Software Engineering for Research on Legal Argumentation

Thomas F. Gordon

Abstract

Research on legal argumentation is interdisciplinary, with contributions from several academic disciplines, including computer science. This paper presents an overview of some software engineering methods from computer science and discusses their potential contribution to the study of legal argumentation.

Introduction

Research on legal argumentation is interdisciplinary, with contributions from legal theory, philosophy, computer science and linguistics, among other academic disciplines. Artificial intelligence (AI) is another interdisciplinary field of study involving these same academic discipines, but with a focus on computational models of cognitive processes, including but not limited to argumentation. Legal theory contributes to AI via the interdisciplinary field of AI and Law, which not only applies AI results to legal tasks but also uses legal tasks to develop innnovative AI methods, including contributions to case-based reasoning, nonmonotonic logic and computational models of argument. In return, computational models developed in AI and Law have shed new insight on legal methods, including legal argumentation, which legal scholars have found interesting and useful (Sartor 2005).

Here we discuss the use of software engineering methods, developed in computer science, for research on legal argumentation and make some preliminary conjectures about contributions to research on legal argumentation made possible by these methods. What does software engineering have to offer the study of legal reasoning? What kinds of insights onto legal argumentation are facilitated by software engineering methods?

Overview of Software Engineering Methodology

It would be misleading to speak of *the* software engineering methodology, since several have been developed and are used in practice. But the distinctions between these various methodologies are not important for our purposes here, since they have much in common and their differences are presumably minor details compared to the difference between their common features and the methodologies of other disciplines.

The traditional software engineering methodology, still used by some, is the *waterfall* methology. It is a process model consisting of several phases:

- Requirements Analysis
- Design
- Implementation
- Verfication, and
- Maintenance

The process is mostly sequential, from one phase to the next, but the model does allow for some feedback from later phases to earlier phases. For example, problems arising in the design phase can cause requirements to be revised.

Currently fashionable is *agile* software development, such as the Fowler's lightweight development process (Fowler and Scott 2000) or the Scrum methodology (Sims and Johnson 2011). Rather than trying to design the whole system before beginning work on the implementation, the development process is broken down into smaller parts, in a cyclic process. In each iteration of the process, called a *sprint*, a small number of requirements are selected from a *backlog* of requirements, and system modifications are designed, implemented, tested, and documented to meet these requirements. Each sprint produces a working, useful system, mitigating the risks of the waterfall model, where no system exists until much later in the process. The phases of the waterfall model remain important when applying agile development methods but occur repeatedly, in each iteration (sprint), rather than only once per project.

Let us try to summarize each phase of the development cycle, starting with requirements analysis. Here too various methods exist. Let us focus on agile requirements analysis methods (Leffingwell 2010). Requirements analysis takes place in collaboration with domain experts, potential users and, in particular, the *customer*, if there is one. For example, when developing legal applications, software engineers would work with lawyers. Requirements are captured by defining roles, personas, user stories, scenarios and acceptance tests. Roles classify the different kinds of users of a system. For a legal application these might be lawyers, judges, clerks, clients and so on. Personas are more concrete descriptions of fictional persons or characters in each of the roles, to help the developers to have a better understanding of the skills, interests and attitudes of typical users. Users stories describe particular requirements by instantiating a template of the following form: As a *role*, I want to be able to perform *action*, so that I can *goal*. For example: As a lawyer, I want to be able interpret the arguments in court opinions, so that I can apply the decisions to build arguments in my cases.[1] Scenarios are more concrete illustrations of interactions among several users stories. In scenarios, users stories are instantiated with personas and more particular actions and goals and then linked together into sequences of actions. Finally, acceptance tests are defined for each user story, with the aim of operationalizing the evaluation of whether or not the user story has been correctly and completely implemented. We will come back to the topic of evaluation below, when discussing the verification and validation of software.

The users stories collected during the requirements phase are entered into a database, called the *backlog*. At the beginning of each sprint, users stories to be implemented during the sprint are selected from the backlog. Work then begins by next designing the system to meet these selected requirements. A variety of design methods exist, ranging from informal, such as user-interface mockups, to semi-formal, such as Unified Modeling Language (UML) diagrams (Fowler and Scott 2000), to formal methods, using mathematical models, simulation, formal logic and proof assistants (Chlipala 2013). Semi-formal methods, using UML, are the most widely used in practice currently.

After the design is ready, implementation can begin. This is where computer programming, unit testing and debugging comes into play. It may be appropriate to quickly implement a first version of the system, using *rapid protoyping* with high-level programming languages. After the prototype has been validated the system may be reimplemented for efficiency reasons, if necessary, using a lower-level programming language.
Often, however, this optimization effort is not worth the required effort.

The sprint is not complete until the new version of the system has been verified, validated and documented. Verification is the process of checking whether the system correctly implements the design. When proof assistants are used to design the system using formal methods, verification is assured, since these assistants provide tools for automatically generating correct implementations from the formal specifications. Validation is the process of checking whether the implemented system meets its requirements. This is a more open issue, which cannot be completely automated. The acceptance tests developed during the requirements analysis phase facilitate a somewhat more objective evaluation, since the tests were developed *before* the system was designed or implemented. Without acceptance tests, it might be difficult to resist the temptation to develop validation criteria which ones knows are satisfied by system. Thus acceptance tests play a role in software engineering similar to double-blind and other safeguards of experimental science methodologies.

[1] Interestingly, the template for users stories is similar to the argumentation scheme for value-based practically reasoning.(Atkinson and Bench-Capon 2007)

Discussion

Let me now try to address the issues raised in the workshop synopsis, from the perspective of software engineering.

The first question is whether legal argumentation requires its own research goals and methods. Surely legal argumentation has its own goals. One of the purposes of requirements analysis is to identify and articulate these goals. What kinds of actors (roles) use legal argumentation and for what purposes? What are the interests of the actors in these various roles and what are they trying to achieve? But does research on legal argumentation require its own methods? Or is it sufficient to apply existing methods from the various scientific and engineering discplines participating in the study of legal argumentation? From a software engineering perspective, legal argumentation is an application domain. I would say that the need for particular methods has yet to be articulated or demonstrated. Research on legal argumentation in the field of AI and Law has applied methods from law and computer science, including the software engineering methods summarized here. AI and Law has developed innovative theories about legal reasoning, and methods and tools to facilitate legal reasoning, but has not, to my knowledge, developed new research methodologies. Since the field is interdisciplinary, some methods may seem new and innovative to some people working in field, because they are methods from other disciplines outside of their own.

The second question asks which methodological ideas, if any, from AI and Law, philosophy of argument and legal theory may be inspiring or instructive to the other disciplines. Again, AI and Law is itself an interdisicplinary field of study, like legal argumentation. Thus I would prefer to reformulate the question to ask what computer science, philosophy and legal theory can learn from each other regarding methodologies. The legal domain has been a valuable source of problems, examples and requirements for computer scientists working on computational models of argument in the field of Artificial Intelligence and Law. But it is doubtful whether computer science has been inspired by methods from other disciplines working in the field, such as legal theory. More generally, I think each discipline involved in the field has tended to focus on applying its own methodologies to the subject matter, without tying to evolve these methodologies further due to influences from other disciplines participating in the field. On the contrary, disciplines tend to be very conservative about perserving their methodologies. Indeed, disciplines tend to define themselves as much by their particular methodologies as by their domain or object of study. Presumably this is also a consequence of the various aptitudes and competencies required by particular methodologies, which tend to attract people to one discipline rather than another. Lawyers, for example, tend to have strong natural language skills, as required in all humanities disciplines, while abhoring mathematics and formal analytical methods. The reverse is true for computer scientists, typically. But the question may be understood as asking not whether the disciplines have in fact been inspired by methodologies of other disciplines in the field, but whether some discipline has methodological ideas which could inspire some other discipline. I am skeptical about the potential to export methodologies from one discipline to another, mainly due to the vastly different sets of skills and competencies required to learn and apply these methodologies. I am more optimistic about the potential of interdisciplinary collaboration, where each discipline applies its own methodologies.

The third question is whether legal argumentation should be analyzed from first principles, top-down, or begin with more concrete, real-life examples, bottom-up. Software engineering offers a third, intermediate path. The roles, personas, user stories and scenarios defined during requirements analysis are neither abstract principles nor concrete examples. Both principles and examples may be used during requirements analysis, but the resulting requirements are more specific than principals and more generic than examples. Perhaps this is a strength of the software engineering approach. It provides a kind synthesis, integrating both general principles and concrete examples.

The fourth question asks about the relative benefits of formal and informal methods for studing legal argumentation. Software engineering applies a variety of methods, ranging from informal to formal. Different methods are appropriate for different tasks. Informal or semi-formal methods facilitate collaboration between computer scientists and domain experts, especially when the domain experts are from disciplines, such as law, with an adversion to formal methods. Formal methods are useful, even necessary, for proving properties of designs (specifications) and verifying the correctness of implementations. But formal methods are less useful, if at all, for validating that the resulting system meets actual requirements. Presumably this general principal applies also to the study of legal argumentation. Both informal and formal methods serve important functions. Neither can supplant the other.

The fifth question asks how the presumptions of the disciplines participating in the field of legal argumentation determine the scientific method and obtained results of the field. I would disagree with what seems to be a presupposition of this question, that the interdisciplinary field of legal argumentation has a scientific method derived from the methodologies of its participating disciplines. Rather, each discipline applies its own methodology, separately. Surely, the results obtained are affected by the methodologies applied. The methodology of a discipline is based on its own epistimological presumptions. Whatever the strengths of these presumptions, they also lead to a kind of blindness, which make potentially interesting results impossible to find.

The sixth and final question asks about descriptive (empirical) and normative models. Physics is the prototypical empirical science and logic the prototypical normative science. But both strive for universal truths, not solutions to practical problems. Conversely, software engineering and the humanities, including the law, have in common the practical goal of helping people to find their way in the world, to achieve their goals, satisfy their interests, and lead a meaningful life, not just to describe the world or articulate universal truths. Software engineering applies and integrates both normative and empirical methods. The choice of methods is based on their effectiveness for achieving goals and satisfying requirements. Which methods are appropriate depends on the task. Formal methods, which are normative, are needed to simulate and verify system designs, to derive or prove their properties. Empirical methods are needed to capture requirements and to validate whether systems meet these requirements. But these empirical methods have a normative side: requirements are identified in collaboration with the "customer". And acceptance tests are negotiated with the same customer. The acceptance tests are similar to a contract. Indeed, they may actually be part of a contract. Requirements and acceptance tests express normative conditions which the system must satisfy. If the system is a work for hire, failure to satisfy these conditions may be a breach of contract. Whereas in scientific disciplines, such as physics and logic, the normative standards are set by "high priests" of the discipline themselves, in engineering fields, such as software engineering, the standards are ultimately set by the market, by users and customers. Interestingly, these two classes of arbiters can converge in the field of legal argumentation, when philosophers or legal theorists play the role of the customer, defining the requirements and acceptability tests, and software engineers apply the tools of their trade to develop models and systems meeting these requirements and passing these tests.

References

Atkinson, Katie, and Trevor J. M. Bench-Capon. 2007. "Practical Reasoning as Presumptive Argumentation Using Action Based Alternating Transition Systems." *Artificial Intelligence* 171 (10-15): 855–74.

Chlipala, Adam. 2013. *Certified Programming with Dependent Types – a Pragmatic Introduction to the Coq Proof Assistant.* MIT Press.

Fowler, Martin, and Kendall Scott. 2000. *UML Distilled – A Brief Guide to the Standard Object Modeling Language.* 2nd ed. Addison Wesley Longman, Inc.

Leffingwell, Dean. 2010. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise.* Addison-Wesley.

Sartor, Giovanni. 2005. "Legal Reasoning: A Cognitive Approach to the Law." In *A Treatise of Legal Philosophy and General Jurisprudence*, edited by E Pattaro, H Rottleuthner, R A Shiner, A Peczenik, and G Sartor, 5:844. Springer.

Sims, Chris, and Hillary Louise Johnson. 2011. *The Elements of Scrum.* Dymaxicon.