SUBSTANTIVE LEGAL SOFTWARE QUALITY— A GATHERING STORM?

Marc Lauritsen
Capstone Practice Systems
marc@capstonepractice.com

Quinten Steenhuis
Greater Boston Legal Services
Qsteenhuis@GBLS.org

ABSTRACT

Readily available interactive programs dispense substantive legal guidance, often including bespoke documents. These are found across a wide spectrum of commercial and non-commercial contexts. Consumers are coming to rely on them as alternatives to expensive lawyer services. Yet their quality is uneven and difficult to assess. We are in danger of serious harm being done to unwitting users. How can we avoid an epidemic of artificial misinformation, systematic inaccuracy, and mechanical malpractice? This paper reviews how those dangers play out in real-world application contexts and explores ways in which the AI & Law community might help address them.

CCS CONCEPTS

• Software verification • Correctness • Interaction design

ACM Reference format:

Marc Lauritsen and Quinten Steenhuis. 2019. Substantive Legal Software Quality: A Gathering Storm? In *Proceedings of the 17th International Conference on Artificial Intelligence and Law*. ACM Press, New York, NY https://doi.org/10.1145/3322640.3326706

1. Introduction

Effective legal work can require a lot of cognitive and communicative labor. More often than ever before, machines are performing that work. We are seeing rapidly growing collections of automated guidance. Yet there is still a vast unmet need for reasonably priced, decent quality forms of legal assistance.

Many law-related software applications purport to dispense valid information and advice about legal situations via their interactive guidance (in screen-based "interviews") and assembled documents. Is there a big problem with quality? We don't really know. Which is a problem.

ICAIL '19, June 17–21, 2019, Montreal, QC, Canada

ACM 978-1-4503-6754-7/19/06…$15.00

How might we make sure that the substantive legal know-how expressed in such applications is correct? That they give the 'right' (or at least 'good') guidance and documents from a legal perspective for the full range of potential user inputs?

Similar questions have long been asked elsewhere in the software field, and oceans of words have been spilled about them. Yet few of those ideas have been systematically applied to the contemporary world of online legal advice systems.

This article is organized as follows: Section 2 describes the kinds of applications under consideration and their characteristic features. Section 3 reviews positive attributes we seek in such applications. Section 4 lays out ways in which applications may fail to exemplify several critical attributes, some reasons why, and the important values at stake. Section 5 takes up questions about how best to characterize the knowledge (and mis-knowledge) embodied in such applications. Section 6 reviews strategies that might be undertaken to ensure decent quality, their limitations, and practices that developers can be encouraged to follow. Section 7 describes related work. Section 8 considers how the AI & Law community and its tools and methods might help address these challenges. Section 9 concludes.

The authors have been deeply involved in developing software that supports legal work, both by fellow professionals and by lay people. One has taught law school courses in which students build applications using several of the tools described here. We are concerned about the current state of affairs but optimistic that significant improvements are within reach. Our hope is that this article frames useful questions and offers fertile suggestions. It is intentionally exploratory. We welcome input from and collaboration with members of the AI & Law community.

2. The world of automated legal expertise

The kinds of programs addressed here are found in a rich variety of contexts. Here is an illustrative list:

- Commercial providers of online assistance such as Legal Zoom (www.legalzoom.com) and Rocket Lawyer (www.rocketlawyer.com) offer collections of reasonably priced packages, some bundled with lawyer services.
- Specialized services like FlightRight (www.flightright.com) have arrived, and there have long been area-specific products like TopForm (topform.law) from Fastcase, a bankruptcy application. Upsolve (www.upsolve.org) provides free assistance in bankruptcy matters.
- Applications have been built for public and private purposes

using expert system and workflow automation development environments such as Autto (autto.io), Neota Logic (www.neotalogic.com), and Bryter (bryter.io).

- Innovative platforms like DoNotPay (donotpay.com) and QnA Markup (qnamarkup.org) have emerged that use chatbot-style mechanisms to deliver legal expertise.
- Legal document automation products such as Contract Express (www.contractexpress.com), Exari (www.exari.com), HotDocs (www.hotdocs.com), Leaflet (leafletcorp.com), Rapidocs (www.directlaw.com), and XpressDox (xpressdox.com) are widely used in law firms, legal departments, insurance companies, and other firms to bottle up legal know-how for use by staff and clients.
- Several dozen law schools now offer courses in which students write software applications as part of their education. Some culminate in public events like the Iron Tech competition at Georgetown Law School.
- There are labs, incubators, and related initiatives at many schools, bar associations, and other organizations.
- Hackathons around legal technology have proliferated, including global ones. Results are often prototypes or flash-in-the-pan demos, but some end up used by real people.
- The A2J Author system (www.a2jauthor.org) developed by the Center for Computer-aided Legal Instruction is widely used in legal aid, academic, and court contexts. CALI has recently launched a new platform at A2J.org via which student projects can be made available to the general public.
- Tyler Technologies offers an analogous system for unrepresented litigants called Odyssey Guide & File (www.tylertech.com). Many other vendors provide at least rudimentary form preparation as part of e-filing services.
- Local, state, and federal government agencies have fielded knowledge-based tools for citizens to get answers, apply for benefits, and otherwise access needed functions.
- The LawHelp Interactive service operated by Pro Bono Net is used in over 40 US states and territories, and several Canadian provinces. It has delivered about five million documents for free, and was used just under one million times in 2018 alone, powered by A2J Author and HotDocs.
- A multi-disciplinary research project is currently exploring the role of automated legal advice technologies (ALATs) in Australia. The project identified many applications in its recent report [2], but only touched on a few of the above.

All of the applications described above provide interactive access to codified legal knowledge. Sometimes this access is linear and conversational, and sometimes it is menu driven. Some offer ongoing assistance across multiple stages of a legal matter. They support a range of functions, including information gathering, answer giving, and document generation.

These programs are sometimes referred to as 'substantive' systems, since they embody legal knowledge in addition to performing generic automation functions. (The SubTech conference series has been devoted to such applications and their use and study in legal education since 1990. It met in Tallinn, Estonia in 2018 and will meet in Nashville, Tennessee in 2020.)

Most of the programs are deterministic and procedurally coded, similar to those highlighted by the technology report in the inaugural issue of the AI & Law Journal over 25 years ago. [10] Many of the platforms allow for graphical or menu-driven design, without requiring low-level coding. The material below summarizes aspects of how these platforms work and the artifacts they enlist and engender.

There are many names for the programs built using these tools: interviews, guided interviews, and applications being among the most common. For clarity, we will call all of the programs built using these platforms interactive legal applications, or apps.

This article will focus on how these issues play out in nonprofit contexts, but very similar ones occur in the other contexts mentioned above. Purveyors of legal knowledge-based systems take many different approaches, but face common challenges.

Among the tools described here, HotDocs (www.hotdocs.com) has the biggest market presence and most developed ecosystem. Over a half million copies have been distributed. Spun out of a research project at Brigham Young University law school, and later owned by LexisNexis, HotDocs is now part of AbacusNext.

HotDocs is a proprietary document assembly platform. Its applications are expressed in an XML format, but authors generally use development tools that offer dialog boxes to edit screens, create variables, and script computations. HotDocs uses a relatively expressive language for application logic and custom markup tags for formatting. Textual templates are edited in Microsoft Word or WordPerfect; graphical ones in a specialized editor that overlays dynamic fields on PDF forms.

Unlike the other listed systems, which provide a mostly linear user experience, HotDocs applications present sets of dialogs that can be freely navigated among to enter information in an order of the user's choice. Authors can alternatively limit forward navigation and encourage a linear flow for ease of use.

HotDocs has a long history and has wide adoption in the authoring of document-oriented applications for both legal aid organizations and commercial firms. It offers a very complete tool for automating graphical forms, with finished applications accessible both on the Windows desktop and online in a web browser. HotDocs has been the primary engine behind the LawHelp Interactive service, which has thousands of templates, some designed to generate dozens of forms from a single set of user answers.

HotDocs provides a broad set of tools for developer effectiveness and quality assurance.

- An Outliner is available to inspect and navigate the logical structure of document templates under construction.
- Interactive syntax checking is available.

- A robust debugger can step through and into code components in order to diagnose problems.
- The Component Manager and Template Manager provide facilities for browsing and revising components.
- A Text Management Tool has been developed by the US legal services community that can extract all textual strings in an interview, generate a document in which versions in a second language can be inserted, and automatically build an interview in the second language.

A2J Author (www.a2jauthor.org) is a web-based application platform for building interactive legal applications that it calls "guided interviews." Early versions used Macromedia Flash, but the latest version no longer requires the Flash plugin. The order of questions is determined at authoring time, but typically is linear, with ways to navigate out of order by linking to other screens and the ability to control paths with branching logic. Previously restricted to use by courts, legal aid, and law schools, it will soon be available under an open source license. A2J Author has served as a user-friendly overlay for HotDocs, but recent releases have included native document assembly, with a graphical template editor.

A2J Author stores applications in an XML format, but authors ordinarily create them on A2Jauthor.org using dialogs. Applications can include 'learn mores' in audio, image, and video format. Question text can include basic formatting, links, and popup helps. Applications can include logic with a syntax similar to HotDocs.

A2J Author includes some error checking via its preview function, which allows authors to view interview variables and evaluate test expressions. Authors can view completed applications as visual maps. Texts and logical expressions can be viewed separately, the latter with syntactical errors highlighted. Authors can run several kinds of reports. A2J Author also provides 'citation' fields to capture statutory or case law behind particular components.

Docassemble (docassemble.org) is an open source platform for building interactive applications based on a combination of procedural code and dependency satisfaction. The app author sets goals (such as assembling a document), and the Docassemble engine asks the questions needed to arrive at each goal. It is possible to force the engine to ask questions in a particular order. At any time, a Docassemble developer can view the goal that led to a particular question being asked.

Docassemble makes use of several technologies and is highly extensible. The core engine is written in Python. App authors use a mix of a human readable text markup language called YAML and logic written in Python. Docassemble provides an online editor with syntax highlighting, basic error checking, and variable insertion. There are also graphical "no code" front-ends for Docassemble that hide some of the complexity, including Community.lawyer and HelpSelf Legal. Docassemble apps can be accessed through a web browser (most common), automated via the REST API, through emails, or even through SMS text messaging. Docassemble apps can connect to external data sources, with some such integrations built-in, such as Twilio (for SMS and fax integration), Google Maps, and email.

App authors can extend Docassemble through modules that may include: custom object-oriented classes usable directly in the application; sets of reusable data (such as drop-down menus); or integrations with external data sources. App authors can share the modules they create on PyPi or Github, and then reference them directly in new applications. In general, Docasemble promotes a high level of code reuse and abstraction. Docassemble's package management and application authoring system also includes version control, through integration with Github, which can aid in code introspection and catching regressions. Basic input validation is included, as well as custom validation functions for run-time checking of user input.

Automated testing of Docassemble interviews has been performed using the Docassemble API and through the use of Gherkin scripts. Docassemble has built-in features for natural language classification as well as Random Forest regression.

Neota Logic (www.neotalogic.com) is a web-based "low-code" platform for developing applications that gather information from users, apply rules, and produce tailored outputs. It has been widely used in law schools, including in Georgetown's Iron Tech Lawyer competitions. Neota Logic describes itself as a rules-based AI platform that non-technical people can use to provide guidance, direct workflows, and assemble documents.

Neota Logic's reasoning engine uses both backward- and forward-chaining mechanisms to apply its rule bases to particular situations. It can work with external sources of data as well as rule sets stored in Excel spreadsheets. This can help with subject matter experts' validation of logic.

Neota has runtime Why Ask and Why Conclude functions, which show authors the logic paths causing questions to appear and results to be set. In Studio (its development environment) there's a similar function called Static Analysis that can display the logic path to and from any variable.

Neota Logic's current code base is a reimplementation of the Jnana expert system platform that began life in the 1980s. Much of its vocabulary and many of its design features hark back to that system. The company provides a deep collection of reference and training materials. Testing of the platform and its interviews has been performed with the Selenium web-testing framework.

QnA Markup (www.qnamarkup.org) is a compact language used to create chatbots, with basic document assembly provided by integration with Microsoft Word's mail merge functionality via a dedicated API endpoint. Applications are a combination of HTML and JavaScript that can be embedded directly into an existing website, allowing for quick iteration. A QnA app is written in a completely declarative and linear way, with branch logic represented by indenting a question underneath the initial question. GOTO statements allow more complex navigation through questions. Creating a basic chatbot uses a very simple

syntax, with Q(label): marking a question and A(label): marking one of multiple optional responses. Questions can also accept free-form response text.

HTML tags can be used to extend QnA markup, which can include arbitrary JavaScript, allowing for theoretically complex interviews, but the syntax becomes more challenging to debug when much such code is included. QnA allows authors to abstract some logic in JavaScript functions, but logic is best suited exclusively to interview endpoints. Runtime logic uses JavaScript, so the built-in debugging console in most web browsers is handy to have open while developing a complex QnA chatbot.

QnA has been used as a teaching tool inside law school laboratories as well as in legal aid and public defender websites, both for assembling simple forms and as a website navigation aid.

3. What We Seek

There are many desiderata when it comes to legal knowledge tools. You might think of them in the form of a pyramid of goals, some layers of which build on or pre-suppose ones below.

Some of these qualities are foundational. Applications need to reflect a base level of technical feasibility vis-à-vis currently available tools, environments, and other resources. They need to *work*. They should offer reasonable returns on the investments they require. The platforms and associated software should be secure, stable, and the delivery mechanisms reasonably accessible. Most importantly, they should produce desired *results*.

Ideal applications are well designed and usable. The user interface and experience can be critical in achieving other objectives. Applications should also be:

- Readable at median grade levels, in plain language
- Available in multiple common languages
- Consistent both in language and visually
- Mobile-responsive
- Ethical and lawful
- Annotated with sources of related resources and information
- Efficient to maintain
- Architecturally elegant
- Complete (within their specified domain)
- Compact and concise
- Explicable (logic should be transparent and auditable)
- Supportive of users' emotional needs

There are thus all kinds of reasons to prefer one tool or approach to another. They pose inevitable tradeoffs. Verbose explanations and disclaimers, for instance, may reduce readability and trust. Attempting to cover a wider range of information may reduce the quality of guidance.

All of the above qualities are important. But a key question to ask about a legal application is **Is it substantively right?** A system can be highly usable but incorrect, inconsistent, or incomplete. Even a system that satisfies these goals when it is completed must be *kept* correct over time as the relevant legal world changes.

Correctness is orthogonal to usability and other desiderata. It should be front and center. That quality is the focus of this article. How do we get to ascertainable correctness? Put another way – how can we make systems that not only earn but deserve the **trust** of their users?

4. Where We're At

Even beautifully presented and highly usable inteactive legal applications can give wrong or misleading guidance to users. Many lack appropriate disclaimers and scope limits. Few can withstand critical analysis. An insightful analyst can readily discover and document defects and holes. This is especially troubling because consumers may suspend, or be unable to use, normal techniques for judging the quality of advice they get.

Apps may have all kinds of latent defects. Substantive errors can be more subtle than functional ones, where it's usually obvious that something is broken. Errors may arise from:

- Failure to correctly translate legal rules into program logic, due to lack of development expertise by lawyers, and lack of legal expertise by developers.
- Unclear understanding by the user of what the application is asking or saying, leading to incorrect input. This risk may increase when there is an intermediary advocate or navigator.
- Use of an application in the wrong circumstances, because of a lack of appropriate screening or 'triage' of the user.
- Failure to maintain software, which then becomes incorrect or "orphaned" over time. Often this is due to failure to budget the cost of maintenance in the original project scope.

Translation errors in particular can arise because apps often are the product of informal collaboration among non-professional developers and SMEs, some with conflicting views about substantive logic. When application logic is embedded in unfamiliar computer code, it can be difficult for an SME to verify it, and it can be hard for a developer to explain how the program operates.

4.1 Some reasons

What are the main drivers of software quality problems in interactive legal applications? Even basic applications can quickly become complex when all of their paths and configurations are considered, which to borrow a term from physics we can call the "phase space" of the application. Both the Is and the Ought (what a program does and should do) are hard to definitively describe. The sheer quantity of scenarios is combinatorial and eludes exhaustive enumeration.

These projects are often committee projects, with multiple SMEs and reviewers. Many teams only work episodically on the applications. Their members have largely non-overlapping mental models. They often have not had prior experience with a software development project, and they are generally not co-located.

In the nonprofit sector, these risks may appear more often. Meager resources may mean that projects lack experienced professional project management and planning, and the nonprofit sector often values wide stakeholder input at the expense of clear project direction. This can result in a cacophony of voices from clients, stakeholders, and subject matter experts. Many projects proceed in a very *ad hoc* way. Projects may launch without clear setting of requirements or interview scripts. Ongoing maintenance and outreach is generally unfunded, which may lead projects to wither, still available online but unmaintained.

Lack of software engineering rigor and standards in the development process can make maintaining code created by a different developer especially challenging in the interactive legal application realm. Software projects with graphical editors, particularly, can lead to program logic being buried and hard to visualize as a whole. Solved problems in the wider software engineering culture are not familiar to many legal application developers.

Compromises are made out of expedience. Design choices can be driven by non-substantive considerations, like data integration aspects with case management or e-filing systems. The rationales for programming and design decisions often are lost. Little adjustments can cause inadvertent substantive degradation.

Systems may make assumptions that end users would not endorse or understand, such as resolving doubts by checking things on a form despite uncertainty in the associated facts or implications.

Users can be too forgiving when services are free or inexpensive. Providers perceive immunity from liability, and disavow responsibility, offering apps on an 'as is', no-warranty basis. Consumers are told to use them at their own risk. Providers take comfort in the fact that authors and publishers of do-it-yourself books and videos are generally not liable for misinformation.

On top of all this, there is a seduction of novelty over bullet-proof-ness, of quantity over quality. Apps can be shiny objects that sponsors can brag about, without looking too closely.

4.2 What's at Stake

When we use a program like Microsoft Word or Excel, we expect it to behave semi-intelligently. We know that it has a rich variety of features and functions, and mechanisms to invoke them. We are pleasantly surprised when it anticipates our needs or corrects our errors. But we don't expect such programs to communicate substantive knowledge about the world. Legal apps are different – they should behave as expected; but we also expect them to give valid guidance and generate appropriate documents.

"Bad" apps and errors in either the substance or use of otherwise good apps provide fodder for critics of the very concept of online self-help tools. Failure to self-police quality could lead to a chilly regulatory environment. Needed experts may lose confidence and become cynical if a system misbehaves too obviously.

Despite these risks, we recognize that only a tiny fraction of the legal know-how that could be codified for interactive delivery has been. In most situations people without lawyers are out of luck. Many populations live in legal advice deserts. Users come to our applications with questions, intentions, problems, hopes, and dreams. Such users are ordinarily dealing with a personal plight or small business problem, and are not being assisted by a lawyer or other expert. The usage context is one of high dependence and vulnerability. They deserve 'food' that is safe to eat, and 'water' that is potable. Our primary obligation is to users who depend on apps like those reviewed here.

4.3 Setting expectations

Some software quality problems are inevitable in interactive legal applications, just like they are in multi-million dollar projects engineered by large technology firms. We have to accept a certain level of fallibility in both machines and people, as well as understand the alternative. After all, we can't exhaustively test people on their knowledge either. We let inexperienced professionals advise and represent people. Should that incline us to be nonchalant about interactive content that is of unknown quality? Or should we demand more from rule-based systems? There's a great chasm between what can reasonably be done with current resources and what ideally should be done. So, should we settle for 'adequate' and tolerate questionable quality? We can't avoid amateur developers. How can we better empower them?

5. Characterizing the knowledge

A central problem in this domain is making legal correctness more transparent, both when the interview author translates subject matter expertise into computer code, and when a user runs an application and receives information or documents.

Interactive legal applications contain at least three kinds of logic: legal rules, interview logic, and formatting and display logic. In software engineering a common design pattern that covers the differences among these types of logic is the classic model, view, controller pattern. Yet, existing application platforms offer few ways to represent legal rules as distinct from the other kinds of logic. Legal rules are mixed in with application logic at best, or worse, embedded in the forms. This makes it difficult for subject matter experts to review logic for correctness as a matter of law.

Interactive legal applications also contain legal information in the form of help text, links to external information, and instructions embedded in the questions. It is important to include this more traditional information in any reviews for correctness.

Applications can convey information or misinformation by what they 'say' (when interacting with the user as well as in outputs):

- Static information and help screens
- Texts that are personalized to the user's situation

And by what they do:

- What questions are asked or omitted
- What documents are generated or omitted
- What passages are included or excluded

- What words go where
- What boxes get checked, or words circled or stricken

This information may be substantively correct or incorrect in several subtle ways:

- It may cover rules, practical considerations, and procedures for different courts and situations.
- When addressing strategy, it may involve questions of judgment that vary among subject matter experts.
- It may not fully advise users about its scope or limitations.

Finally, design defects may lead users to misunderstand correct information or to provide incorrect input.

Even a system that does as intended may result in bad user outcomes. The user may misunderstand a question, or give an inaccurate answer. Even if the system operates flawlessly, other parties may not do as expected.

In order to make reasoning visible and verifiable, the system must expose the underlying assumptions it makes as well as the logical inferences it draws and the strength of those inferences. Much of this behavior is expressible in "If this then that" structures. The data entered by users mostly controls what happens. But programs can also react to other information. In any event they pose a finite set of *relevant* states to be evaluated, with predictable edge cases.

Much relevant knowledge is expressible as a collection of deontic propositions: obligations, prohibitions, and permissions. Certain forms are required in certain circumstances; certain information is required in certain places in those forms in certain circumstances; certain kinds of information are permissibly included in certain places; other kinds are impermissible.

In addition to their explicit utterances, programs cast a penumbra of implications—understandings that a reasonable person might draw and rely on. Users expect applications offered by reputable organizations to embody trustworthy voices of experience.

Data types and validity conditions are kinds of messages. Affordances can be interpreted as implicit permissions. ("You let me do ___ even though I said ___.") They can reasonably be understood as recommendations and suggestions: "It's OK to omit/include this information;" "It's OK if we abbreviate here."

Providers of online tools have an affirmative obligation to avoid misunderstandings. We need to take into account the competencies and vulnerabilities users bring to the experience.

Dealing well with a legal problem or opportunity involves more than information and documents: things need to be done with other parties, courts, agencies. At present these are ordinarily left as 'an exercise for the reader.' But systems will eventually take up more roles as agents, interacting on behalf of users with those external parties, such as electronically filing documents.

6. Strategies and their limitations

Making programs behave the way you want can be subtle and painstaking work. Fixing and updating apps are chores. How can we make it easier to be right? Quality is expensive. Might there be ways to radically reduce its cost?

There are a number of approaches (both preventive and remedial) to development processes, including program architecture and features. Two venerable software development disciplines of course can play a big role: specification and testing. Although discussed below as independent phases, these should be considered iterative cycles, consistent with Behavior and Test Driven Development principles.

6.1 Specification

Externalizing logical rules would aid in some of the goals mentioned in Section 3, and various authors have made efforts to represent logical rules in a platform-independent way. Some authors have used Google Sheets to represent rules. In the case of HotDocs, such rules need to be compiled into code, but Docassemble can make direct use of those rules. [14] Neota Logic also allows for external representations of logical rules.

Jason Morris has adapted the Legal Case Based Analogical Reasoning Tool to create applications with logic independent of a specific interview, reasoning by analogy to previous cases, and directly usable within a Docassemble interview. [11]

From the business process world, the Decision Model and Notation (DMN) representation can express complex logical rules. Rules expressed as DMNs could be directly integrated into systems such as Docassemble, or compiled into code for other platforms. This is a promising avenue for improving interactive legal application quality and maintainability. Dimyadi et al. [4] report one successful effort along these lines.

Another interesting approach to specifying legal rules is to write them as functions or object-oriented classes with abstract interfaces. Rather than embody the logic inside an interview, then, the interview can just present the relevant variables as parameters to the function, and let the function tell you whether the rule applies or not. The code still must be written by a developer and reviewed by a subject matter expert. The main advantage is when the same legal rules are used in multiple interviews. This approach would also assist authors in developing unit tests to avoid regressions as an interview is amended over time.

Proscriptively, in the planning and implementation phase, authors should use methods to inventory and codify the legal and business rules that the system will be implementing. Often the legal rules may derive from a body of case law and statutes, but subject matter experts should translate any rules that the system will follow into plain English, in a form analogous to a syllogism (IF A and B, then C)[1]. Then it can be the author's project to translate those rules into computer code. Methods of externally expressing

[1] In case-based reasoning approaches, the SME would help classify cases and select appropriate probabilistic thresholds, rather than writing deterministic rules.

the legal rules that can be directly used by the reasoning engine discussed above will cut down on translation errors to the extent that they can reviewed or authored by the SME more easily than traditional code can be. This is likely to be an iterative process. Business rules that are familiar to the subject matter expert through their daily practice may be unwritten or invisible, but become clear when the automated system is used.

The same care and attention must be given to the information that is not expressed in a rule, but is still delivered to the user, through help texts or materials produced for the user. Such texts should be written and reviewed completely by the SME, with editing for plain language and usability also subject to SME review. If alternate information is delivered to the end-user, rules should be written to clearly express in which circumstance a given set is delivered. This information can be referenced in comments.

Visual representations of interview paths can be useful, but a flowchart representing an interview with hundreds of paths will often be harder to read than the code. Legal applications can be visualized as finite state automatons, but this representation may not be useful when it is too hard to translate from this state into working code or difficult for a subject matter expert to review.

6.2 Testing

Testing is often the least rewarding (and perhaps uncompensated) phase of an application development project. Yet steps can be taken to make it more feasible. In the testing phase, authors should take systematic efforts to comprehensively validate the business and legal rules that were made explicit in the planning phase. Automation can be useful here.

Users of API-driven platforms such as Docassemble can develop test answer sets to validate logic against interviews, verifying that variables are correctly set. A similar facility exists in Neota Logic. Almost all application platforms offer interaction via a web browser, and web automation platforms such as Selenium or cucumber.io can be used to drive the interview exactly as an interview user would.

Neota authors have a testing mechanism called App Test to identify regressive errors in application logic. An author can save any session as a Test Case, and accumulate them into a Test Set, which can be run automatically. Test Cases can also be imported from Excel. Deviations from expected results are displayed. NLS Solver enables authors to generate large sets of permutations of input parameters to use in functional and regression testing.

Overall, automated tests are best at capturing the errors that the interview author has already considered, and then for catching regressions caused by later changes. Even with automation, there is no substitute for testing by real-world users.

The primary barriers to testing are the lack of testing expertise in the application development world, lack of budget and planning for the testing phase, and lack of purpose-built tools that take advantage of the rich semantic meaning in an application. Each legal application has much more in common with other such applications than the wider universe of software applications, and that similarity should be used to advantage. Most testing platforms are built to validate web sites, not for semantic or legal correctness, but for elements staying in the correct place on the screen. A purpose-built legal application-testing platform could take advantage of knowledge about legal rules and semantic data validation, for example, to generate automated testing data that follows realistic patterns. This would make it easier to test new features without writing completely new test cases, and just as in the specification phase, tests that directly pull from external rule sources will minimize translation errors.

6.3 Validation by users

End-users rightfully want to understand the reasons for the legal information that they were given or the form that was produced for them. Systems that can externally express business rules might also allow for the user to receive an audited report that shows their singular path through the "phase space" of the interview. This is possible in systems without externalized logic as well, but likely would require writing rules twice, leading to possible errors. Short of a fully audited path, feedback provided to the user in the course of the interview will deliver most of the benefits. After reaching a key branching point, expressive applications should tell the user the impact of the information they just entered. For example: "Because you requested discovery, your court case will be postponed by two weeks until June 20, 2019." Users can also be given a chance to watch brief overview videos or read a short summary that explains the basics of what they will do.

Providing interim feedback can keep users from feeling that they are trusting their legal outcome to a black box. Because users know more about the outside world than can be codified into an application, these validations can also help subject matter experts catch errors in the logic early. For example, a user may recognize that the system is relying on an answer that they did not provide, or know a real-world fact (such as the fact that the court is closed on the rescheduled date) that the author did not take into consideration. If the decision isn't explained, the user may be left feeling that the interview knows something they don't and blindly follow incorrect advice. User testing leads to actionable improvements; leaving answers unexplained means some users may never know that they received bad advice.

Leveraging end user error-catching requires having easy ways for them to bring them to the attention of developers. Systems could allow reports of erroneous or dubious results, and include anonymized dumps of answer sets, when that happens.

6.4 Formal Verification

Formal verification overlaps with testing, and is possible for code sets that can be reduced to a finite state automaton. Such sets can probably be limited to the business and legal rules, and omit the questions of judgment, usability, and correctness of generalized advice that requires a human's review. Focusing on a specific aspect of the legal problem, contract enforceability, Meng Wong and others have created L4, a domain-specific language for

expressing formally provable legal agreements. [16] This area of research is new and may provide important lessons.

Formal verification is only useful to the extent that it is performed on exactly the same rules that are relied upon by the reasoning engine. Externalized rules are a great aid here. Without external rules, logic must be translated into the provable language, which can itself be a source of errors. It's not a replacement for manual review for correctness, but can supplement such review and aid in preventing regressions.

One of us has proposed a debugger that could inspect the de facto network of pages in an A2J interview and

- identify which pages can never be reached;
- plot the various paths that *can* be traversed;
- for a given page, identify the path(s) via which it can have been reached (and the associated 'statements' the user is assumed to have made by pressing certain buttons)

The approach essentially is to translate the interview into a finite state automaton. Once in this state, it can be easier to evaluate the its endpoints and verify that they are correct. The "phase space" of the interview can then be walked in an automated way, similar to API-driven testing of a Docassemble or Neota Logic interview.

6.5 Other tools and approaches

Solutions to the visibility question can target three different aspects of application development: planning and initial implementation, testing and auditing, and explicability. Systems that limit the steps in translation between the SME and developer, and those that allow for verification at the level of individual variables rather than relying on parsing generated output, will be more likely to succeed in achieving validation goals.

Mechanical Turk approaches, where humans are paid or otherwise incented to test and comment on applications, are worth exploring. When gamified, tedious but socially valuable tasks can be eased. One interesting example is Learned Hands, a gamified project to train a legal classifier. [6] Social production approaches in general could be promising, such as collaborative code review.

There are things that platform operators can do: track 'freshness,' require updates, or channel feedback. Developers can seek to make their applications bullet-proof by anticipating problems. When combinations of answers would lead to branches of unknown quality, it is best to prevent users from going down them. Input validity checking should be used aggressively.

Having subject matter experts actively participate in a system project is critical, both for purposes of specifying and validating the its knowledge. Even if it is impossible for such a person to thoroughly confirm every possible behavior of the system.

One thing that legal advice software can and should be clear about is that there's a lot of important legal knowledge that can't yet effectively be expressed in software. We should document and declare all compromises with quality. Opinion or advocacy should also be labeled as such. There should be clarity about provenance and scope in order to avoid misleading users.

A worthy effort might be to develop coding guidelines like the PEP8 Python style Guide. [15]

6.6 Standards

For such an active and growing field, surprisingly few standards of practice have emerged.

How can we best characterize and measure the distance of mismatch between a system's performance and some ideal? How do we define 'good' advice or forms? What's the standard of care? Is it the same standard we would apply to a paralegal or attorney taking the role of the interactive legal application? What ideal should we aspire to? Avoid things that would be malpractice if done by a lawyer? The 'whole truth and nothing but the truth?

Both the end(s) and the means can be murky – that is, what standards to seek, and how to meet them. But much of the time the rules are straightforward. What's 'right' is often uncontested and non-controversial.

At a high level of abstraction, system behavior consists of presenting texts, images, and sounds to users that communicate and elicit information. Those presentations vary based on user responses (and sometimes other information about the presenting situation that the system can ascertain, such as the user's profile, the device they are using, data previously gathered and stored, or their nonverbal behavior in the present session.)

In analyzing these exchanges it is important to adopt a wide-angle lens. Systems should be accountable not just for data explicitly entered into fields by users, but for everything they can reasonably 'know' and infer about a user and their current session. Likewise, they should be accountable not just for the communications they explicitly render, but those that reasonable users may infer from the system behavior. Users also bear some responsibility for how they use these applications, and what they make of their results.

One reasonable approach is to ask what a human expert would say and do if he or she were presented with the same set of facts and requests. He or she should not need to be a credentialed lawyer; many paralegals have equal expertise in legal procedures.

We suggest this general framework:

- For every possible set of user inputs, the system should behave in a way that a human reasonably proficient in the relevant legal 'art' would not find objectionable. That could either be by providing information and/or documents that represent the, or at least *a*, 'right' response, or by advising the user that such a response cannot be given. (This of course is a high bar. The ability to say "I don't know" or "I'm not sure" however provides considerable wiggle room.)
- It should *not* ever behave in a way that a similarly situated human expert presented with the same fact pattern would regard as wrong. (Many points are uncontroversial. Strategic guidance, like whether to ask for a jury trial in an eviction

case, can be more contentious.)

6.7 Ethical and policy issues

Software doesn't have principles, or a code of ethics. But lawyers and software developers do. Even if lawyers involved in software projects are not acting in a trusted relationship with a specific client that triggers the full panoply of their professional responsibilities, they still have generalized duties to promote the public good and not do harm. And computing professionals have frameworks like the Association for Computing Machinery's Code of Ethics and Professional Conduct, which includes mandates to 'avoid harm' and 'strive to achieve high quality in both the processes and products of professional work.'

Besides issues of social desirability these applications raise questions of legality. In some jurisdictions authorities contend that they represent the unauthorized practice of law. Current developments in North Carolina and Washington State warrant attention.

7. Related Work

Hokkanen and Lauritsen long ago pointed out that legal knowledge tool makers can and should make better use of knowledge tools themselves. [7]

Conrad and Zeleznikow [3] remind us of the critical role of evaluation in research projects, especially those that produce applications intended for real-world use. Most of the applications described in the present paper are not products of academic research, but nonetheless would benefit from the kinds of multi-faceted assessment that Conrad and Zeleznikow outline. That includes performance evaluations where the system is compared to known baselines, ideally using publicly available data sets.

Ramakrishna et al. [13] lay out techniques for bridging the gaps between domain experts and knowledge modelers. Their goal is to represent knowledge in a way that can easily be understood by a practitioner yet be expressive enough for a knowledge modeler to formalize. Semi-formal and more formal representations are required. They propose a process based on competency questions. While this is applied to the case of developing ontologies, such work involves qualities like accuracy, completeness, and consistency that are central to the present paper.

Al-Abdulkarim, Atkinson, and Bench-Capon [1] outline an 'Angelic' methodology for designing case-based reasoning systems using an Abstract Dialectical Framework. While this method is intended for applications that reason using factors and dimensions rather than ones embodying rules and document models, it offers practical ideas for tackling problems in a systematic and reproducible manner, using a database that encapsulates a domain theory, tools for visualizing and querying that data, and tools to facilitate collection and use of test data.

Faciano et al. [5] describe a tool called FormaLex that checks legal documents for coherence problems. Three state-space reduction strategies are described. Along the way their article reminds us that there is a vibrant community of model checkers and the performance challenges they have faced.

Muthuri et al. [12] explore how normative spaces can be made accessible at the information architecture level so as to enable non-experts to manage legal risks. They use Value Delivery Modeling Language (VDML) and the Easy Approach to Requirements Syntax (EARS) framework to present legal jargon in an accessible form to engineers. EARS provides six patterns for expressing preconditions, triggers, and responses to events. Argumentation schemes are used to reduce the complexity in interpreting legal provisions, which can be summarized in compliance patterns following a context-problem-solution format.

Several efforts in the area of contract formalization and verification raise similar themes to those explored here. Tom Hvitved's dissertation "Contract Formalisation and Modular Implementation of Domain-Specific Languages" [8] is one.

Professor Finale Doshi-Velez at Harvard University's School of Engineering and Applied Sciences is doing important work around interpretability, which includes extracting explanations from arbitrary models.

The Essence framework proposed by Ivar Jacobson and his colleagues [9] offers promising ideas for legal software developers. By reifying practices – such as scrum, kanban boards, use cases, and user stories – its visual language facilitates conversations about ways of working and surfaces convenient places to ask and answer questions, including via interactive games. Some challenges described in this article could be addressed by implementing, or emulating, aspects of the Essence kernel.

We could also pay much more attention to foundational content-driven standardization, such as is being pursued in initiatives like Akoma Ntoso and Discourse Representation Theory.

8. AI to the rescue?

So, what's to be done?

One short answer is that some of us doing this work need to do a better job. We should start following better development paradigms.

A lot of real-world legal knowledge work automation is happening. It may not be dealing directly with advanced AI & Law themes. But the contexts described here present an opportunity rich space, with powerful R&D challenges. How might the AI & Law community help? You might consider this the unfinished business of earlier AI & Law.

Should we devise programs that 'exercise' other programs; that define and document them? A program that interrogates other programs might extract a decision tree or truth table from an interview so that it can be validated by domain experts. It might notice gaps and contradictions. What else might such utilities do, and how? Could they find faulty 'circuits' and generate catalogs of potential errors and deficiencies? Might they set cognitive

breakpoints, so as to more easily surface intermittent inferences, and shine a light on the middle layers of reasoning at play?

9. Conclusion

Achieving progress along the lines imagined here presents enormous challenges. Applications shouldn't just be correct, but *ascertainably* correct. And that should be true not only with respect to the explicit messages being communicated in interviews and documents, but with respect to all of the implicit ones users might receive. We're dealing with complex communicative processes, both during development and at runtime.

Are those of us who have been enthusiastically building the kinds of apps described here in danger of flooding the legal 'roads' with vehicles that are *Unsafe at Any Speed*? That may be overdramatic. But if there's a storm coming, we're not very ready for it. In a world of imperfect tools, imperfect developers, and imperfect users how do we avoid an epidemic of bad legal software?

These are ancient issues and ideas. Many were current in the heyday of expert systems in the 1980s. But regrettably few are front-of-mind in any of the contexts mentioned. There are clearly many *existing* tools and techniques that are not yet routinely exploited. Yet the thorny challenges described above will be increasingly central as more and more legal work is done with machine assistance.

Machines can't yet do a lot of things that human lawyers can do. But those things they can do they should do flawlessly. Which requires better machine tools, and better machinists.

Automated legal services may be the best hope for access to justice and legal wellness for billions of our fellow humans. AI & Law activists are encouraged to find ways to bring their utensils to the feasts of knowledge automation that lie ahead.

Acknowledgments

We are grateful to Bart Earle, Claudia Johnson, Gabe Teninbaum, Karen Cannata, Jonathan Pyle, Kevin Mulcahey, Michael Mills, and anonymous reviewers for their thoughtful reactions to drafts of this article.

References

[1] Al-Abdulkarim, L., Atkinson, K. and Bench-Capon, T., 2017. Angelic Environment: Support for the Construction of Legal KBS. Technical Report ULCS-17-002, University of Liverpool.

[2] Bennett, J. et al. Current State of Automated Legal Advice Tools, Networked Society Institute Discussion Paper. April 2018. https://networkedsociety.unimelb.edu.au/__data/assets/pdf_file/0020/2761013/2018-NSI-CurrentStateofALAT.pdf

[3] Conrad, J.G. and Zeleznikow, J., 2015. The role of evaluation in AI and Law: an examination of its different forms in the AI and Law journal. In Proceedings of the 15th international conference on artificial intelligence and law (pp. 181-186). ACM.

[4] Dimyadi, J., Bookman, S., Harvey, D. and Amor, R., 2019. Maintainable process model driven online legal expert systems. *Artificial Intelligence and Law*, *27*(1), pp.93-111.

[5] Faciano, C., Mera, S., Schapachnik, F., Di Iorio, A.H., Clara, B.L., Uriarte, V., Giaccaglia, M.F., Ruffa, M.B. and Marcos, C., 2017. Performance improvement on legal model checking. In Proceedings of the 16th edition of the International Conference on Articial Intelligence and Law (pp. 59-68). ACM.

[6] Hagan, M., Colarusso, D., 2018. Learned Hands: What is It. Retrieved from https://learnedhands.law.stanford.edu/, *archived at* https://perma.cc/2VXG-CYHT

[7] Hokkanen, J. and Lauritsen, M., 2002. Knowledge tools for legal knowledge tool makers. Artificial Intelligence and Law, 10(4), pp.295-302.

[8] Hvitved, T. "Contract Formalisation and Modular Implementation of Domain-Specific Languages"—Ph.D. dissertation, The Faculty of Science, University of Copenhagen. https://drive.google.com/file/d/0BxOaYa8pqqSwbl9GMWtwVU5HSFU/view

[9] Jacobson, I., 2019. What is Essence? Retrieved from https://www.ivarjacobson.com/services/what-essence

[10] Lauritsen, M., 1992. Technology report: Building legal practice systems with today's commercial authoring tools. Artificial Intelligence and Law, 1(1), pp.87-102.

[11] Morris, J., 2018. Legal Expert Systems Just Got Smarter. (October 2018). Retrieved from https://medium.com/@jason_90344/legal-expert-systems-just-got-smarter-e7e12b75e872, *archived at* https://perma.cc/URL5-HQSA

[12] Muthuri, R., Boella, G., Hulstijn, J., Capecchi, S. and Humphreys, L., 2017. Compliance patterns: harnessing value modeling and legal interpretation to manage regulatory conversations. In Proceedings of the 16th edition of the International Conference on Articial Intelligence and Law (pp. 139-148). ACM.

[13] Ramakrishna, S. et al., Legal Vocabulary and its Transformation Evaluation using Competency Questions, 2015. In Proceedings of the 15th International Conference on Artificial Intelligence and Law (pp. 211-215). ACM.

[14] Steenhuis, Q., 2018. Separating Interview Logic from the Law. (June 2018). Retrieved from https://www.nonprofittechy.com/2018/06/28/separating-interview-logic-from-the-law/, *archived at* https://perma.cc/2M3S-S5X6

[15] Von Rossum, G., Warsaw, B., Coghlan, N., 2013. PEP 8 -- Style Guide for Python Code. (August 2013). Retrieved on April 24 2019 from https://www.python.org/dev/peps/pep-0008/.

[16] Wong, M. et al, 2019. L4: a domain-specific language (DSL) for law. Retrieved from https://legalese.com/#L4, *archived at* https://perma.cc/V8T7-3AZT