The Specification Problem of Legal Expert Systems

Denis Merigoux

The Specification Problem of Legal Expert Systems

DENIS MERIGOUX, Inria, France

1 INTRODUCTION

Automated legal decision-making relies on computer programs called legal expert systems, that are executed on machines not capable of legal reasoning by themselves. Rather, it is up to the programmer to ensure that the behavior of the computer program faithfully captures the letter and intent of the law. This situation is merely an instance of the more general *specification problem* of computer science. Indeed, the way programs are written and executed requires the programmer to express her intention in a particular form of logic or statistical model imposed by the programming language or framework. On the other hand, the intended behavior of the program or *specification*, here communicated through the law, is usually described using natural language or domain-specific insights. Hence, every software endeavor begins with a requirement analysis, which consists in extracting from the specification corpus a set of requirements that the computer system must obey.

In the case of automated legal decision-making and legal expert systems, the members of this set of requirements are the possible legal reasoning bits that the computer program is expected to perform. Viewing the problem through this lens immediately allows for identifying the key questions for assessing the safety and correctness of legal expert systems. First, when and how is it possible to express legal reasoning as a set of requirements for a computer system? Second, how to check that these requirements are correctly translated to computer code? Third, can we ensure that the computer code does not introduced unwanted, unlawful behavior? In this article, we take a tour of the general computer science answers to these three questions and assess their efficiency in the particular situation of legal expert systems. To do so, we introduce the distinction between result-constrained and process-constrained legal specifications. From this distinction naturally stem different software solutions, ranging from machine-learning-based to algorithm-based. Finally, we conclude by a discussion about the *critical software* qualification for legal expert systems, and what this qualification could entail in terms of technical and organizational change.

2 LAW AS SPECIFICATIONS

Before writing a computer program, the programmer must first know what this program is supposed to do. This simple statement underpins one of the most difficult challenges in computer science, which arises not with algorithmic complexity but rather by the imprecision and the vagueness of the intent leading to writing the program. In that regard, the issues of legal expert systems fall in the broader category of business-logic-oriented systems. Business rules can be found in contracts that, like legal texts, are documents that follow a drafting process where the goal is not to produce an artefact feedable to a machine, but rather to settle down a social, political or business compromise in terms that all parties can agree and refer to. Because of that goal mismatch, business rules and legal texts are by construction *informal*, in contrast to the *formality* of executable computer programs. Indeed, a computer is can oly perform additions, multiplications and other kinds of basic operations. It is up to the programmer to give a higher-level meaning to the numbers being crunched, set in stone in the program. But the program and its semantics cannot suffer any ambiguity since they merely describe in a high-level way the sequence of basic operations executed by the machine. This

Author's address: Denis Merigoux, denis.merigoux@inria.fr, Inria, Paris, France.

rigidity and lack of adaptation to ambiguity leads to the "ruleishness" of computer code described by Diver [2021].

Behind programs, there usually exist textual documents that act as a reference to their behavior. These documents, which can be legal texts or business contrats, are called *specifications*, and the informality of specifications has been on the computer's scientists mind for a long time (see for example [Balzer et al. 1978; Furtado 1983; Le Charlier and Flener 1998; Rich and Waters 1986]). Indeed, how to reconcile the informality of the specifications, that lead to ambiguities, to the rigidity of the computer programs that cannot accept ambiguities? A wide range of solutions have been devised to partially solve this problem, and I claim that they fully apply to the case where specifications are legal texts. However, the solutions depend on what the legal text is trying to specify. The categories that we will make here are not based on legal concepts but rather on the mathematical meaning of the intent of a piece of law.

First, some legal specifications describe *constructively* the object of their regulation. These specifications assume some data as given by the user of to be determined by observations, and then define a sequence of computation-like steps that operate on these inputs and intermediate quantities, eventually yielding a result. This result can be then used as the content of a decision affecting a legal persona. The part of tax law that computes the amount of taxes owed after having filled the income declaration is an archetypal example of such a constructive legal specification that we can also dub *process-constrained*, because the law encodes the process used to reach the result. This type of specification is the more easily translatable into traditional, rule-based computer programs, because these are by essence constructive.

Second, other legal specifications describe *existentially* the object of their regulation. These specifications assume that their users will come up with their own way to produce an instance of the object being regulated; the specification only cares about the final characteristics of the object. For instance, the decision of whether to admit or not a student into a typical anglo-saxon university, or to hire an employee in a company, is existentially regulated; universities and companies are free to make up their own internal, constructive selection process, the law only cares about properties of the final result, *e.g* whether it discriminates against legally protected groups. Hence the name of *result-constrained* legal specifications. However, one can note that a result-constrained specification can embed a little bit of process constraints when defining how the desired properties of the result should be checked.

Those two categories of legal specifications both suffer from the informality mentioned earlier. This informality can manifest itself under several forms of increasing gravity. Firstly, syntactic ambiguities concern the precise drafting of the law in natural language that sometimes leave multiple possibilities for parsing the contents of a sentence (*e.g* multiple and/or clauses with improper punctuation). These syntactic problems were already identified by Allen [1956], and it is relatively simple to address them. Secondly, structural ambiguities also concern the drafting of the legal document but in a non-local way. For instance, multiple articles can define conflicting amounts for a same quantity, or leave unexplained in which order they should be applied during the computation. This kind of ambiguities has been masterfully emphasized by Lawsky [2017] in case studies about the US Tax Code; however, it is also possible to fix them by merely precising the intent of the text of the law. Lastly, conceptual ambiguities raise a bigger problem. Indeed, law often leaves purposefully ambiguous the definitions of some of its terms, leaving their interpretation to the user of the law. These interpretations can be seen as a leeway to fit in the user's own interest.

But, the interpretation of the law encoded in a legal expert system is inherently restrictive; in a non-automatized setting, the defendant can chose her own interpretation of the law, and uphold it in court until a judge settles it. But when using a legal expert system, the user is placed in the position of a defendant that is bound to use only the legal interpretations of the prosecutor, as these are set inside the non-modifiable source code of the system. Contrary to the trial process where each party can argue according to their interest, a legal expert system only reflects the views and interest of its designer. In this sense, a legal expert system is simply a *projection* (in the mathematical acceptation of the term) of a piece of legislation into a interpretation plane.

Of course, this projection can be more or less authoritative depending on the way the source code of the legal expert system is written. For instance, a legal expert system can leave as a user input the choice between multiple interpretations of a single legislative item. But enriching legal expert systems with more and more interpretations that could benefit users is costly, and it is often impossible to foresee all of these interpretations in advance – more discussion on that later. To remedy this flaw, one could imagine a system where the user could bring its own interpretations of the law in the form of pluggable source code that would extend or modify the legal expert system. This idea is somewhat similar the XAlgorithms alliance and its notion of *oughtomation* [Potvin et al. 2021], that it aims to apply to international trade law – although only trusted parties could bring rules to the table, and not regular users. But such a system has to somehow consider the interpretations of the legislation as data, which prevents it from encoding them in the source code directly. Even so, the informality problem of the legislative specification is still present, in a decentralized form.

For the rest of this article, we will focus on traditional, authoritative legal expert systems as they remain the real-world standard of automatic law enforcement, used for instance by many government agencies [Blank and Osofsky 2020; Escher and Banovic 2020; JetBrain MPS 2019; Merigoux et al. 2021b] or companies [Connell 1987; SMU Centre for Computational Law 2020; Voelter et al. 2021]. Specifically, we will now present the computer science solutions to the problem of transcribing the informal specifications of these systems into executable code.

3 CHECKING FOR FAITHFULNESS TO THE LEGISLATIVE SPECIFICATION

In this section and in the rest of the paper, we will consider a legal expert system implemented as a codebase in a given programming language. The specification of the legal expert system consists of various legal texts that can come from law, official regulations or internal bulletins. The legal expert system expects at runtime a certain number of inputs provided by the user, and automatically outputs its decision. The decision should be faithful to what would have happened if a lawyer would have manually applied legal reasoning to the case described by the input of the legal expert system.

The methods for validating the legal expert system depend on whether it is result-constrained or process-constrained. Respectively, these two categories lead to validating the system as a black box, or a white box.

3.1 Result-constrained specifications

When the legal specification only characterizes the properties that the output of the legal expert system should have, the actual implementation of how the system actually comes up with its answers is irrelevant. This is the black box scenario, which comes with a loose and dynamic form of correctness checking. Indeed, the most basic and effective method of checking whether such a

system is faithful to the specification is simply to check at runtime that all its outputs individually respect the contraints set out by the specification.

Unfortunately, this basic method comes with serious drawbacks. First, what happens when the dynamic checking invalidates a result returned by the black box system? Unless the black-box system can learn online from this rejection and try to formulate on the spot a different result that will satisfy the constraints, such a rejection will trigger a review of the case, by a different computerized system or by a human operator. A too high rate of rejection can lead to serious efficiency problems for the system. In the case of a statistical learning system, a re-training of the model with better data or constraints would be needed to decrease the rejection rate, which is a relatively costly operation.

Second drawback of the individual correctness checking for result-constrained legal specification: it is often not possible to to check that each output *individually* respects the constraints, for instance because the constraints relate to properties of *all* the outputs of the legal expert system. These global properties can consist of statistical constraints that express for instance the absence of bias in the output. Hence, the validation has to operate on a log of all the decisions taken by the legal expert system, a log which then can be queried for statistical or other properties like in the system proposed by Henin and Le Métayer [2019]. For systems based on statistical learning, this topic intersects with active research areas around "explanable AI" or statistical fairness, represented since 2018 at the ACM Conference on Fairness, Accountability, and Transparency (FaccT). The lack of accountability on the result of statistical learning systems has for instance lead the French constitutional court to disallow such systems for administrative decisions [Conseil Constitutionnel 2018].

Therefore, it is often desirable to build the legal expert system in a way that satisfies the result constraints by design, moving from *ex post* validation to *ex ante*. For instance, the Parcoursup French higher education admission system is result-constrained, with a property stating that each university should admit at least a certain percentage of students with state-funded scholarships [Légifrance 2018]. Rather than checking *ex post* if the matching between universities and students proposed by Parcoursup validates that constraint, Parcoursup's internal design incorporate this constraint with a relatively complex algorithm whose description is public [Gimbert 2021] and has been drafted by a computer science researcher. Other researchers then tried to formally connect the actual code that implements the algorithm to the high-level description of the algorithm [Becker et al. 2020], but did not manage to complete the task due to technical difficulties. Hotly debated in France and under a lot of scrutiny, Parcoursup can still be considered as a model for the particular aspect of checking the faithfulness of this computer system with respect to legal constraints on its results. However, the technical complexity of such a checking operation remains very high, and more importantly it involves actively designing the system to comply with the constraints, moving from a black box model to a white box model.

This last Parcoursup example is conceptually important because it shows how constraints on the result of a computer system actually translate to constraints on the process of the system, *i.e.* its implementation. We will come back to the consequences of this observation in Section 4; but first, let us introduce the common methods for checking correctness of a computer system with respect to a process-based specification.

3.2 Process-based specification

Because of their constructive nature, process-based specifications have to describe precisely how to get the desired result. In a sense, these specifications describe a computation that could be

performed by a machine given a set of user inputs. But because of their informal nature and the ambiguities detailed in Section 2, the translation from the legal specification to executable code is not always straightforward. Multiple actors are involved in the process of validating that the legal expert system is indeed faithful to its specification : the programmers of the legal expert systems, but also the lawyers of the legal department of the organization that contracted the implementation of the legal expert system.

In the interdisciplinary process of translating process-constrained legal specifications into code, only the lawyers possess the domain-specific knowledge required to validate the translation. On the other hand, only the programmer knows precisely what the code is doing. Then, the crux of this issue is to find a common ground between lawyers and programmers to communicate precisely what the code translation is doing and how it relates to the specification. The usual methods for validating the translation are based on different types of common ground.

The most widespread common ground and validation methods, in legal expert systems and elsewhere in the software world, is the crafting of test cases. A test case for a software system is an imaginary or real-world list of inputs of the software systems, along with the *expected* result of what the software system should produce on this list of inputs. A test case is like the conclusion of a court judgment; it describes the inputs facts along with the ruling about how the software should behave on these inputs. However, unlike court rulings that only happen after a law is enacted, test cases can be imagined before the software is deployed in production and produces actual decisions. In that way and in general for software systems, the crafting of test cases can be compared to pre-litigation discussions, where we imagine how a court might rule a specific situation according to interpretations of the law. In the case of legal expert systems, the crafting of those test cases maps precisely pre-litigation discussions of how different scenarios might unfold in court for specific situations.

Test cases are an efficient way to describe and reason about what a computer program is doing. Moreover, test cases are intuitive to lawyers because one of the main competence of lawyers is to know how to apply the law to a specific case. Therefore, the primary validation method for legal expert systems consists of test cases, as we've been able to empirically witness it in various parts of the French administration – though we see no reason why it would be different abroad.

Nevertheless, as the famous computer scientist Dijkstra pointed out as early as 1970 [Dijkstra et al. 1970], "program testing can be used to show the presence of bugs, but never to show their absence!" Indeed, a "bug" here can consist in a specific behavior of the legal expert system that does not respect the legal specification. A test case can exhibit such a bug, and allow the lawyer and the programmer to find the common ground necessary to understand and fix it. But potentially buggy behaviors of the legal expert systems come in often astronomical numbers, so much so that it is physically impossible to write a test case for each corresponding behavior. Consider the example of a legal expert system that computes the amount of taxes owed by a household depending on their income declaration. The French version of this program has multiple thousands of input variables needed to describe completely the income declaration. Not only there should be at least one test case per input variable, but also one test case per different combination of those input variables that trigger a specific provision of the tax law!

The combinatorial of the input space escalates the number of test cases needed to thousands or more. And each of those test cases requires a manual expertise from a lawyer that has to compute the expected output of the computer system by hand. Hence, in a test-case-based validation process, the role of lawyers is paramount and difficult: they have not only to craft the test cases by themselves,

but also be very imaginative as to find in advance all the possible corner cases that might reveal an incoherence of the computer system. This last task is made all the more difficult by the lack of access to and understanding of the computer system. The lawyers have to perform a kind of retro-engineering to produce high quality test cases, a competence not part of the typical lawyer training.

To sum it up, test cases are an intuitive way for lawyers to find a common ground with the programmers to specify the behavior of the application. However, validation by testing is inherently indirect and it is difficult in practice to achieve very high level of confidence on the exhaustive correctness of the code. Hence, it is interesting to complement testing with another method of code validation that is more direct: code review.

4 GUARDING AGAINST UNLAWFULNESS IN CRITICAL LEGAL EXPERT SYSTEMS

After having introduced the traditional computer science techniques for checking the correctness of legal expert systems with respect to their legislative specification, we explore the idea of a more direct validation using code review and mutual understanding between lawyers and programmers.

Reviewing a piece of code consists in reading and understanding the computer code, while imagining all the different inputs that could flow through it and trigger unwanted behavior. Code review is a commonplace technique in software engineering, so much so that popular source code management platforms like GitHub feature the central concept of *pull request* (or *merge request*). A pull request is a code modification proposal from an author of the software, that clearly displays the places where the proposal changes the source code. Pull requests can include a textual specification of what the code changes imply in terms of behavioral changes to the program, and are very often reviewed by other authors of the software before being accepted. In a pull request, the review is here to ensure that the source code change accurately match the specification of the changes in the program behavior – and of course that the changes in the program behavior are desired.

Meticulous code review, like test case crafting, is all about imagining all the possible situations the software might be confronted to, and discussing what should be the desired behavior of the program in those situations. Actually, code review often leads to adding new test cases with problematic inputs determined by inspection of the source code. In the case of legal expert systems, the desired behavior of the program is not always known, since law can be interpreted in multiple ways, and legal debates can only be settled by the courts following the legal process. But the developer of the legal expert system, when writing the code, has to determine in advance the behavior of the software for *all possible inputs*; she does not have access in advance to all the future court decisions that will inform the system's behavior. This is a major and fundamental incompatibility between the legal system and the operation of software systems. The former is reacts to reality *ex post*, the latter plans all possible behaviors *ex ante*.

However, this gap can be filled in practice by imagining in advance as many possible inputs that may result in court rulings that shape the behavior of the system. While courts cannot give opinions on imaginary cases, the lawyers responsible for validating legal expert systems have to, and this has already been happening in many government agencies and private organizations for decades. Each year, tax offices around the world struggle with new tax provisions that can be interpreted in different ways; however the legal expert system that computes taxes has to be ready by tax collection time, before any court can rule on the new tax provisions. As long as courts and magistrates are not interested in looking at legal expert systems during their design phase, these *ex ante* rulings on imaginary cases will have to continue to be performed by civil servants or private lawyers behind the curtains of the organizations crafting the software systems.

Going back to technical matters, while code review allows for a more direct validation than test case, its efficiency is heavily dependent on the competence of the code reviewer to link the abstract program behavior to the concrete computer code that implements it. In the case of legal expert system, this competence is difficult to achieve because only the lawyers have the abstract program behavior knowledge – derived from the legislative specification – while only programmers can read and understand the code. There are several partial solutions to this problem.

The first approach is to turn lawyers into programmers. This can be done via the use of low-code/no-code tools [Morris 2020]. These tools effectively help their users write an executable program using graphical user interfaces that are deemed more accessible than textual computer code for non-programmers. The weakness of these tools lie in their promise of accessibility and simplicity, as they often lack the advanced features used by software engineers to organize large computer systems into manageable and maintainable artifacts. Another proposal intends to merge lawyers' and programmers' skill by designing a programming language that either looks like the legislative specification, or can be translated to exactly the legislative specification. This is the intent of a long line of work based on logic programming [Sergot et al. 1986] or, more recently, on controlled natural languages [Listenmaa et al. 2021]. In this perspective, an extremely tight mapping between the law and the computer code is sought, sometimes referred to as an "isomorphism" [Mohun and Roberts 2020].

But instead of trying to create a new kind of lawyer-programmer hyper-specialist, a different approach emphasizes interdisciplinary collaboration at all stages of the code production process. Inspired by the Agile approach to software development [Beck et al. 2001], we have proposed a pair and literate programming methodology for transforming legislative specifications into executable code, based on the Catala domain-specific language [Merigoux et al. 2021a]. By locally interleaving fragments of legislative specification and fragments of executable code (literate programming), the lawyer and programmer (pair programming) can find a common ground for their discussion about the behavior of the legal expert system.

Such a collaborative process enables a kind of defensive programming when translating law into code. Concretely, on a given little piece of legislative specification, the programmer will start by proposing a rough translation to code. This first translation will be examined by the lawyer who will try to think of a situation that breaks the implicit assumptions that underlie the translation. For instance, to compute social benefits, a programmer might assume that each child is attached to one household. But in this situation, the lawyer may remind the programmer about split custodies: how are they handled in the translation? Is there another piece of legislation that specify what happens in this case? Iterating the translations and questions will lead to a code that is increasingly robust to corner cases, while maintaining faithfulness to the legislative specification at all times.

Often, a meticulous translation (*formalization*) of a legislative specification to executable code will reveal ambiguities (as described in Section 2), or worse: incorrections of the legislative specification itself. Then, the formalization process can also help build more robust legislative specifications.

5 TOWARDS ROBUST LEGISLATIVE SPECIFICATIONS

Building robust specifications is a problem that spans multiple domains of computer science. Often, a specification describes a process whose goal is to achieve a result that also enjoy its own specification. For instance, in cryptography, programs that encrypt data enjoy two levels of specification. First, the process-constrained specification describes meticulously each step of the computation to be reproduced by the executable program. Think of this first step as a cooking recipe. But second, a result-constrained specification states that the computation enjoys a specific security

property: without the cryptographic key, it is impossible to decrypt an encrypted message. To continue with the cooking analogy, the result-constrained specification is similar to the description of the final taste of the meal, after it has been prepared.

The first and second specifications are intertwined. More specifically, one must be able to show that following the process of the first specification leads to obtaining the results of the second specification. In computer science, this demonstration is done with a mathematical proof that relates the two specifications, yielding a correctness theorem. This is exactly what happens in cryptography, where researchers have successfully proved the security of important artifacts such as the TLS protocol [Beurdouche et al. 2015] that secures Internet traffic. Moreover, this correctness proof is written as a computer program that can be itself checked for logical correctness, to ensure that no proof mistakes could endanger the robustness of the result.

The correctness proofs for specifications are a particular link of the more general concept of chain of trust for critical software systems. This chain of trust relates the high-level intent of the program behavior to what happens exactly on the machine that executes the program. The computer science field of program verification is dedicated to building toolchains and frameworks for improving each link of this chain of trust, including specification proofs.

Now, we can apply some of the techniques of program verification to legal expert systems and their legislative specifications. The idea is to build a chain of trust for legal expert systems. First, the part of the code that follows a legislative specification should be cleanly separated from the rest of the IT system. Without this separation, it is more difficult to maintain the legal expert system since legal behavior issues can be mixed with other mundane IT-related bugs of the program, leading to confusion. Second, the legal-related part of the code should be linked to a process-constrained legal specification. This link can be made either through test-based validation or direct code review as discussed in Section 4. Third, the process-constrained legal specification should be proven correct with respect to a higher-level, result-based specification that states the intent of the law. This correctness proof can either be done informally or formally, provided that both specifications are formalized into some kind of programming language.

The correctness proof between the result-constrained intent of the law and the process-constrained legal specification of the computation goes beyond the problem of legal expert systems, and touches to the area of legal drafting. We claim that clearly laying out in the law the intent of the legislators before describing a computation improves understanding, readability and robustness of the law. For instance, article L521-1 of the French Social Security Code states that family benefits should depend on the income of the household and the number of dependent children. Then, article D521-1 actually defines the formula for computing the amount of family benefits that indeed depends on the two parameters named by L521-1. This dependence can be mathematically checked on the formalization of the formula defined by L521-1 to ensure the internal coherence of the law in force. Of course, the proof in this example is very simple, but the principle can be scaled to more complex cases where the legislator might capture the intent for *e.g.* fairness behind a computation also defined in the law.

By building more robust legal specifications for computations that clearly capture the intent of the legislator, it could become easier to create an informed public debate around reform proposals that tweak the computation parameters. The availability of code translations for important legally-defined computations is critical for an efficient legislative debate, as a minor tweak in a formula can have huge impact on a large subset of the population. Tools like LexImpact [Equipe Leximpact de L'Assemblée nationale 2019] that provide legislators a practical way of assessing the impact of a

reform proposal could be one of the best applications of a new generation of legal expert systems, based on increased trust over their correctness and robustness to legal ambiguities.

Beyond these technical checks and improvements, more implication of the legal system in the precise design and implementation of legal expert systems is also needed. As the goal of legal expert systems is to automate the enforcement of laws, they are confronted to the same the diversity of real situations as courts, except they cannot exert human judgment and are bound to follow what is in their code. Giving individuals affected by such automated decisions the right to appeal to human judgment, as disposed by article 22 of the GPDR, is a good countermeasure to possible deficiencies of the software system. But it should be complemented by a tight review and testing of the code of these systems by law professionals that act in coordination with the existing legal system. In practice, we can imagine a software development team from the tax authority submitting a batch of imaginary problematic situations in need of a court ruling to determine how the software should behave on this. Then, "imaginary court rulings" could be issued by a special court, creating case precedents *before any precedent is created in reality*. Of course, those "imaginary" rulings should be weaker than actual court rulings to ensure sovereignty of human judgment over real situations. But while imaginary, those rulings could smoothen up the legal expert system development and avoid bugs affecting large numbers of individuals that might have difficulty accessing the traditional legal recourses.

ACKNOWLEDGMENTS

Many thanks to Liane Huttner and Sarah Lawsky for their invaluable legal insights and deep discussions about legal formalization. A lot of the ideas developed in this piece originate from the common thought pool of the Catala research group, that also features Jonathan Protzenko. The 2021 edition of COHUBICOL's philosophers seminar also allowed me to gather precious feedback on the paper, and I would like to thank all participants for the lively discussions and debates.

REFERENCES

Layman E Allen. 1956. Symbolic logic: A razor-edged tool for drafting and interpreting legal documents. *Yale LJ* 66 (1956), 833.

Robert Balzer, Noreen Goldman, and David Wile. 1978. Informality in program specifications. *IEEE Transactions on Software Engineering* 2 (1978), 94–103.

Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development.

Benedikt Becker, Jean-Christophe Filliâtre, and Claude Marché. 2020. *Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup.* Ph.D. Dissertation. Université Paris-Saclay.

Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 535–552.

Joshua D Blank and Leigh Osofsky. 2020. Automated Legal Guidance. *Cornell L. Rev.* 106 (2020), 179.

N. A. D. Connell. 1987. Expert Systems in Accountancy: A Review of Some Recent Applications. *Accounting and Business Research* 17, 67 (1987), 221–233. https://doi.org/10.1080/00014788.1987.9729802 arXiv:https://doi.org/10.1080/00014788.1987.9729802

Conseil Constitutionnel. 2018. Décision 2018-765DC du 18 juin 2018. https://www.conseil-constitutionnel.fr/decision/2018/2018765DC.htm

Edsger Wybe Dijkstra et al. 1970. Notes on structured programming.

Laurence Diver. 2021. Interpreting the Rule(s) of Code: Performance, Performativity, and Production. *MIT Computational Law Report* (15 7 2021). https://law.mit.edu/pub/interpretingtherulesofcode https://law.mit.edu/pub/interpretingtherulesofcode.

Equipe Leximpact de L'Assemblée nationale. 2019. *LexImpact.* https://leximpact.an.fr/

Nel Escher and Nikola Banovic. 2020. Exposing Error in Poverty Management Technology: A Method for Auditing Government Benefits Screening Tools. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1, Article 064 (May 2020), 20 pages.

https://doi.org/10.1145/3392874

Antonio L. Furtado. 1983. An informal approach to formal specifications. *ACM Sigmod Record* 14, 1 (1983), 45–54.

Hugo Gimbert. 2021. Document de présentation des algorithmes de ParcourSup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/blob/master/doc/presentation_algorithmes_parcoursup_2021.pdf

Clement Henin and Daniel Le Métayer. 2019. Towards a generic framework for black-box explanations of algorithmic decision systems. In *IJCAI 2019 Workshop on Explainable Artificial Intelligence (XAI)*.

JetBrain MPS. 2019. MPS Case Study for the Dutch Tax and Customs Administration. https://resources.jetbrains.com/storage/products/mps/docs/MPS_DTO_Case_Study.pdf

Sarah B. Lawsky. 2017. Formalizing the Code. *Tax Law Review* 70, 377 (2017).

Baudouin Le Charlier and Pierre Flener. 1998. Specifications are necessarily informal or: Some more myths of formal methods. *Journal of Systems and Software* 40, 3 (1998), 275–296.

Inari Listenmaa, Maryam Hanafiah, Regina Cheong, and Andreas Källberg. 2021. Towards CNL-based verbalization of computational contracts. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*.

Légifrance. 2018. Loi n°2018-166 du 8 mars 2018 relative à l'orientation et à la réussite des étudiants. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000036683777

Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021a. Catala: A Programming Language for the Law. *Proc. ACM Program. Lang.* 5, ICFP, Article 77 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473582

Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021b. A Modern Compiler for the French Tax Code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3446804.3446850

James Mohun and Alex Roberts. 2020. Cracking the code: Rulemaking for humans and machines. (2020).

Jason Morris. 2020. Spreadsheets for Legal Reasoning: The Continued Promise of Declarative Logic Programming in Law. *Available at SSRN 3577239* (2020).

Joseph Potvin, Don Kelly, William Olders, Wayne Cunneyworth, Ryan Fleck, Craig Atkinson, Calvin Hutcheon, Ted Kim, Angela Bernal, and Stéphane Gagnon. 2021. Oughtomation: Practical Normative Data. (2021). https://gitlab.com/xalgorithms-alliance/oughtomation-paper/-/raw/master/ThesisJPotvin_Oughtomation_EXCERPT_1-87+references_draft-2021-05-09PDF.pdf

Charles Rich and Richard C Waters. 1986. Toward a requirements apprentice: On the boundary between informal and formal specifications. (1986).

M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. 1986. The British Nationality Act As a Logic Program. *Commun. ACM* 29, 5 (May 1986), 370–386.

SMU Centre for Computational Law. 2020. *The L4 domain-specific language*. https://github.com/smucclaw/dsl

Markus Voelter, Sergej Koščejev, Marcel Riedel, Anna Deitsch, and Andreas Hinkelmann. 2021. *A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV*. Springer International Publishing, Cham, 93–130. https://doi.org/10.1007/978-3-030-73758-0_4