce, F., and Taentzer, G. (2000). Consistency checking and visualization of OCL constraints. In *Proceedings of 3rd International Conference on The Unified Modeling Language: Advancing the Standard (UML'00)*, pages 294–308.
- [Bourguignon et al., 1988] Bourguignon, F., Chiappori, P., and Sastre, J. (1988). SYSIFF: a simulation program of the french tax-benefit system. *Tax Benefit Models*, 10.
- [Bousse et al., 2014] Bousse, E., Combemale, B., and Baudry, B. (2014). Scalable armies of model clones through data sharing. In *Proceedings of 17th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, pages 286–301.
- [Breaux, 2009a] Breaux, T. (2009a). Exercising due diligence in legal requirements acquisition: A tool-supported, frame-based approach. In *Proceedings of 17th IEEE International Requirements Engineering Conference (RE'09)*, pages 225–230.
- [Breaux, 2009b] Breaux, T. (2009b). *Legal Requirements Acquisition for the Specification of Legally Compliant Information Systems*. PhD thesis, North Carolina State University.
- [Breaux and Anton, 2008] Breaux, T. and Anton, A. (2008). Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, 34(1):5–20.
- [Breaux and Powers, 2009] Breaux, T. D. and Powers, C. (2009). Early studies in acquiring evidentiary, reusable business process models from laws for legal compliance. In *Proceedings of 6th International Conference on Information Technology: New Generations (ITNG'09)*, pages 272–277.
- [Breuker et al., 1997] Breuker, J., Valente, A., Winkels, R., et al. (1997). Legal ontologies: a functional view. In *Proceedings of 1st LegOut Workshop on Legal Ontologies*, pages 23–36. Citeseer.
- [Cabot et al., 2010] Cabot, J., Clarisó, R., Guerra, E., and Lara, J. (2010). A UML/OCL framework for the analysis of graph transformation rules. *Software & Systems Modeling*, 9(3):335–357.
- [Cabot et al., 2008] Cabot, J., Clariso, R., and Riera, D. (2008). Verification of UML/OCL class diagrams using constraint programming. In *Proceedings of 1st IEEE International Conference on Software Testing Verification and Validation (ICST'08)*, pages 73–80.
- [Cabot et al., 2014] Cabot, J., Clarisó, R., and Riera, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23.
- [Canova et al., 2009] Canova, L., Piccoli, L., and Spadaro, A. (2009). SYSIFF 2006: A microsimulation model for the French tax system. Technical report, MicroSimula - Paris School of Economics.

- [Cha, 2007] Cha, S.-H. (2007). Comprehensive survey on distance/similarity measures between probability density functions. *Mathematical Models and Methods in Applied Sciences*, 1(2):300–307.
- [Corbin and Strauss, 2008] Corbin, J. and Strauss, A. (2008). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 3rd edition.
- [Corder and Foreman, 2014] Corder, G. W. and Foreman, D. (2014). *Nonparametric Statistics: A Step-by-Step Approach*. John Wiley & Sons.
- [Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3rd edition.
- [Cunha et al., 2015] Cunha, A., Garis, A., and Riesco, D. (2015). Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, 14(1):5–25.
- [De Capitani di Vimercati et al., 2010] De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., and Samarati, P. (2010). Fragments and loose associations: Respecting privacy in data publishing. *Proceedings of Very Large Data Bases Endowment*, 3(1):1370–1381.
- [de la Vara and Panesar-Walawege, 2013] de la Vara, J. L. and Panesar-Walawege, R. K. (2013). Safetymet: A metamodel for safety standards. In *Proceedings of 16th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'13)*, pages 69–86.
- [Di Nardo et al., 2015] Di Nardo, D., Pastore, F., and Briand, L. C. (2015). Generating complex and faulty test data through model-based mutation analysis. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15)*, pages 1–10.
- [Eclipse Foundation, 2002] Eclipse Foundation (2002). EMF: Eclipse Modeling Framework. <http://www.eclipse.org/emf>, last accessed: May 2017.
- [Eclipse Foundation, 2006] Eclipse Foundation (2006). Acceleo - transforming models into code. <http://www.eclipse.org/acceleo/>. Last accessed: May 2017.
- [Eclipse Foundation, NAa] Eclipse Foundation (NAa). Eclipse OCL. <http://www.eclipse.org/modeling/mdt/?project=ocl>, last accessed: May 2017.
- [Eclipse Foundation, NAb] Eclipse Foundation (NAb). Papyrus modeling environment. <https://eclipse.org/papyrus/>, last accessed: May 2017.
- [Emmerich et al., 1999a] Emmerich, W., Finkelstein, A., Montangero, C., Antonelli, S., Armitage, S., and Stevens, R. (1999a). Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):826–851.
- [Emmerich et al., 1999b] Emmerich, W., Finkelstein, A., Montangero, C., Antonelli, S., Armitage, S., and Stevens, R. (1999b). Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):338–342.

- [Figari et al., 2015] Figari, F., Paulus, A., and Sutherland, H. (2015). Microsimulation and policy analysis. *Handbook of Income Distribution*, 2:2141–2221.
- [Fraser and Arcuri, 2013] Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291.
- [General Data Protection Regulation, 2016] General Data Protection Regulation (2016). General Data Protection Regulation (Regulation (EU) 2016/679).
- [Ghanavati et al., 2007] Ghanavati, S., Amyot, D., and Peyton, L. (2007). Towards a framework for tracking legal compliance in healthcare. In *Proceedings of 19th International Conference on Advanced Information Systems Engineering (CAiSE'07)*, pages 218–232.
- [Ghanavati et al., 2014] Ghanavati, S., Rifaut, A., Dubois, E., and Amyot, D. (2014). Goal-oriented compliance with multiple regulations. In *Proceedings of 22nd IEEE International Conference on Requirements Engineering (RE'14)*, pages 73–82.
- [Goedertier and Vanthienen, 2006] Goedertier, S. and Vanthienen, J. (2006). Designing compliant business processes with obligations and permissions. In *Proceedings of 7th Workshop on Business Process Management (BPM'06)*, pages 5–14.
- [Gogolla et al., 2005] Gogolla, M., Bohling, J., and Richters, M. (2005). Validating UML and OCL models in USE by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398.
- [Government of Luxembourg, 2013] Government of Luxembourg (2013). Modified income tax law of december 4, 1967.
- [Guen et al., 2004] Guen, H. L., Marie, R., and Thelin, T. (2004). Reliability estimation for statistical usage testing using markov chains. In *Proceedings of 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 54–65. IEEE.
- [Hammond and Bickel, 2015] Hammond, R. K. and Bickel, J. E. (2015). Discretization methods for continuous probability distributions. In *Wiley Encyclopedia of Operations Research and Management Science*. Wiley.
- [Hartmann et al., 2014] Hartmann, T., Fouquet, F., Klein, J., Le Traon, Y., Pelov, A., Toutain, L., and Ropitault, T. (2014). Generating realistic smart grid communication topologies based on real-data. In *Proceedings of 5th IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*, pages 428–433.
- [Hassan and Logrippo, 2008] Hassan, W. and Logrippo, L. (2008). Requirements and compliance in legal systems: a logic approach. In *Proceedings of 1st International Workshop on RE and Law (RELAW'08)*, pages 40–44.
- [Herbold et al., 2017] Herbold, S., Harms, P., and Grabowski, J. (2017). Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. *International Journal on Software Tools for Technology Transfer*, 19(3):309–324.

- [Hermans et al., 2012] Hermans, F., Pinzger, M., and van Deursen, A. (2012). Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of 34th IEEE International Conference on Software Engineering (ICSE'12)*, pages 441–451.
- [Hermans et al., 2015] Hermans, F., Pinzger, M., and van Deursen, A. (2015). Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575.
- [Hurley, 2014] Hurley, P. (2014). *A concise introduction to logic*. Nelson Education.
- [IBM, 2003a] IBM (2003a). IBM Rational Software Architect Simulation Toolkit. <http://www-03.ibm.com/software/products/en/ratisoftarchsimutool>, last accessed: May 2017.
- [IBM, 2003b] IBM (2003b). Rational Rhapsody Designer for Systems Engineers. <http://www-03.ibm.com/software/products/en/ratirhapdesiforsystengi>, last accessed: May 2017.
- [Ingolfo et al., 2014] Ingolfo, S., Siena, A., and Mylopoulos, J. (2014). Nòmos 3: Reasoning about regulatory compliance of requirements. In *Proceedings of 22nd IEEE International Conference on Requirements Engineering (RE'14)*, pages 313–314.
- [Ingolfo et al., 2013] Ingolfo, S., Siena, A., Mylopoulos, J., Susi, A., and Perini, A. (2013). Arguing regulatory compliance of software requirements. *Data & Knowledge Engineering*, 87:279–296.
- [Iqbal et al., 2013] Iqbal, M. Z., Arcuri, A., and Briand, L. C. (2013). Environment modeling and simulation for automated testing of soft real-time embedded software. *Software & Systems Modeling*, 14(1):483–524.
- [Islam et al., 2011] Islam, S., Mouratidis, H., and Jürjens, J. (2011). A framework to support alignment of secure software engineering with legal regulations. *Software & Systems Modeling*, 10(3):369–394.
- [Jackson, 2012] Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.
- [Jürjens, 2002] Jürjens, J. (2002). Umlsec: Extending UML for secure systems development. In *Proceedings of 5th International Conference on the Unified Modeling Language (UML'02)*, pages 412–425.
- [Kallepalli and Tian, 2001] Kallepalli, C. and Tian, J. (2001). Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036.
- [Korherr and List, 2006] Korherr, B. and List, B. (2006). Extending the UML 2 activity diagram with business process goals and performance measures and the mapping to BPEL. In *Proceedings of 2nd International Workshop on Best Practices of UML (ER BP-UML'06)*, pages 7–18.

- [Krieger and Knapp, 2008] Krieger, M. P. and Knapp, A. (2008). Executing underspecified OCL operation contracts with a SAT solver. *Electronic Communication of the European Association of Software Science and Technology*, 15:1–16.
- [Larman, 2004] Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall.
- [Luxembourg’s Ministry of Finance, 2015] Luxembourg’s Ministry of Finance (2015). Compendium sur les données statistiques des impôt luxembourgeois, 10th of march 2015.
- [Melz and Valente, 2004] Melz, E. and Valente, A. (2004). Modeling the tax code. In *Proceedings of 2nd International Workshop on Regulatory Ontologies (WORM’04)*, pages 652–661.
- [Mijatov et al., 2015] Mijatov, S., Mayerhofer, T., Langer, P., and Kappel, G. (2015). Testing functional requirements in UML activity diagrams. In *Proceedings of 9th International Conference on Tests and Proofs (TAP’15)*, pages 173–190.
- [Milanovic et al., 2007] Milanovic, M., Gasevic, D., Giurca, A., Gerd, W., and Devedzic, V. (2007). Towards sharing rules between OWL/SWRL and UML/OCL. *Electronic Communications of European Association of Software Science and Technology*, 5:2–19.
- [Miltersen et al., 2005] Miltersen, P. B., Radhakrishnan, J., and Wegener, I. (2005). On converting CNF to DNF. *Theoretical Computer Science*, 347(1-2):325–335.
- [Mougenot et al., 2009] Mougenot, A., Darrasse, A., Blanc, X., and Soria, M. (2009). Uniform random generation of huge metamodel instances. In *Proceedings of 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA’09)*, pages 130–145.
- [Musa, 1993] Musa, J. D. (1993). Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32.
- [No Magic, 2005] No Magic (2005). Cameo Simulation Toolkit. <http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html>, last accessed: May 2017.
- [Object Management Group, 2004] Object Management Group (2004). Object Constraint Language 2.4 Specification. <http://www.omg.org/spec/OCL/2.4/>, last accessed: May 2017.
- [Object Management Group, 2006] Object Management Group (2006). Meta object facility (MOF) core specification. <http://www.omg.org/spec/MOF/2.0/>, last accessed: May 2017.
- [Object Management Group, 2011a] Object Management Group (2011a). Modeling and Analysis of Real-time and Embedded Systems (MARTE), version 1.1. <http://www.omg.org/spec/MARTE/1.1/>, last accessed: May 2017.

- [Object Management Group, 2011b] Object Management Group (2011b). UML Superstructure Specification. <http://www.omg.org/spec/UML/2.4/Superstructure/PDF>, last accessed: May 2017.
- [Object Management Group, 2013] Object Management Group (2013). Semantics of a foundational subset for executable UML models (fUML), version 1.1. <http://www.omg.org/spec/FUML/1.1>, last accessed: May 2017.
- [Object Management Group, 2015] Object Management Group (2015). OMG Unified Modeling Language (UML). <http://www.omg.org/spec/UML/2.5>, last accessed: May 2017.
- [Panko, 1998] Panko, R. (1998). What we know about spreadsheet errors. *Journal of End User Computing*, 10:15–21.
- [Papyrus, 2009] Papyrus (2009). Moka overview. <https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution>, last accessed: May 2017.
- [Petersen, 2002] Petersen, K. (2002). The regulation of assisted reproductive technology: a comparative study of permissive and prescriptive laws and policies. *Journal of law and medicine*, 9(4):483–497.
- [Poore and Trammell, 1999] Poore, J. H. and Trammell, C. J. (1999). Application of statistical science to testing and evaluating software intensive systems. In Cohen, M. L., Steffey, D. L., and Rolph, J. E., editors, *Statistics, Testing, and Defense Acquisition*, chapter 3. National Academies Press.
- [Reynoso et al., 2004] Reynoso, L., Genero, M., and Piattini, M. (2004). Towards a metric suite for OCL expressions expressed within UML/OCL models. *Journal of Computer Science and Technology*, 4(1):38–44.
- [Risland and Skalak, 1991] Risland, E. and Skalak, D. (1991). CABARET: Rule interpretation in a hybrid architecture. *International Journal of Man-Machine Studies*, 34(6):839–887.
- [Rojas et al., 2017] Rojas, J. M., Vivanti, M., Arcuri, A., and Fraser, G. (2017). A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893.
- [Ruiter, 1993] Ruiter, D. (1993). *Institutional legal facts: legal powers and their effects*, volume 18. Springer Science & Business Media.
- [Runeson and Wohlin, 1995] Runeson, P. and Wohlin, C. (1995). Statistical usage testing for software reliability control. *Informatica*, 19(2):195–207.
- [Sannier et al., 2017] Sannier, N., Adedjouma, M., Sabetzadeh, M., and Briand, L. C. (2017). An automated framework for detection and resolution of cross references in legal texts. *Requirements Engineering*, 22(2):215–237.

- [Sannier and Baudry, 2014] Sannier, N. and Baudry, B. (2014). INCREMENT: A mixed MDE-IR approach for regulatory requirements modeling and analysis. In *Proceedings of 20th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'14)*, pages 135–151.
- [SAS Institute, 1996a] SAS Institute (1996a). Statistical Analysis System (SAS). <http://www.sas.com/>, last accessed: May 2017.
- [SAS Institute, 1996b] SAS Institute (1996b). Statistical Analysis System (SAS) for Econometrics and Time Series Analysis (ETS). <https://support.sas.com/documentation/onlinedoc/ets/>, last accessed: May 2017.
- [Smith, 2000] Smith, G. (2000). *The Object-Z specification language*. Kluwer.
- [Soltana et al., 2014] Soltana, G., Fourneret, E., Adedjouma, M., Sabetzadeh, M., and Briand, L. C. (2014). Using UML for modeling procedural legal rules: Approach and a study of luxembourg’s tax law. In *Proceedings of 17th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, pages 450–466.
- [Soltana et al., 2016a] Soltana, G., Sabetzadeh, M., and Briand, L. (2016a). Model-based simulation of legal requirements: Experience from tax policy simulation. In *Proceedings of 24th IEEE International Requirements Engineering Conference (RE'16)*. IEEE.
- [Soltana et al., 2017] Soltana, G., Sabetzadeh, M., and Briand, L. (2017). Synthetic data generation for statistical testing. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. (to appear).
- [Soltana et al., 2016b] Soltana, G., Sabetzadeh, M., Briand, L. C., , Prommenschenkel, T., and Balmer, L. (2016b). Policy simulation in action: A case study on how the potential abolishment of joint taxation will impact personal income taxes. Technical report, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. http://people.svv.lu/soltana/Simulation_Report_2016.pdf.
- [Soltana et al., 2016c] Soltana, G., Sannier, N., Sabetzadeh, M., and Briand, L. (2016c). Model-based simulation of legal policies: Framework, tool support, and validation. *Software & Systems Modeling*. (in press).
- [Soltana et al., 2015] Soltana, G., Sannier, N., Sabetzadeh, M., and Briand, L. C. (2015). A model-based framework for probabilistic simulation of legal policies. In *Proceedings of 18th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'15)*, pages 70–79.
- [Statistics Canada, 2015] Statistics Canada (2015). The Social Policy Simulation Database and Model (SPSD/M). <http://www.statcan.gc.ca/eng/microsimulation/spsdm/spsdm>, last accessed: May 2017.

- [Stein et al., 2004] Stein, D., Hanenberg, S., and Unland, R. (2004). A graphical notation to specify model queries for MDA transformations on UML models. In *Proceedings of 2nd Conference on Model Driven Architecture: Foundations and Applications (MDAFA'04)*, pages 60–74.
- [Sutherland, 1995] Sutherland, H. (1995). The development of tax-benefit models: a view from the UK. Technical report, University of Cambridge.
- [Tonella and Ricca, 2004] Tonella, P. and Ricca, F. (2004). Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):103–127.
- [Utting and Legeard, 2007] Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers.
- [Van Engers et al., 2008] Van Engers, T., Boer, A., Breuker, J., Valente, A., and Winkels, R. (2008). *Digital Government: E-Government Research, Case Studies, and Implementation*, chapter Ontologies in the Legal Domain, pages 233–261. Springer US.
- [van Engers et al., 2001] van Engers, T., Gerrits, R., Boekenoogen, M., Glassée, E., and Kordelaar, P. (2001). POWER: using UML/OCL for modeling legislation - an application report. In *Proceedings of 8th International Conference on Artificial Intelligence and Law (ICAIL'01)*, pages 157–167.
- [van Kralingen, 1997] van Kralingen, R. (1997). A conceptual frame-based ontology for the law. In *Proceedings of 1st International Workshop on Legal Ontologies (LEGONT'97)*, pages 15–22.
- [Whittaker and Poore, 1993] Whittaker, J. A. and Poore, J. H. (1993). Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106.
- [Zeni et al., 2015] Zeni, N., Kiyavitskaya, N., Mich, L., Cordy, J. R., and Mylopoulos, J. (2015). GaiusT: supporting the extraction of rights and obligations for regulatory compliance. *Requirements Engineering*, 20(1):1–22.

UML. Similarly, the special datatype `Domain Object` is not represented. This special datatype from the field study can map onto any instance of the domain model concepts (e.g, `TaxPayer`).

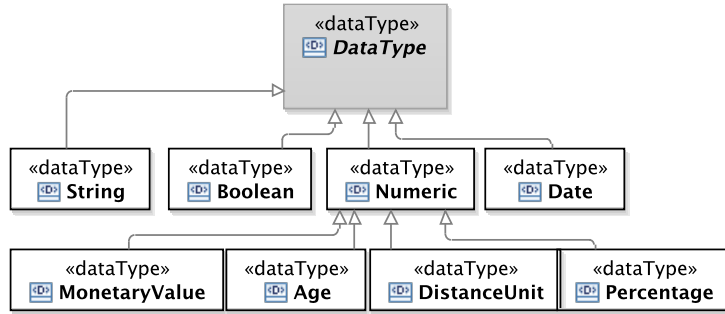


Figure A.2. Datatype Library of the Profile for Specifying Legal Policies

A.3 Consistency Constraints

In Table C.1, we show our profile’s consistency constraints. In the first column, we provide a description of the consistency constraints and their OCL formalization. In the second column, we provide the list of stereotypes on which each of consistency constraint is applied.

Table A.1. Consistency Constraints (Profile for Specifying Legal Policies)

Consistency constraint	Involved stereotype(s)
Description: An AD that represents a legal policy must have the stereotype «context». OCL constraint: <pre> context context inv : not self.base_Activity .getAppliedStereotype('Profile::context') .ocllsUndefined() </pre>	«context»
Description: If the stereotype «context» is applied to an AD, the <code>context_class</code> attribute of the stereotype cannot be null. OCL constraint: <pre> context context inv : not self.base_Activity .getAppliedStereotype('Profile::context') .ocllsUndefined() implies self.context_class.size()>0 </pre>	«context»
Description: Any input modeled by <code>InputPin</code> or <code>ActivityParameterNode</code> must use the stereotype «in».	«in»

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p>OCLE constraint:</p> <p>1. Input Pin constraint: context in inv: not self.base_InputPin. getAppliedStereotype('Profile::in'). oclIsUndefined()</p> <p>2. Activity Parameter Node constraint: context in inv: (not self.base_ActivityParameterNode. getApplicableStereotype('Profile::in'). oclIsUndefined()) and self.base_ActivityParameterNode. .parameter.direction = ParameterDirectionKind::in</p>	
<p>Description: The stereotype «out» is applicable only to outputs modeled by an OutputPin.</p> <p>OCLE constraint: context out inv: not self.base_OutputPin. getAppliedStereotype('Profile::out'). oclIsUndefined()</p>	«out»
<p>Description: If the stereotype «in» is applied to an input, the input must use one of the following stereotypes to specify the source where it comes from: «fromlaw», «fromrecord», «fromagent».</p> <p>OCLE constraint: context in inv: not self.base_ObjectNode .getAppliedStereotype('Profile::in').oclIsUndefined() implies self.base_ObjectNode.getAppliedStereotypes()->select(s s.qualifiedName() = 'Profile::fromlaw' or s.qualifiedName() = 'Profile::fromrecord' or s.qualifiedName() = 'Profile::fromagent')->size()=1</p> <p>Remark: The constraint applies to all types of input, modeled as either an ActivityParameterNode or an InputPin. In the OCLE constraint, the different metaclasses are referred to by their UML abstract class ObjectNode. Note that although a CentralBufferNode is an ObjectNode, it cannot be annotated by the stereotype «in» as it is not used as an input in the profile.</p>	«fromlaw», «fromrecord», «fromagent», «in»
<p>Description: The stereotypes «fromrecord», «fromlaw», «fromagent» cannot be applied simultaneously to an input (tagged with the stereotype «in»).</p>	«fromrecord», «fromlaw», «fromagent»

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p>OCLE constraint:</p> <pre> context source inv: self.base_ObjectNode.getAppliedStereotypes()->select(s s.qualifiedName() = 'Profile::fromlaw' or s.qualifiedName() = 'Profile::fromrecord' or s.qualifiedName() = 'Profile::fromagent')->size()=1 </pre>	
<p>Description: If any of the stereotypes «fromlaw», «fromrecord», «fromagent» is applied to an input, then the input must be associated with a Comment that is stereotyped with «query».</p> <p>OCLE constraint:</p> <pre> context source inv: let obj_node : ObjectNode = self.base_ObjectNode .oclAsType(ObjectNode) in self.obj_node.getAppliedStereotypes()->select(s s.qualifiedName() = 'Profile::fromlaw' or s.qualifiedName() = 'Profile::fromrecord' or s.qualifiedName() = 'Profile::fromagent')->notEmpty() implies self.obj_node.allOwnedElements()->select(s s.oclIsTypeOf(Comment) and (not s.oclAsType(Comment). getAppliedStereotype('Profile::query') .oclIsUndefined())->size()=1 or (not Comment.allInstances()->select(c c.annotatedElement-> select(el el.oclIsTypeOf(Pin) and el.oclAsType(Pin).qualifiedName = self.base_Pin.qualifiedName)->size()=1)->any(true). getAppliedStereotype('Profile::query'). oclIsUndefined() </pre>	<p>«fromlaw», «fromrecord», «fromagent», «query»</p>
<p>Description: If an input's source is «fromlaw», the associated query must meet the following constraints: source must not be null; question, constraint, and agent_role must be null.</p>	<p>«fromlaw», «query»</p>

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p>OCLE constraint:</p> <pre> context query inv: self.base_Comment.annotatedElement ->select (el not el. getAppliedStereotype('Profile :: fromlaw'). oclIsUndefined())->size ()=1 implies self.source.size () > 0 and self.question.size () = 0 and self.constraint->size ()=0 and self.agent_role=Agent_Role ::NONE </pre>	
<p>Description: If the input's source is «fromrecord», the associated query must meet the following constraints: question agent_role must be null; constraint must not be null.</p> <p>OCLE constraint:</p> <pre> context query inv: self.base_Comment.annotatedElement ->select (el not el. getAppliedStereotype('Profile :: fromrecord'). oclIsUndefined())->size ()=1 implies self.constraint->size ()=1 and self.agent_role = Agent_Role ::NONE and self.question.size () = 0 </pre>	«fromrecord», «query»
<p>Description: If the input's source is «fromagent», the associated query must meet the following constraints: source and constraint must be null; question and agent_role must not be null.</p> <p>OCLE constraint:</p> <pre> context query inv: self.base_Comment.annotatedElement ->select (el not el. getAppliedStereotype('Profile :: fromagent'). oclIsUndefined())->size ()=1 implies self.agent_role <> Agent_Role ::NONE and self.question.size ()>0 and self.source.size ()=0 and self.constraint->size ()=0 </pre>	«fromagent», «query»
<p>Description: Each operation must be annotated either by the stereotype «assert» or «calculate». These stereotypes cannot be applied simultaneously.</p>	«assert», «calculate»

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p> OCL constraint: context operation inv: not self.base_OpaqueAction. getAppliedStereotype('Profile::assert'). oclIsUndefined() xor not self.base_OpaqueAction. getAppliedStereotype('Profile::calculate'). oclIsUndefined() </p>	
<p> Description: A constraint must be annotated either by the stereotype «statement» to specify the operation’s statement, or by the stereotype «formula» to specify the operation’s formula. These stereotypes cannot be applied simultaneously. </p> <p> OCL constraint: context expression inv: not self.base_Constraint. getAppliedStereotype('Profile::statement'). oclIsUndefined() xor not self.base_Constraint. getAppliedStereotype('Profile::formula'). oclIsUndefined() </p>	<p>«statement», «formula»</p>
<p> Description: An operation annotated with «calculate» must have some outgoing flow to an intermediate variable. </p> <p> OCL constraint: context calculate inv: self.base_OpaqueAction. getAppliedStereotype('Profile::calculate'). oclIsUndefined() implies self.base_OpaqueAction.output.outgoing → select(ele.target.oclIsTypeOf(CentralBufferNode)) →notEmpty() </p>	<p>«calculate», «intermediate»</p>
<p> Description: If an operation is annotated with «calculate», then the operation must have associated with it a constraint with the stereotype «formula». </p>	<p>«calculate», «formula»</p>

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p>OCL constraint:</p> <pre> context calculate inv: not self.base_OpaqueAction. getAppliedStereotype('Profile::calculate'). oclIsUndefined() implies self.base_OpaqueAction.allOwnedElements()->select(ele.oclIsTypeOf(Constraint) and not e.oclAsType(Constraint). getAppliedStereotype('Profile::formula'). oclIsUndefined())->size()=1 or (not Constraint.allInstances()->select(c c.constrainedElement->select(el el.oclIsTypeOf(OpaqueAction) and el.oclAsType(OpaqueAction).qualifiedName = self.base_OpaqueAction.qualifiedName)->size()=1) ->any(true). getAppliedStereotype('Profile::formula'). oclIsUndefined() </pre>	
<p>Description: If an operation is annotated with «assert», then the operation must have associated with it a constraint with the stereotype «statement».</p> <p>OCL constraint:</p> <pre> context assert inv: not self.base_OpaqueAction. getAppliedStereotype('Profile::assert').oclIsUndefined() implies self.base_OpaqueAction.allOwnedElements()->select(ele.oclIsTypeOf(Constraint) and not e.oclAsType(Constraint). getAppliedStereotype('Profile::statement'). oclIsUndefined())->size()=1 or (not Constraint.allInstances()->select(c c.constrainedElement->select(el el.oclIsTypeOf(OpaqueAction) and el.oclAsType(OpaqueAction).qualifiedName = self.base_OpaqueAction.qualifiedName)->size()=1) ->any(true). getAppliedStereotype('Profile::statement'). oclIsUndefined() </pre>	«assert», «statement»
<p>Description: Each ExpansionRegion must be annotated with the stereotype «iterative» (the only execution mode allowed by our profile).</p> <p>OCL constraint:</p> <pre> context iterative inv: not self.base_ExpansionRegion. getAppliedStereotype('Profile::iterative'). oclIsUndefined() </pre>	«iterative»

Consistency Constraints

Consistency constraint	Involved stereotype(s)
<p>Description: The mode attribute of the ExpansionRegion must be <i>iterative</i> (the application of the «iterative» stereotype per se does not ensure that the mode is set correctly).</p> <p>OCL constraint:</p> <pre> context iterative inv: not self.base_ExpansionRegion. getAppliedStereotype('Profile::iterative'). oclIsUndefined() implies self.base_ExpansionRegion.mode = ExpansionKind::iterative </pre>	«iterative»
<p>Description: An iterator, represented by the element ExpansionNode, must be unique per ExpansionRegion.</p> <p>OCL constraint:</p> <pre> context iterative inv: not self.base_ExpansionRegion. getAppliedStereotype('Profile::iterative'). oclIsUndefined() implies self.base_ExpansionRegion.inputElement->size()=1 </pre>	«iterative»
<p>Description: An Activity representing a legal policy must be annotated by the stereotype «policy».</p> <p>OCL constraint:</p> <pre> context context inv: not self.base_Activity. getAppliedStereotype('Profile::policy'). oclIsUndefined() </pre>	«policy»
<p>Description: ForkNodes, JoinNodes and MergeNodes cannot be used.</p> <p>OCL constraint:</p> <pre> context policy inv: ForkNode.allInstances()->size() = 0 and JoinNode.allInstances()->size() = 0 and MergeNode.allInstances()->size() = 0 </pre>	«policy»
<p>Description: ValuePins and DataStoreNodes cannot be used.</p> <p>OCL constraint:</p> <pre> context policy inv: DataStoreNode.allInstances()->size() = 0 and ValuePin.allInstances()->size() = 0 </pre>	«policy»
<p>Description: No object nodes other than ExpansionRegions and OpaqueActions can be used.</p>	«policy»

Consistency Constraints

Consistency constraint	Involved stereotype(s)
OCL constraint: context policy inv : Action. allInstances () ->select (a (not a. oclIsTypeOf (ExpansionRegion)) or (not a. oclIsTypeOf (OpaqueAction))) ->size ()=0	
Description: The activity diagram must be a Directed Acyclic Graph (DAG). Implementation: Preprocessing step for OCL transformation, implemented in Acceleo.	«policy»

Appendix B

Activity Diagrams to Text Transformations

B.1 Transforming Activity Diagrams to OCL: Detailed Algorithms

In this appendix, we provide a detailed version of our algorithm for transforming Activity Diagrams to OCL expressions, ADToOCL (Alg. 12), and a number of supporting algorithms (Algs. 7–10).

Main (Alg. 7) initializes the transformation’s inputs and makes the first call to ADToOCL. Algs. 3–6 are called from within ADToOCL. retrieveDependentInputs (Alg. 11) retrieves all inputs that need to be declared before transforming a given input pattern. getInitialNode (Alg. 9) returns the initial node of an Activity or an ExpansionRegion. getFlow (Alg. 8) returns the flow targeting the next UML element that should be visited. recognizePattern (Alg. 10) identifies and creates the appropriate pattern that matches the shape formed by the visited UML element, its stereotype, its container (Activity or ExpansionRegion) and its neighborhood.

Alg. 7: Transform Policy Models to Code (Main)

Inputs : An Activity Diagram, AD .

Output: A file containing the OCL constraint resulting from transforming AD .

```
1 if ( $AD.root$  is not  $NULL$ ) then
2   | Let  $inputs\_all$  be the set of inputs modeled in  $AD$ .
3   | Let  $inputs\_declared$  be the set of declared inputs in  $AD$  ( $inputs\_declared = \{\}$ ).
4   | Let  $file$  be the OCL output file of  $AD$ .
5   | write(ADToOCL( $AD$ ,  $AD.root$ ,  $inputs\_all$ ,  $inputs\_declared$ ,  $file$ ))
```

Alg. 8: Choose the Next Flow to Traverse (getFlow)

Inputs : (1) A pattern P .

Output: The outgoing flow from P (can be $NULL$).

```
1 if ( $P$  is an Intermediate Value Pattern) then
2   | return  $P.element.flow.target.flow->first()$ 
3 else return  $P.element.flow->first()$ 
```

Alg. 9: Get First Pattern to Transform into Code (getInitialNode)**Inputs :** (1) A UML element.**Output:** The internal initial node of element.

```

1 if ( element is an ExpansionRegion) then
2 |   return element.oclAsType(ExpansionRegion).owned->select(oclIsTypeOf(InitialNode))->first()
3 else
4 |   if ( element is an Activity) then
5 |     Return element.oclAsType(Activity).owned->select(oclIsTypeOf(InitialNode))->first()
6 |   else return NULL

```

Alg. 10: Identifying the Appropriate Pattern to Transform to Code (recognizePattern)**Inputs :** (1) An Activity Diagram, \mathcal{AD} . (2) An element $\in \mathcal{AD}$.**Output:** The appropriate pattern, P , corresponding to element.

```

1 switch (TypeOf(element)) do
2 |   case InitialNode: do return new Pattern(element, 'Initial Node Pattern')
3 |   case Activity Diagram Root: do return new Pattern(element, 'Context Pattern')
4 |   case Input: do
5 |     if (element has stereotype «fromagent») then
6 |     |   if (element.owner is an Activity) then
7 |     |   |   return new Pattern(element, 'From Agent Input To Activity Pattern')
8 |     |   if (element.owner is an ExpansionRegion) then
9 |     |   |   return new Pattern(element, 'From Agent Input To Expansion Pattern')
10 |    if (element has stereotype «fromlaw») then
11 |    |   if (element has an OCL expression) then
12 |    |   |   return new Pattern(element, 'From Law Variable Input Pattern')
13 |    |   else return new Pattern(element, 'From Law Constant Input Pattern')
14 |    if (element has stereotype «fromrecord») then
15 |    |   return new Pattern(element, 'From Record Input Pattern')
16 |   case OpaqueAction: do
17 |     if (element has stereotype «assert» and element.flows = {}) then
18 |     |   return new Pattern(element, 'Assert Pattern')
19 |     if (element has stereotype «action») then
20 |     |   if (element.owner is an Expansion Region With Output and element.flows = {}) then
21 |     |   |   return new Pattern(element, 'Action Inside an Expansion Region With Output Pattern')
22 |     |   if ((element.flows->size()) = 1 and (element.flows->first().target is a CentralBufferNode)) then
23 |     |   |   return new Pattern(element, 'Intermediate Value Pattern')
24 |   case DecisionNode: do
25 |     if (element.owner is an Activity) then
26 |     |   return new Pattern(element, 'Decision Node Pattern')
27 |     else return new Pattern(element, 'Decision Node Inside Expansion Region With Output Pattern')
28 |   case Final Flow: do
29 |     if (element.owner is an ExpansionRegion) then
30 |     |   return new Pattern(element, 'Final Flow Inside an Expansion Region With Output Pattern')
31 |   case ExpansionRegion: do
32 |     if (element has is an Output) then
33 |     |   return new Pattern(element, 'Expansion Region With Output Pattern')
34 |     else return new Pattern(element, 'Expansion Region Without Output Pattern')
35 |   Default: return NULL

```

Alg. 11: Get Dependent Variables to Declare (retrieveDependentInputs)

Inputs : (1) An Activity Diagram, \mathcal{AD} . (2) An element $\in \mathcal{AD}$.
(3) The set of \mathcal{AD} 's inputs, $inputs_all$. (4) The set of declared inputs, $inputs_declared$.
Output: The set of inputs that element depends on, $inputs_dependent$.

```

1   $inputs\_dependent \leftarrow \{\}$ 
2  if ( $element$  is  $NULL$ ) then
3  |   return  $inputs\_dependent$ 
4  Let  $inputs$  be the non-declared inputs required by element.
5  foreach  $input_i$  do
6  |    $inputs\_dependent \leftarrow inputs\_dependent \cup \text{retrieveDependentInputs}(\mathcal{AD}, input_i, inputs\_all, inputs\_declared \cup input_i)$ 
7  return  $inputs\_dependent$ 

```

Alg. 12: Detailed Algorithm for Transforming Activity Diagrams into OCL (ADToOCL)

Inputs : (1) An Activity Diagram, \mathcal{AD} . (2) An element $\in \mathcal{AD}$. (3) The set of \mathcal{AD} 's inputs, $inputs_all$. (4) The set of declared inputs, $inputs_declared$.

Output: An OCL string, result.

```



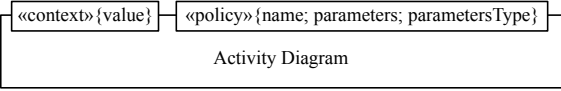
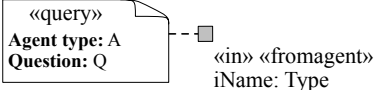
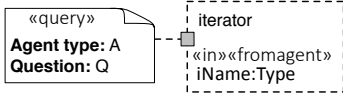
1  result  $\leftarrow ''$ 
2  if ( $element$  is  $NULL$ ) then return result
3  Let  $P$  be the pattern that will be applied to element.
4   $P \leftarrow \text{recognizePattern}(\mathcal{AD}, element)$ 
5  if ( $P$  is  $NULL$ ) then return result
6  Let  $inputs$  be the non-declared inputs required by  $P$ .
7  foreach  $input_i$  do
8  |   result  $\leftarrow$  result +  $\text{ADToOCL}(\mathcal{AD}, input_i, inputs\_all, inputs\_declared)$ 
9  |   Let  $inputs\_dependent$  a set of inputs required by  $input_i$  ( $inputs\_dependent \leftarrow \{\}$ ).
10 |    $inputs\_dependent \leftarrow inputs\_dependent \cup \text{retrieveDependentInputs}(\mathcal{AD}, input_i, inputs\_all, inputs\_declared)$ 
11 Let  $inputs\_declared\_save$  be a copy of  $inputs\_declared$ .
12  $inputs\_declared \leftarrow inputs\_declared \cup inputs \cup inputs\_dependent$ 
13 if ( $element$  is not a  $DecisionNode$ ) then
14 |   Let  $st_1$  be the opening OCL fragment obtained from applying  $P$  (See Appendix B.2).
15 |   Let  $st_2$  be the closing OCL fragment obtained from applying  $P$  (See Appendix B.2).
16 |   Let  $next$  be the next element that have to be visited.
17 |   if ( $element$  is an  $ExpansionRegion$  or  $element$  is an  $Activity$ ) then
18 |   |    $next \leftarrow \text{getInitialNode}(element)$ 
19 |   else  $next \leftarrow \text{getFlow}(P).target$ 
20 |   result  $\leftarrow$  result +  $st_1 + \text{ADToOCL}(\mathcal{AD}, next, inputs\_all, inputs\_declared) + st_2$ 
21 |   if ( $P$  is an  $ExpansionRegion$  With Output Pattern) then
22 |   |   Let  $out$  be the output element of  $P$ .
23 |   |   result  $\leftarrow$  result +  $\text{ADToOCL}(\mathcal{AD}, out, inputs\_all, inputs\_declared\_save)$ 
24 else
25 |   Let  $f_1, \dots, f_m$  be the outgoing flows from element.
26 |   if ( $m \geq 1$ ) then
27 |   |   foreach  $f_i$  do
28 |   |   |   if ( $i = 1$ ) then
29 |   |   |   |   result  $\leftarrow$  result + 'if (' + element.name + ') = ' +  $f_i.name$  + 'then' +
30 |   |   |   |   |    $\text{ADToOCL}(\mathcal{AD}, f_i.target, inputs\_all, inputs\_declared)$ 
31 |   |   |   else
32 |   |   |   |   result  $\leftarrow$  result + 'else if (' + element.name + ') = ' +  $f_i.name$  + 'then' +
33 |   |   |   |   |    $\text{ADToOCL}(\mathcal{AD}, f_i.target, inputs\_all, inputs\_declared)$ 
34 |   |   |   if ( $P$  is an  $DecisionNode$  Pattern) then
35 |   |   |   |   result  $\leftarrow$  result + 'else false' +  $\overbrace{\text{'endif' + \dots + \text{'endif'}}$   $m$  times
36 |   |   |   else
37 |   |   |   |   result  $\leftarrow$  result + 'else acc + 0' +  $\overbrace{\text{'endif' + \dots + \text{'endif'}}$   $m$  times
38 |   |   return result

```

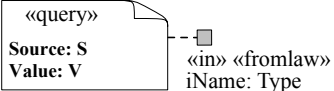
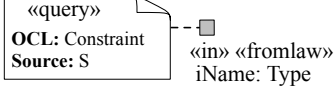
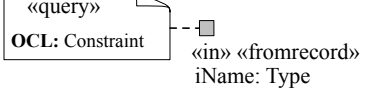
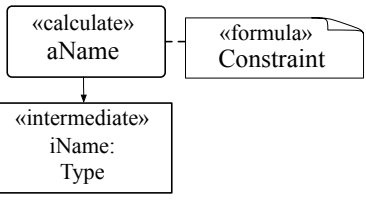
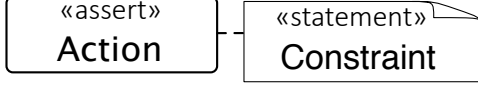
B.2 Patterns for Transforming Activity Diagrams to OCL

Table B.1 shows the patterns used by our transformation algorithm ADToOCL (Alg. 12) for generating the appropriate OCL expressions (L. 14-15 of Alg. 12). The first column of table B.1 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 OCL fragments for each pattern. These fragments correspond to the *opening* and *closing* expressions respectively denoted in ADToOCL by st_1 and st_2 (L. 14-15 of Alg. 12). The *opening* and *closing* expressions delimit respectively the beginning and the end of a branch or a loop. Patterns that do not result in the creation of a branch or loop do not require a *closing* expression.

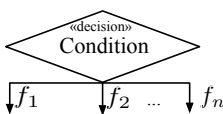
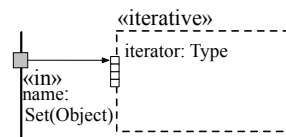
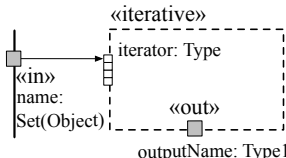
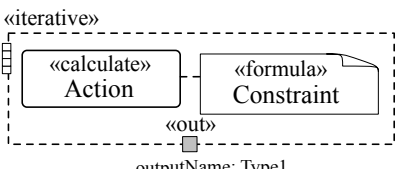
Table B.1. Patterns for Transforming Activity Diagrams to OCL

Pattern	OCL transformation
<p style="text-align: center;">Initial Node Pattern</p>  <p>Description: <i>Initial Node Pattern</i> is an <i>Elementary</i> pattern composed of an Initial Node.</p>	<p>opening OCL fragment: <code>' '</code></p> <p>closing OCL fragment: <code>' '</code></p>
<p style="text-align: center;">Final Node Pattern</p>  <p>Description: <i>Final Node Pattern</i> is an <i>Elementary</i> pattern composed of a FinalNode.</p>	<p>opening OCL fragment: <code>' true '</code></p> <p>closing OCL fragment: <code>' '</code></p>
<p style="text-align: center;">Policy Pattern</p>  <p>Description: <i>Context Pattern</i> is an <i>Elementary</i> pattern composed of an Activity having the <code><<policy>></code> and <code><<context>></code> stereotypes.</p>	<p>opening OCL fragment: <code>' context' + value + 'inv' + name + ' : '</code></p> <p>closing OCL fragment: <code>' '</code></p>
<p style="text-align: center;">FromAgent Input Pattern</p>  <p>Description: <i>FromAgent Input To Activity Pattern</i> is an <i>Aggregated</i> pattern composed of an input to an Activity having the <code><<in>></code> and <code><<fromagent>></code> stereotypes, and a comment with the <code><<query>></code> stereotype. This input obtains, from an agent, information that can be used inside its Activity.</p>	<p>opening OCL fragment: <code>' let' + iName + ' : ' + Type + ' = self.' + iName + ' / ** TRACEABILITY : ' + A + Q + ' ** / in '</code></p> <p>closing OCL fragment: <code>' '</code></p>
<p style="text-align: center;">FromAgent Input To Expansion Pattern</p>  <p>Description: <i>FromAgent Input To Expansion Pattern</i> is an <i>Aggregated</i> pattern composed of an input to an ExpansionRegion having the <code><<in>></code> and <code><<fromagent>></code> stereotypes, and a comment with the <code><<query>></code> stereotype. This input obtains, from an agent, information that can be used inside its ExpansionRegion.</p>	<p>opening OCL fragment: <code>' let' + iName + ' : ' + Type + ' = ' + iterator + ' . ' + iName + ' / ** TRACEABILITY : ' + A + Q + ' ** / in '</code></p> <p>closing OCL fragment: <code>' '</code></p>

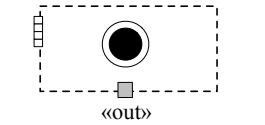
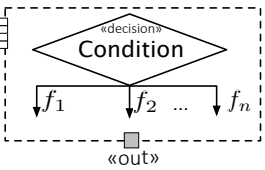
Patterns for Transforming Activity Diagrams to OCL

Pattern	OCL transformation
<p style="text-align: center;">FromLaw Constant Input Pattern</p>  <p>Description: <i>FromLaw Constant Input Pattern</i> is an <i>Aggregated</i> pattern composed of an input to an Activity or to an ExpansionRegion having the «in» and «fromlaw» stereotypes, and a comment with the «query» stereotype. This input obtains static information directly defined by law.</p>	<p>opening OCL fragment: <code>'let'+ iName +' :'+ Type +' = Constant ::'+ iName.toUpperCase() +' .oclAsType('+ Type +')' +' / ** TRACEABILITY :'+ S +' ** / in'</code></p> <p>closing OCL fragment: <code>''</code></p>
<p style="text-align: center;">FromLaw Variable Input Pattern</p>  <p>Description: <i>FromLaw Variable Input Pattern</i> is an <i>Aggregated</i> pattern composed of an input to an Activity or to an ExpansionRegion having the «in» and «fromlaw» stereotypes, and a comment with the «query» stereotype. This input captures concepts that are directly defined by law but may change depending on the domain objects.</p>	<p>opening OCL fragment: <code>'let'+ iName +' :'+ Type +' ='+ Constraint +' / ** TRACEABILITY :.+ S +' ** / in'</code></p> <p>closing OCL fragment: <code>''</code></p>
<p style="text-align: center;">FromRecord Input Pattern</p>  <p>Description: <i>FromRecord Input Pattern</i> is an <i>Aggregated</i> pattern composed of an input to an Activity or to an ExpansionRegion having the «in» and «fromlaw» stereotypes, and a comment with the «query» stereotype. This input obtains information from a record.</p>	<p>opening OCL fragment: <code>'let'+ iName +' :'+ Type +' ='+ Constraint +' in'</code></p> <p>closing OCL fragment: <code>''</code></p>
<p style="text-align: center;">Intermediate Value Pattern</p>  <p>Description: <i>Intermediate Value Pattern</i> <i>Aggregated</i> 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.</p>	<p>opening OCL fragment: <code>'let'+ iName +' :'+ Type +' ='+ Constraint +' in'</code></p> <p>closing OCL fragment: <code>''</code></p>
<p style="text-align: center;">Assert Pattern</p>  <p>Description: <i>Assert Pattern</i> is an <i>Aggregated</i> pattern composed of an OpaqueAction with the «assert» stereotype annotated by a Constraint having the «statement» stereotype. This pattern is used to verify if a given Constraint holds. Note that the Constraint must return a Boolean value.</p>	<p>opening OCL fragment: <code>Constraint</code></p> <p>closing OCL fragment: <code>''</code></p>

Patterns for Transforming Activity Diagrams to OCL

Pattern	OCL transformation
<p style="text-align: center;">Decision Node Pattern</p>  <p>Description: <i>Decision Node Pattern</i> is an <i>Elementary</i> pattern composed of a <i>DecisionNode</i> having the «decision» stereotype. The flows were explicitly included to the shape to enable understanding the generated OCL expression related to this pattern.</p>	<p>opening OCL fragment for $F_i = F_1$: <code>'if(' + Condition + ') = ' + f_i.name + 'then'</code></p> <p>opening OCL fragment for $F_i = F_2 \dots F_n$: <code>'else if(' + Condition + ') = ' + f_i.name + 'then'</code></p> <p>closing OCL fragment: <code>'else false' + ^{number of f} <code>'endif' + ... + 'endif'</code></code></p>
<p style="text-align: center;">Expansion Region Without Output Pattern</p>  <p>Description: <i>Expansion Region Without Output Pattern</i> is an <i>Aggregated</i> pattern composed of an input of type <i>Set</i> having the «in» stereotype and an <i>ExpansionRegion</i> having the «iterative» stereotype. The input is the collection over which the <i>ExpansionRegion</i> iterates. Note that the <i>ExpansionRegion</i> does not have an output; otherwise, the transformation will be different.</p>	<p>opening OCL fragment: <code>name + ' → forAll(' + iterator + ' : ' + Type + ' '</code></p> <p>closing OCL fragment: <code>)'</code></p>
<p style="text-align: center;">Expansion Region With Output Pattern</p>  <p>Description: <i>Expansion Region Without Output Pattern</i> is an <i>Aggregated</i> pattern composed of: (1) an input of type <i>Set</i> having the «in» stereotype, (2) an <i>ExpansionRegion</i> having the «iterative» stereotype, and (3) an output having the «out» stereotype. The input is the collection over which the <i>ExpansionRegion</i> performs a sum according to the UML elements contained inside it.</p>	<p>opening OCL fragment: <code>'let' + outputName + ' : ' + Type1 + ' = ' + name + ' → iterate(' + iterator + ' : ' + Type + ' ; acc : ' + Type1 + ' = 0 '</code></p> <p>closing OCL fragment: <code>) in'</code></p>
<p style="text-align: center;">Calculate Inside an Expansion Region With Output Pattern</p> 	<p>opening OCL fragment: <code>'acc + ' + Constraint</code></p> <p>closing OCL fragment: <code>''</code></p>



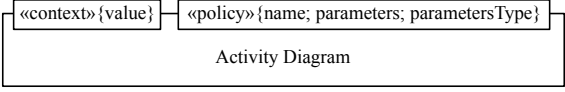
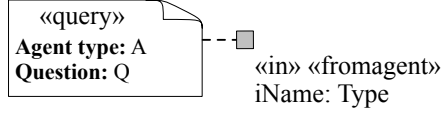
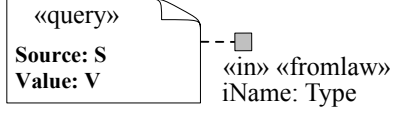
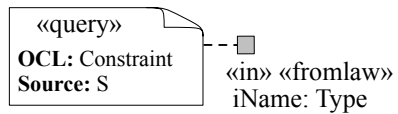
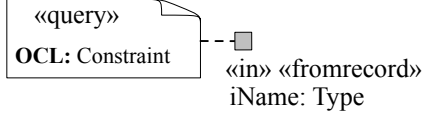
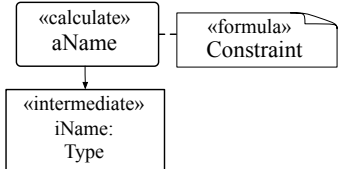
Patterns for Transforming Activity Diagrams to OCL

Pattern	OCL transformation
<p>Description: <i>Action Inside an Expansion Region With Output Pattern</i> is an <i>Aggregated</i> pattern composed of an <i>OpaqueAction</i> having the «calculate» stereotype. This <i>OpaqueAction</i> is annotated by a <i>Constraint</i> having the «formula» stereotype. Elements composing this pattern must be inside an <i>ExpansionRegion</i> having an output with the «out» stereotype. This pattern allows the formula of a <i>Constraint</i> to be considered in the calculation of the <i>ExpansionRegion</i>. Note that the latter OCL expression must have the same type as the output.</p>	
<p style="text-align: center;">Final Flow Inside an Expansion Region With Output Pattern</p>  <p><i>Final Flow Inside an Expansion Region With Output Pattern</i> is an <i>Aggregated</i> pattern composed of a <i>FinalFlowNode</i> contained inside an <i>ExpansionRegion</i> having an output with the «out» stereotype. This pattern indicates that the value calculated by the <i>ExpansionRegion</i> should not change.</p>	<p>opening OCL fragment: <code>'acc + 0'</code></p> <p>closing OCL fragment: <code>' '</code></p>
<p style="text-align: center;">Decision Node Inside Expansion Region With Output Pattern</p>  <p><i>Decision Node Inside Expansion Region With Output Pattern</i> is an <i>Aggregated</i> pattern composed of a <i>DecisionNode</i> having the «decision» stereotype. This <i>DecisionNode</i> is contained inside an <i>ExpansionRegion</i> having an output with the «out» stereotype. This pattern makes it possible to create branches inside an <i>ExpansionRegion</i>.</p>	<p>opening OCL fragment for $F_i = F_1$: <code>'if(' + Condition + ') = ' + f_i.name + 'then'</code></p> <p>opening OCL fragment for $F_i = F_2 \dots F_n$: <code>'else if(' + Condition + ') = ' + f_i.name + 'then'</code></p> <p>closing OCL fragment: <code>'else acc + 0' + $\underbrace{+ 'endif' + \dots + 'endif'}$ ^{number of f}</code></p>

B.3 Patterns for Transforming Activity Diagrams to (Java) Simulation Code

Table B.2 shows the patterns used by our transformation algorithm (Alg. 12 of Appendix B.1) for generating the appropriate Java simulation code. The first column of the table provides the name and the generic shape of each pattern. The description of patterns that are also used to generate OCL invariants can be found in Table B.1. The second column shows the resulting Java fragments for each pattern. These fragments correspond to the *opening* and *closing* expressions respectively denoted in ADToOCL by st_1 and st_2 (L. 14-15 of Alg. 12).

Table B.2. Patterns for Transforming Legal Policies to Java Simulation Code

Pattern	Generated JAVA Code Fragments
<p>Initial Node Pattern</p> 	<p>opening Java fragment: <code>''</code></p> <p>closing Java fragment: <code>''</code></p>
<p>Final Node Pattern</p> 	<p>opening Java fragment: <code>' return ;'</code></p> <p>closing Java fragment: <code>''</code></p>
<p>Policy Pattern</p> 	<p>opening Java fragment: <code>'public static void '+ name +'(' + forEach(parameters_i){parametersType_i +' '+ parameters_i} +') { OCLInJAVA.setContext(' + getParameterByContext(value) + '); String OCL="";'</code></p> <p>closing Java fragment: <code>'}'</code></p>
<p>FromAgent Input Pattern</p> 	<p>opening Java fragment: <code>' String OCL = "FromAgent. ' + toUpperCase(iName) + '";' + typeToJava(Type) +' ' + iName + ' = ' OCLInJava.eval' + getAbbreviationType(Type) +' (' + getObjectFromPM() +' , OCL);'</code></p> <p>closing Java fragment: <code>''</code></p>
<p>FromLaw Constant Input Pattern</p> 	<p>opening Java fragment: <code>typeToJava(Type) +' ' + iName + ' = ' V + ' ;' + ' /*TRACEABILITY: ' + S + ' * //'</code></p> <p>closing Java fragment: <code>''</code></p>
<p>FromLaw Variable Input Pattern</p> 	<p>opening Java fragment: <code>' String OCL = " ' + Constraint + '";' + typeToJava(Type) +' ' + iName + ' = ' OCLInJava.eval' + getAbbreviationType(Type) +' (' + getObjectFromPM() +' , OCL);'</code></p> <p>closing Java fragment: <code>''</code></p>
<p>FromRecord Input Pattern</p> 	<p>opening Java fragment: <code>' String OCL = " ' + Constraint + '";' + typeToJava(Type) +' ' + iName + ' = ' OCLInJava.eval' + getAbbreviationType(Type) +' (' + getObjectFromPM() +' , OCL);'</code></p> <p>closing Java fragment: <code>''</code></p>
<p>Intermediate Value Pattern</p> 	<p>opening Java fragment: <code>typeToJava(Type) +' ' + iName + ' = ' operatorsToJava(Constraint) + ' ;'</code></p> <p>closing Java fragment: <code>''</code></p>

Pattern	Generated JAVA Code Fragments
<p style="text-align: center;">Update Pattern</p> <div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <p style="text-align: center;">«update» {property; value} Action</p> </div> <p>Description: Update Pattern is an Elementary pattern composed of an OpaqueAction with the «update» stereotype.</p>	<p>opening Java fragment: If property is primitive: property + '=' + value + ';' ' If property is not primitive: ' OCLInJava.update(' + getPMContext().parameters₀ + ',' + value + ',' + property + ');' closing Java fragment: ' '</p>
<p style="text-align: center;">Assert Pattern</p> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 5px; margin: 5px;"> <p style="text-align: center;">«assert» Action</p> </div> <div style="border: 1px solid black; padding: 5px; margin: 5px;"> <p style="text-align: center;">«statement» Constraint</p> </div> </div>	<p>opening Java fragment: ' assert(' + operatorsToJava(Constraint) + '); ' closing Java fragment: ' '</p>
<p style="text-align: center;">Decision Node Pattern</p> <div style="text-align: center; margin: 10px auto;"> <div style="border: 1px solid black; padding: 5px; width: 100px;"> <p style="text-align: center;">«decision» Condition</p> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> ↓ f₁ ↓ f₂ ... ↓ f_n </div> </div>	<p>opening Java fragment for $F_i = F_1$: ' if(' + operatorsToJava(Condition) + ' == ' + operatorsToJava(f_i.name) + ') '{ opening Java fragment for $F_i = F_2 \dots F_n$: ' } else { if(' + operatorsToJava(Condition) + ') == ' + operatorsToJava(f_i.name) + ') '{ closing Java fragment: ' } else { System.err.println("Unhanded situation!"); } + ^{number of f-1} ' + ... + ' }</p>
<p style="text-align: center;">Expansion Region Without Output Pattern</p> <div style="margin: 10px auto;"> <div style="border: 1px dashed black; padding: 5px; width: 150px;"> <p style="text-align: center;">«iterative» iterator: Type</p> </div> <div style="margin-top: 10px;"> <p>«in» name: Set(Object)</p> </div> </div>	<p>opening Java fragment: ' for(EObject ' iterator + ':' + name + ') { OCLInJava.newIteration("'" + iterator + "','" + iterator + "','" + name + "','" + name + ');' closing Java fragment: ' } OCLInJAVA.iterationExit();'</p>
<p style="text-align: center;">Expansion Region With Output Pattern</p> <div style="margin: 10px auto;"> <div style="border: 1px dashed black; padding: 5px; width: 150px;"> <p style="text-align: center;">«iterative» iterator: Type</p> </div> <div style="margin-top: 10px;"> <p>«in» name: Set(Object)</p> </div> <div style="margin-top: 10px; text-align: center;"> <p>«out» outputName: Type1</p> </div> </div>	<p>opening Java fragment: typeToJava(Type1) + ' ' + outputName + '= 0; for(EObject ' iterator + ':' + name + ') { OCLInJava.newIteration("'" + iterator + "','" + iterator + "','" + name + "','" + name + ');' closing Java fragment: ' } OCLInJAVA.iterationExit();'</p>
<p style="text-align: center;">Calculate Inside an Expansion Region With Output Pattern</p> <div style="margin: 10px auto;"> <div style="border: 1px dashed black; padding: 5px; width: 150px;"> <p style="text-align: center;">«iterative»</p> <div style="display: flex; justify-content: space-around; align-items: center; margin: 5px 0;"> <div style="border: 1px solid black; padding: 5px; width: 60px;"> <p style="text-align: center;">«calculate» Action</p> </div> <div style="border: 1px solid black; padding: 5px; width: 60px;"> <p style="text-align: center;">«formula» Constraint</p> </div> </div> <p style="text-align: center;">«out» outputName: Type1</p> </div> </div>	<p>opening Java fragment: outputName + '=' + outputName + '+' + operatorsToJava(Constraint) closing Java fragment: ' '</p>
<p style="text-align: center;">Final Flow Inside an Expansion Region With Output Pattern</p> <div style="margin: 10px auto;"> <div style="border: 1px dashed black; padding: 5px; width: 100px; text-align: center;"> <p>«out»</p> </div> </div>	<p>opening Java fragment: ' ' closing Java fragment: ' '</p>

Appendix C

Consistency Constraints of the Profile for Expressing Probabilistic Information

Table C.1 shows the consistency constraints that check the sound application of our profile (the profile defining the probabilistic information of a simulation population) 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 C.1. Consistency Constraints (Profile for Expressing Probabilistic Characteristics)

Consistency constraint	Involved stereotypes
<p>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)</p> <p>OCL constraint:</p> <pre>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</pre>	<p>«from distribution» «from histogram» «from barchart» «fixed value»</p>
<p>Description: This constraint ensures that a context is specified when a stereotype from «probabilistic value»’s subtypes uses OCL.</p> <p>OCL constraint:</p> <pre>context probabilistic_value inv: self.usesOCL implies not self.context.oclIsUndefined()</pre>	<p>«from distribution» «from histogram» «from barchart» «fixed value»</p>

Consistency constraint	Involved stereotypes
<p>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.</p> <p>OCL constraint:</p> <pre> 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)) </pre>	<p>«from distribution»</p>
<p>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.</p>	<p>«from distribution»</p>

Consistency constraint	Involved stereotypes
<p>OCL constraint:</p> <pre> 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')) </pre>	

Consistency constraint	Involved stereotypes
<p>Description: This constraint ensures that all parameter values of a given distribution are numeric.</p> <p>OCL constraint:</p> <pre> context from_distribution inv: if (self.parameterValues ->notEmpty()) then self.parameterValues ->forall(s: String not s.toReal().oclIsUndefined()) else true endif </pre>	<p>«from distribution»</p>
<p>Description: This constraint ensures that the frequencies specified in a stereotype from «from chart»'s subtypes add-up to 1 (or to 100).</p> <p>OCL constraint:</p> <pre> 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 </pre>	<p>«from histogram» «from barchart»</p>
<p>Description: This constraint ensures that the frequencies specified for a stereotype from «from chart»'s subtypes are positive.</p> <p>OCL constraint:</p> <pre> context from_chart inv: if (self.frequencies ->notEmpty()) then self.frequencies -> forall(s: String s.toReal() >= 0) else true endif </pre>	<p>«from histogram» «from barchart»</p>
<p>Description: This constraint verifies that the number of frequencies in a «from histogram» is equal to the number of bins.</p> <p>OCL constraint:</p> <pre> context from_histogram inv: self.frequencies ->size() = self.bins ->size() </pre>	<p>«from histogram»</p>
<p>Description: This constraint verifies that the number of frequencies specified for a «from barchart» is equal to the number of items.</p> <p>OCL constraint:</p> <pre> context from_barchart inv: self.frequencies ->size() = self.items ->size() </pre>	<p>«from barchart»</p>
<p>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.</p>	<p>«multiplicity»</p>

Consistency constraint	Involved stereotypes
<p>OCLE constraint:</p> <pre> 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 </pre>	
<p>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.</p> <p>OCLE constraint:</p> <pre> 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 </pre>	«multiplicity»

Consistency constraint	Involved stereotypes
<p>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».</p> <p>OCL constraint:</p> <pre> 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 </pre>	«multiplicity»
<p>Description: This constraint ensures that the number of reuse probabilities specified in a «use existing» stereotype is equal to the number of OCL queries.</p> <p>OCL constraint:</p> <pre> context use_existing inv: self.reuseProbabilities ->size() = self.queries ->size() </pre>	«use existing»
<p>Description: This constraint ensures that the reuse probabilities specified for a «use existing» are positive.</p> <p>OCL constraint:</p> <pre> context use_existing inv: if (self.reuseProbabilities ->notEmpty()) then self.reuseProbabilities -> forall(r: Real r >= 0) else true endif </pre>	«use existing»
<p>Description: This constraint ensures that OCL queries specified using «OCL query» are not empty.</p> <p>OCL constraint:</p> <pre> context OCL_query inv: self.expressions ->forall(s : String s.specification.toString().trim().size() > 0) </pre>	«OCL query»
<p>Description: This constraint ensures that «type dependency» stereotype is not applied on primitive attributes.</p>	«type dependency»

Consistency constraint	Involved stereotypes
<p>OCLE constraint:</p> <pre> 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 </pre>	

Appendix D

Details of the Case Study for Generating Valid and Representative Data

In this appendix, we present: (1) the complete list of validity constraints used to express data validity (Appendix D.1), and (2) some examples of corrective constraints generated by our data generator (Appendix D.2).

D.1 OCL Validity Constraints

Table D.1 lists the OCL constraints that must be enforced by our data generator. The first column contains one or many OCL constraints alongside a brief description written in natural language. The second column indicates the source of the constraint(s). Our data generator gets the validity constraints from three different sources: 1) from users, 2) from the structure of the underlying domain model, i.e., multiplicity constraints, and 3) from annotations of the domain model, e.g., value ranges for attributes.

Table D.1. Complete List of the Input OCL Validity Constraints

OCL constraint	Source
<p>Description: checks that expenses declared by taxpayers are with a certain range (regulated). The amount for expenses should be between from €50 to not more than half of the taxpayer's income.</p> <pre>context Expense inv : let max1:Real = if ((self .income .income_amount / 2) >50) then self .income .income_amount / 2 else 50 endif in self .declared_amount >= 50 and self .declared_amount <= max1</pre>	User-defined (domain related)

OCL constraint	Source
<p>Description: checks that each taxpayer has at least one local income.</p> <pre> context Tax_Payer inv : self.incomes->select (i : Income i.oclIsTypeOf(Local_Income))->size () = 1 </pre>	User-defined (domain related)
<p>Description: ensure that legal unions (marriages or partnerships) are not composed by two non-taxpayer members.</p> <ol style="list-style-type: none"> <pre> context Legal_Union_Record inv : (self.individual_A.oclIsTypeOf(Physical_Person) and not self.individual_B.oclIsTypeOf(Physical_Person)) or (self.individual_B.oclIsTypeOf(Physical_Person) and not self.individual_A.oclIsTypeOf(Physical_Person)) or (not self.individual_A.oclIsTypeOf(Physical_Person) and not self.individual_B.oclIsTypeOf(Physical_Person)) </pre> <pre> context Legal_Union_Record inv : (self.individual_A.oclIsKindOf(Tax_Payer) and self.individual_B.oclIsTypeOf(Physical_Person)) or (self.individual_B.oclIsKindOf(Tax_Payer) and self.individual_A.oclIsTypeOf(Physical_Person)) or (self.individual_A.oclIsKindOf(Tax_Payer) and self.individual_B.oclIsKindOf(Tax_Payer)) </pre> 	User-defined (domain related)
<p>Description: enforces that each income detail refers to a different month (from 1 to 12).</p> <pre> context Income_Detail inv : let val : Integer = self.income.details->asOrderedSet()->indexOf(self) in self.month = val </pre>	User-defined (common sense)
<p>Description: checks the correctness of the declared address for non-resident taxpayers.</p> <pre> context Tax_Payer inv : self.addresses->exists (fiscal_add : Address fiscal_add.oclIsTypeOf(Fiscal_Address) and fiscal_add.country <> TaxCard::Country::LU) and self.addresses->exists (hab_add : Address hab_add.oclIsTypeOf(Habitual_Address) and hab_add.country <> TaxCard::Country::LU) and self.incomes->exists (inc : TaxCard::Income inc.oclIsTypeOf(TaxCard::Local_Income)) implies self.oclIsTypeOf(Non_Resident_Tax_Payer) </pre>	User-defined (domain related)

OCL constraint	Source
<p>Description: checks the correctness of the declared address for resident taxpayers.</p> <pre> context Tax_Payer inv: self.addresses ->exists(fiscal_add:Address fiscal_add.oclIsTypeOf(Fiscal_Address) and fiscal_add.country = TaxCard::Country::LU) or self.addresses ->exists(hab_add:Address hab_add.oclIsTypeOf(Habitual_Address) and hab_add.country = TaxCard::Country::LU) implies self.oclIsTypeOf(Resident_Tax_Payer) </pre>	<p>User-defined (domain related)</p>
<p>Description: restricts the possible tax property, e.g., income-splitting, declarations over time. Here, users are only interested in generating the tax property for a single year.</p> <pre> context Legal_Union_Record inv: self.properties ->size() = 1 </pre>	<p>User-defined (simplification)</p>
<p>Description: define the persons who can be widowers. For example, taxpayers who never got married, cannot be widowers.</p> <pre> 1. context Tax_Payer inv: (self.getSpouse(1900).name.oclIsInvalid()) or ((not self.getSpouse(1900).name.oclIsInvalid()) and not self.is_widower) 2. context Physical_Person inv: ((self.oclIsTypeOf(Dependent) or self.oclIsTypeOf(Physical_Person)) and not self.is_widower) or ((not self.oclIsTypeOf(Dependent) and not self.oclIsTypeOf(Physical_Person) and (self.is_widower or not self.is_widower))) </pre>	<p>User-defined (common sense)</p>
<p>Description: ensures that the starting year of a given tax property, e.g., income splitting, is less than the end year of the property.</p> <pre> context Tax_Property inv: self.starting_year >= self.union_record.start_year </pre>	<p>User-defined (simplification)</p>
<p>Description: checks that income amounts are inline with their monthly details.</p> <pre> context Income inv: let value:Real = if(self.details ->size() = 0) then -1 else self.details ->any(true).amount endif in self.income_amount = value </pre>	<p>User-defined (common sense)</p>

OCL constraint	Source
<p>Description: enforce that income amounts (as well as their details) are positive or undefined (-1).</p> <ol style="list-style-type: none"> context Income inv: <code>self.income_amount > 0 or self.income_amount = -1</code> context Income_Detail inv: <code>self.amount > 0 or self.amount = -1</code> 	User-defined (common sense)
<p>Description: ensure that parents are older (enough) than their children.</p> <ol style="list-style-type: none"> context Physical_Person inv: <code>let val:Integer = self.birth_year + 16 in self.dependents->forall(d:Dependent d.birth_year > val)</code> context Tax_Payer inv: <code>let children11:Set(Dependent)=self.dependents in let legel_unions:Set(Legal_Union_Record) = Legal_Union_Record.allInstances()->select(((individual_A = self or individual_B = self))) in let possible_unions:Set(Legal_Union_Record) = legel_unions->select(start_year <= 2017) in let lasted_union:Legal_Union_Record= possible_unions->select(start_year=possible_unions.start_year ->max())->any(true) in let house1:Household = lasted_union.household in let children1:Set(Dependent) =if(house1.ocIsUndefined()) then children11 else children11->union(house1.children) endif in let maxAge1:Integer = if(children1->size() = 0) then 2000 else children1.birth_year->max() - 17 endif in self.birth_year < maxAge1 and self.birth_year > 1900</code> context Dependent inv: <code>let house:Household = Household.allInstances()->select(h:Household h.children->select(c:Dependent c = self)->size() > 0)->any(true) in let val1:Integer = if(not house.parents.ocIsInvalid()) then house.parents.individual_A.birth_year + 16 else self.birth_year - 20 endif in let val2:Integer = if(not house.parents.ocIsInvalid()) then house.parents.individual_B.birth_year + 16 else self.birth_year - 20 endif in self.birth_year > val1 and self.birth_year > val2</code> 	User-defined (common sense)

OCL constraint	Source
<p>Description: checks that the amount of a given allowance matches the calculations stipulated in the law.</p> <pre> context External_Allowance inv: let origin:Physical_Person = self.reciver in let children1:Set(Dependent) = if(origin. oclIsKindOf(Tax_Payer)) then origin. oclAsType(Tax_Payer).dependents ->select(allowances->size() > 0) else Set\{\} endif in let union:Legal_Union_Record = origin. getLegalUnionRecord(2017) in let house:Household = union.household in let children:Set(Dependent) =if(house. oclIsUndefined()) then children1 else children1->union(house.children->select(allowances->size() > 0)->select(allowances->any(true).reciver = origin)) endif in let eligible_children:Set(Dependent) = children->select(birth_year < 2017) in let is_disabled:Boolean = self. person.disability_type <> Disability_Types::NONE and self.person. disability_percentage > 0.5 in let age:Integer = 2017 - self.person.birth_year in (self.person.birth_year >= 2017 and self.amount = 580) or (self.person.birth_year < 2017 and (eligible_children->size() = 1 or eligible_children->size() = 0) and self.amount = self.getAmount(185.60, age, is_disabled)) or (self.person.birth_year < 2017 and eligible_children->size() = 2 and self.amount = self.getAmount(220.36, age, is_disabled)) or (self.person.birth_year < 2017 and eligible_children->size() = 3 and self.amount = self.getAmount(267.59, age, is_disabled)) or (self.person.birth_year < 2017 and eligible_children->size() > 3 and self.amount = self.getAmount(361.83, age, is_disabled)) </pre>	<p>User-defined (domain related)</p>

OCL constraint	Source
<p>Description: restricts the type of incomes based on the taxpayers' age. For example, taxpayers aged more than 60 should have a pension.</p> <pre> context Tax_Payer inv: (self.birth_year < 1958 and self.incomes->any(true) .income_type.oclIsKindOf(Pensions_and_Annuities_Income)) or (self.birth_year > 1961 and not self.incomes->any(true) .income_type.oclIsKindOf(Pensions_and_Annuities_Income)) or (self.birth_year >= 1958 and self.birth_year <= 1961) </pre>	User-defined (common sense)
<p>Description: verifies that only pensioners and employees have tax-cards.</p> <pre> context Income inv: (not self.income_type.oclIsKindOf(Employment_Income) and not self.income_type.oclIsTypeOf(Pensions_and_Annuities_Income) and self.tax_card.oclIsUndefined()) or (self.income_type.oclIsKindOf(Employment_Income) and not self.tax_card.oclIsUndefined()) or (self.income_type.oclIsTypeOf(Pensions_and_Annuities_Income) and not self.tax_card.oclIsUndefined()) </pre>	User-defined (domain related)
<p>Description: checks the soundness of the starting year of social allowances. For example, a parent cannot receive an allowance for a child that is not born yet.</p> <pre> context External_Allowance inv: (self.person.birth_year >= 2017 and self.starting_year = 2017) or (self.person.birth_year < 2017 and (self.starting_year >= self.person.birth_year + 2) and self.starting_year <= 2017) </pre>	User-defined (domain related)
<p>Description: defines the dependents for whom a taxpayer might get an allowance.</p> <pre> context Dependent inv: (self.birth_year >= 2017 and self.allowances->size() = 1) or (self.birth_year < 2017 and self.birth_year >= 2000 and self.allowances->size() = 1) or (self.birth_year >= 1994 and self.continued_studies and self.allowances->size() = 1) or (self.birth_year < 2000 and not self.continued_studies and self.allowances->size() = 0) </pre>	User-defined (domain related)

OCL constraint	Source
<p>Description: checks that the information of legal separations is properly sound.</p> <pre> context Legal_Union_Record inv: (self.separation_cause = Separation_Causes::NONE and self.end_year = -1) or (self.separation_cause <> Separation_Causes::NONE and self.end_year >= self.start_year and self.end_year <= 2017) </pre>	User-defined (common sense)
<p>Description: checks that disability related information does not contain logical anomalies.</p> <pre> context Physical_Person inv: (self.disability_type = Disability_Types::NONE and self.disability_percentage = 0) or (self.disability_type <> Disability_Types::NONE and self.disability_percentage > 0 and self.disability_percentage <= 1) </pre>	User-defined (common sense)
<p>Description: defines the taxpayers who are eligible to pay pension contributions.</p> <pre> context Income_Detail inv: (self.income.income_type.oclIsTypeOf(Pensions_and_Annuities_Income) and not self.is_contributing_pension) or (not self.income.income_type.oclIsTypeOf(Pensions_and_Annuities_Income) and self.is_contributing_pension) </pre>	User-defined (domain related)
<p>Description: ensures that the declared commutation distance is constant for a same taxation year.</p> <pre> context Income inv: let val:Real = self.details ->any(true).distance in self.details ->forAll(d:Income_Detail d.distance = val) </pre>	User-defined (domain related)

OCL constraint	Source
<p>Description: represent the multiplicity constraints that are explicitly specified in OCL.</p> <ol style="list-style-type: none"> 1. context Tax_Payer inv: self.incomes->size() >= 1 2. context Tax_Payer inv: not self.from_agent.oclIsUndefined() 3. context Tax_Payer inv: not self.from_law.oclIsUndefined() 4. context External_Allowance inv: not self.reciver.oclIsUndefined() 5. context Income inv: not self.income_type.oclIsUndefined() 6. context Income inv: not self.taxPayer.oclIsUndefined() 7. context Income inv: self.details->size() = 12 8. context Income_Type inv: not self.income.oclIsUndefined() 9. context Legal_Union_Record inv: not self.household.oclIsUndefined() 10. context Legal_Union_Record inv: not self.individual_A.oclIsUndefined() 11. context Legal_Union_Record inv: not self.individual_B.oclIsUndefined() 12. context Household inv: not self.parents.oclIsUndefined() 	<p>Extracted from the class diagram</p>

OCL constraint	Source
<p>Description: validity constraints extracted from the annotations of the class diagram.</p> <ol style="list-style-type: none"> 1. context Tax_Payer inv: self.AEP_deduction = 0 2. context Physical_Person inv: self.birth_month >= 1 and self.birth_month <= 12 3. context Physical_Person inv: self.birth_day >= 1 and self.birth_day <= 28 5. context Tax_Payer inv: self.dependents->size() = 0 or self.dependents->size() = 1 or self.dependents->size() = 2 or (self.dependents->size() >= 3 and self.dependents->size() <= 4) 6. context External_Allowance inv: self.amount >= 0 7. context External_Allowance inv: self.ending_year = -1 8. context FromAgent inv: self.taxation_year = 2017 9. context Income inv: self.year = 2017 10. context Income inv: self.start_year = 2017 11. context Tax_Card inv: self.deduction_FD_yearly = 0 12. context Tax_Card inv: self.credit_CIS_yearly = 0 13. context Tax_Card inv: self.credit_CIS_monthly = 0 14. context Tax_Card inv: self.credit_CIP_yearly = 0 15. context Tax_Card inv: self.credit_CIP_monthly = 0 16. context Tax_Card inv: self.deduction_CE_invalidity_yearly = 0 17. context Tax_Card inv: self.deduction_DS_Debt_yearly = 0 18. context Income_Detail inv: self.is_contributing_CNS = true 	<p>Extracted from the class diagram's annotations</p>

OCL constraint	Source
<p>Description: validity constraints extracted from the annotations of the class diagram.</p> <ol style="list-style-type: none"> 1. context Physical_Person inv: <code>self.last_start_year_widower = -1</code> 2. context Physical_Person inv: <code>self.name = 'Not important'</code> 3. context Expense inv: <code>self.year_expense_was_incurred_in = 2017</code> 4. context Household inv: <code>self.children->size() = 0 or self.children->size() = 1 or (self.children->size() >= 2 and self.children->size() <= 10)</code> 5. context Legal_Union_Record inv: <code>self.start_year >= 1900 and self.start_year <= 2017</code> 6. context Legal_Union_Record inv: <code>self.properties->size() >= 1</code> 7. context Tax_Property inv: <code>self.taxed_jointly = true</code> 8. context Tax_Property inv: <code>self.starting_year >= self.union_record.start_year and self.starting_year <= 2017</code> 9. context Income_Detail inv: <code>self.is_worked = true</code> 10. context Income_Detail inv: <code>(not self.income.income_type.ocIsTypeOf(Employment_Income) and self.worked_days = 0) or (self.income.income_type.ocIsTypeOf(Employment_Income)=true and self.worked_days >= 10 and self.worked_days <= 25)</code> 11. context Income_Detail inv: <code>(not self.income.income_type.ocIsTypeOf(Employment_Income) and self.distance = 0) or (self.income.income_type.ocIsTypeOf(Employment_Income) = true and ((self.distance >= 0 and self.distance <= 4) or (self.distance >= 5 and self.distance <= 9) or (self.distance >= 10 and self.distance <= 30) or (self.distance >= 30 and self.distance <= 100)))</code> 	

OCL constraint	Source
<p>Description: validity constraints extracted from the annotations of the class diagram.</p> <p>1. context Income_Detail inv : (self.amount >= 0 and self.amount <= 833) or (self.amount >= 834 and self.amount <= 1666) or (self.amount >= 1667 and self.amount <= 2500) or (self.amount >= 2501 and self.amount <= 3333) or (self.amount >= 3334 and self.amount <= 4166) or (self.amount >= 4167 and self.amount <= 5000) or (self.amount >= 5001 and self.amount <= 5833) or (self.amount >= 5834 and self.amount <= 6666) or (self.amount >= 6667 and self.amount <= 7500) or (self.amount >= 7501 and self.amount <= 8333) or (self.amount >= 8334 and self.amount <= 9166) or (self.amount >= 9167 and self.amount <= 10000) or (self.amount >= 10001 and self.amount <= 10833) or (self.amount >= 10834 and self.amount <= 11666) or (self.amount >= 11667 and self.amount <= 12500) or (self.amount >= 12501 and self.amount <= 13333) or (self.amount >= 13334 and self.amount <= 14166) or (self.amount >= 14167 and self.amount <= 15000) or (self.amount >= 15001 and self.amount <= 15833) or (self.amount >= 15834 and self.amount <= 16666) or (self.amount >= 16667 and self.amount <= 20833) or (self.amount >= 20834 and self.amount <= 41666) or (self.amount >= 41667 and self.amount <= 50000) or (self.amount >= 50001 and self.amount <= 58333) or (self.amount >= 58333 and self.amount <= 83333) or (self.amount >= 83333 and self.amount <= 333333)</p> <p>2. context Physical_Person inv : ((self.oclIsTypeOf(Dependent)=true) and ((self.birth_year >= 2010 and self.birth_year <= 2017) or (self.birth_year >= 2005 and self.birth_year <= 2009) or (self.birth_year >= 2000 and self.birth_year <= 2004) or (self.birth_year >= 1995 and self.birth_year <= 1999) or (self.birth_year >= 1990 and self.birth_year <= 1994) or (self.birth_year >= 1985 and self.birth_year <= 1989)) or ((not self.oclIsTypeOf(Dependent)) and ((self.birth_year >= 1995 and self.birth_year <= 1999) or (self.birth_year >= 1990 and self.birth_year <= 1994) or (self.birth_year >= 1985 and self.birth_year <= 1989) or (self.birth_year >= 1980 and self.birth_year <= 1984) or (self.birth_year >= 1975 and self.birth_year <= 1979) or (self.birth_year >= 1970 and self.birth_year <= 1974) or (self.birth_year >= 1965 and self.birth_year <= 1969) or (self.birth_year >= 1960 and self.birth_year <= 1964) or (self.birth_year >= 1955 and self.birth_year <= 1959) or (self.birth_year >= 1950 and self.birth_year <= 1954) or (self.birth_year >= 1945 and self.birth_year <= 1949) or (self.birth_year >= 1940 and self.birth_year <= 1944) or (self.birth_year >= 1935 and self.birth_year <= 1939) or (self.birth_year >= 1930 and self.birth_year <= 1934))))</p>	

D.2 Example of Dynamically Generated Corrective Constraints

In this appendix, we provide three examples of corrective constraints created during data generation. The first example was generated from the *birth_year* attribute of our class diagram. The second example was derived from the household-children association whereas the last example comes from a generalization that denotes the possible types for of cases.

D.2.1 Example of a Corrective Constraint Derived from an Attribute (*birth_year*)

```

context Tax_Payer inv add1:
(((Tax_Payer.allInstances()->select(SSNo = '1')->
forall(not(birth_year >= 1979 and birth_year <= 1998)))and
(Tax_Payer.allInstances()->select(SSNo = '1')->
forall(not(birth_year >= 1959 and birth_year <= 1978))))and
(Tax_Payer.allInstances()->select(SSNo = '1')->
forall(birth_year >= 1900 and birth_year <= 1933)))
or
((if(self.incomes->notEmpty()) then
  self.incomes->exists(oclIsTypeOf(
    Pensions_and_Annuities_Income)) else false endif)
and
((Tax_Payer.allInstances()->select(SSNo = '1')->
forall(birth_year >= 1957 and birth_year <= 1960))and
(Tax_Payer.allInstances()->select(SSNo = '1')->
forall(not(birth_year >= 1917 and birth_year <= 1956))))))

```

D.2.2 Example of a Corrective Constraint Derived from an Association (*household-children*)

```

context Household inv add2:
(Household.allInstances()->select(id1 =
  '16841')->forall(children->size() <> 0)
and
Household.allInstances()->select(id1 =
  '16841')->forall((children->size() = 1) or (children->size()
  >= 2 and children->size() <= 4)))

```

D.2.3 Example of a Corrective Constraint Derived from a Generalization (Types of Tax Cases)

```

context Tax_Case inv add3:
(((not Tax_Case()->select(id1 =
  '17862')->any(true).oclIsTypeOf(Household)) and
(not Tax_Case()->select(id1 =
  '17862')->any(true).oclIsTypeOf(Non_Resident_Tax_Payer)))
and
(Tax_Case()->select(id1 =
  '17862')->any(true).oclIsTypeOf(Resident_Tax_Payer)))

```