A Theory Construction Approach to Legal Document Assembly

Thomas F. Gordon[1]

German National Research Center for Computer Science (GMD)
Institute for Applied Information Technology
Expert Systems Research Group
Sankt Augustin, West Germany

thomas@gmdzi.usenet

Key Words. Document Assembly, Natural Deduction, ATMS, Default Reasoning.

Abstract. An overview of a software system under development for "assembling" legal documents is presented. The system applies Artificial Intelligence (AI) methods and is founded on a *theory construction* or *abduction* view of legal reasoning. The AI methods employed include an Assumption-based Truth Maintenance System (ATMS), a Natural Deduction theorem prover, and an implementation of Poole's approach to default reasoning.

Legal Document Assembly

There has been very little published work on the application of Artificial Intelligence (AI) technology to the problem of constructing legal documents. The bulk of the legal expert systems literature is concerned "only" with the problem of supporting the task of deciding legal cases, i.e. of identifying and analyzing the legal issues raised by the facts of some case. With the exception of the German KOKON project [Kowalewski 86], where an attempt was made to represent legal rules using Horn clauses, I am aware of no previous attempt to develop an integrated approach to legal analysis and documentation using AI methods. The work described here is one attempt to fill this gap.

The usual approach to "assembling" legal documents is *procedural*. James Sprowl developed the seminal system of this type, the ABF Processor [Sprowl 80]. Essentially, the ABF Processor is a special-purpose imperative programming language for writing programs for drafting documents of a certain type. Sprowl's achievement was in making the ABF Processor especially "lawyer-friendly". In principle, of course, such programs can be written in any general-purpose programming language, such as Lisp or Pascal, but these general purpose programming languages demand a great deal more programming expertise from the lawyer/programmer; the required data types and operations must first be constructed out of lower-level facilities. Moreover, Sprowl's language has the attractive feature that expressions and commands of the language are embedded directly in the text segments to be used, which greatly increases the comprehensibility of the resulting programs.

Sprowl published his paper on the ABF Processor in 1979, but only recently have such document assembly systems begun to be used in actual legal practice. The ABF Processor itself is now commercially available. Similar programs are available,

---

[1]The work reported here was conducted as part of the Assisting Computer (AC) project of GMD's Institute for Applied Information Technology. I would like to thank my colleagues Joachim Hertzberg, Alexander Horz and Hartmut Bewernick for their assistance and encouragement.

however, from other sources, such as the Document Modeler package from a Canadian software house called, interestingly enough, LegalWare.

Despite its recent commercial success, Sprowl's approach to legal document assembly suffers from a variety of problems. The content of a legal document is obviously dependent on an adequate legal analysis of the facts and relevant law. In the ABF Processor, the knowledge required to assist the lawyer in performing these legal analysis and reasoning tasks must also be brought into procedural form. A great deal of progress in the AI fields of knowledge representation and reasoning has been made since Sprowl designed his system. This procedural approach to legal reasoning no longer reflects the state of the art. AI methods for constructing software systems for supporting legal reasoning offer potentially several advantages, including:

• **Declarative Knowledge Representation.** Laws as they are expressed in statutes and cases are not represented as procedures for determining legal consequences, but as *definitions* of legal concepts and *rules* defining relationships between such concepts. By allowing interpretations of the law to be represented in a manner closely related to the way laws are represented in natural language, declarative knowledge representation languages make it easier for lawyers to express their interpretation of the law. (To be fair, I should mention that Sprowl did a very good job of disguising the proceduralness of his programming language: In simple cases, at least, procedures appear to be expressions in a version of Layman Allen's method for *normalized drafting* [Allen 57], which is primarily a syntactic variant of propositional logic.)

• **Explainability.** Systems which support legal reasoning must be capable of explaining the legal conclusions made by listing, for example, whatever assumptions were made and explaining why some particular interpretation of the legal sources was adopted. Users of legal document assembly systems are not well served by systems which are incapable of explaining their behavior. For example, an article in a recent issue of California Lawyer, a magazine published by the California Bar Association, describes some of the difficulties caused by programs for calculating child and spousal support which are not capable of producing adequate explanations [Kroll].

• **Default Reasoning.** Legal rules are usually formulated as general rules subject to exceptions, where the exceptions are often stated separately in other paragraphs. Moreover, due to the open texture of legal concepts, exceptions are often discovered by the courts in the course of deciding particular cases. The procedural approach assumes that a correct and complete decision procedure can be formulated for the relevant area of law. This is rarely possible. Moreover, the resulting procedures no longer reflect the original structure of the law; separate general rules and exceptions are *collapsed* into decision trees.

• **Reasoning in Multiple Contexts.** Legal reasoning is not primarily a deductive task. That is, legal reasoning is not merely a matter of applying pre-compiled rules to the facts of a case. Rather, a major part of a lawyer's responsibility involves creatively interpreting primary legal sources (statutes and cases) to discover new interpretations supporting arguments favorable to the position of his client. This interpretation process in not unconstrained. The resulting arguments must satisfy a variety of requirements, including such technical restrictions as being logically coherent. Legal reasoning can be viewed as exploration through the space of interpretations of the law and facts of the case. Because of the open texture of legal concepts, there is no practical way to automatically *generate* all possible interpretations. Thus there can be no complete method for automatically *searching* the space of interpretations. (Of course, even if the space of interpretations could be generated, its size is so large that no complete method would be practical.) Nonetheless, AI methods, in particular reason maintenance systems, can support a variety of bookkeeping chores necessary to search the space of technically defensible arguments effectively.

The only legal document configuration system I am aware of which applies AI methods is the KOKON system. However, KOKON, at least as it is described in [Kowalewski 86], does not appear to support default reasoning or reasoning in multiple contexts. Its principle contribution was to apply the logic programming approach to legal reasoning, as described for example in [Sergot 86], to the problem of legal document assembly. Thus, subject to the limitations of Horn clause logic for the purpose of legal reasoning [Gordon 87], KOKON does allow interpretations of legal sources to be represented declaratively, and does, to a limited but useful extent, explain its reasoning.

System Overview

The document assembly system we are developing addresses the limitations of KOKON by supporting default and multiple context reasoning. In this section, an overview is sketched out of the complete system, including those components required for editing and printing documents. A principal feature of the system is that the same methods used to construct legal arguments are applied to the problem of generating legal documents.

Figure 1, below, shows a functional diagram of the system. The shadowed boxes represent *processes*; the other boxes represent *data*. (For the sake of simplicity, the user is also represented as a process.) There are three main software modules: the theory construction module, *Reasoner*, the text generation module, *Generator*, and a structure-oriented text editor, *Editor*. Each of these components are briefly described in the following sections.

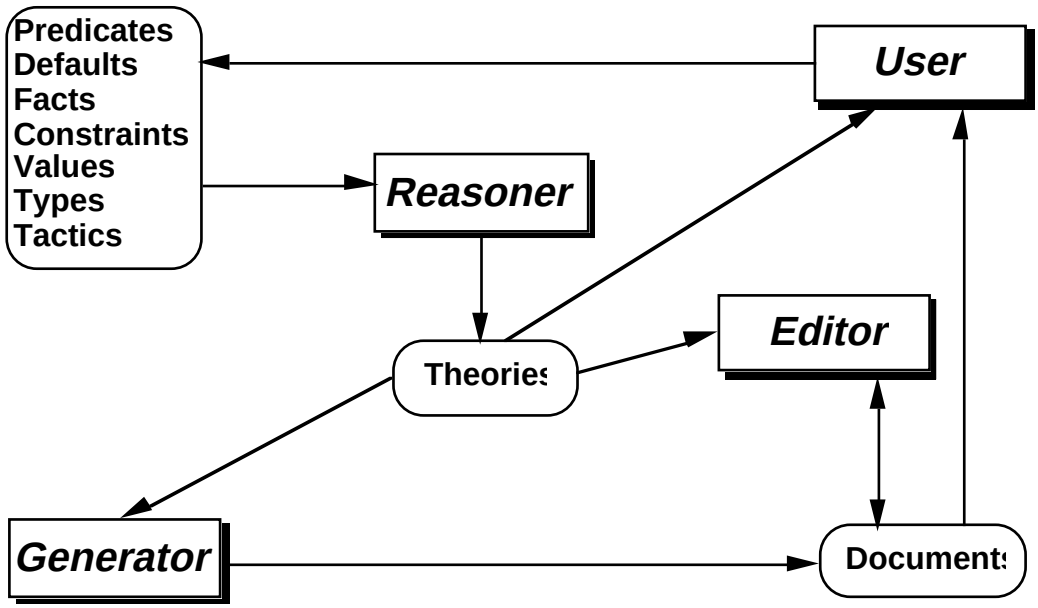

Figure 1. System Overview

The Reasoner

At the last Logica, Informatica, Diritto conference, in 1985, Herbert Fiedler of the GMD presented a *constructive* view of legal reasoning and contrasted this view with the *deductive* view adopted for legal expert systems [Fiedler 85]. The perhaps most well-

known deductive expert system is Marek Sergot's British Nationality Act program [Sergot]. Thorne McCarty's recent work, although based on intuitionistic rather than classical logic, also adheres to this approach [McCarty 88]. Among McCarty's previous work, his *prototypes* and *deformations* theory of case-based reasoning may be viewed as constructive in the sense intended here [McCarty 81]. Although she did not describe her system in these terms, Anne Gardner was the first person to my knowledge to *implement* an AI system based on a constructive approach to legal reasoning [Gardner 87]. The legal reasoning subsystem of the document assembly described here presents an alternative to Gardner's approach.

Fiedler's paper on legal reasoning as a design process, and his vision of a computer system to support this process analogous to CAD systems for engineers, was quite abstract. The reasoner to be described here is our second attempt to realize Fiedler's vision, at least in part, in a concrete system. The first system, the Argument Construction Set, was implemented as a *Diplomarbeit* by Karsten Schweichhart [Schweichart 88]. It used a truth maintenance system similar to Doyle's [Doyle 79], and a knowledge representation language based on Horn clauses, with *ad hoc* extensions for default reasoning.

Our current design is an integration of three main components:

1) an implementation of de Kleer's Assumption-based Truth Maintenance System (ATMS) [deKleer 86];

2) a theorem prover for a many-sorted predicate logic with equality;

3) an implementation of Poole's approach to default reasoning [Poole 88], constructed using the other two components.

In [Gordon 89] I describe how to use our system to spot legal issues. In that paper, our view of legal reasoning, de Kleer's ATMS and Poole's approach to default reasoning are described in somewhat greater detail than there is room for here. In the sections on multiple context reasoning and default reasoning, below, I demonstrate how these features can be useful in the context of a document assembly system.

The Document Generator

Documents can be described and processed at various levels of abstraction. At the lowest level, a *page description language*, such as Postscript, is used to draw images, including text, onto the page. At the next level, a *procedural formatting language*, such as TeX or Unix's nroff, is used to arrange blocks of text into lines, columns, paragraphs and pages, and allows such character attributes as font, style and size to be selected. A WYSIWYG "what you see is what you get" editor allows these attributes of text to be selected in an intuitive fashion, but operates at this layout-oriented level of abstraction. At the next higher level, the so-called "logical" structure of the document can be described. Here the various elements or parts of a document are marked with *tags* according to the function or role they play in the context of the particular document type. In a memorandum, for example, sections could be marked as being the *sender, recipient, date, subject* and *body* of the text. In a paper submitted to a scientific conference, there would be elements for the *author, title, abstract*, and so on. Languages for representing documents at this level include LaTeX, and the fairly new international ISO standard, SGML (Standard Generalized Markup Language) [Bryan 88].

It is this logical level that is of principal interest to us when assembling legal documents. However, we will not be using LaTeX or SGML directly to reason about or construct documents. Rather, we will be using predicate logic for this purpose. Although there are other possibilities, we have chosen to represent documents as *terms* in first-order predicate logic. Our approach to representing "boilerplate" text using logic is described in

greater detail below. Here I would just like to describe the function and purpose of the document generating subcomponent of our approach:

Given a document represented as a logical term, the process of getting this document printed is somewhat complicated. The first task of the document generator is to translate this term into the equivalent SGML document. Using an SGML parser, the SGML document is then translated into a TeX document, which is then processed by TeX and printed, or displayed on the screen, in the usual way.

The term and SGML representations of a document are both are at the logical level. It is possible, of course, to translate terms representing documents directly into TeX, or some other formatting language. But in contrast to our term representation, SGML is an international standard for exchanging and processing structured documents. There exist convenient tools for mapping SGML documents into formatting languages. There are also structure editors for creating and modifying SGML documents in a WYSIWYG manner. Translating terms into SGML is easier than recreating such tools.

Although we intend to use TeX, any formatting language can be in principal used instead. A version using Microsoft's RTF (Rich Text Format) would allow MS Word to be used as a WYSIWYG editor (although primarily at the layout level of abstraction). So long as convenient WYSIWYG SGML editors which hide the details of TeX are not readily available, RTF may more appropriate than TeX in a law office environment, but TeX is more convenient for us and is satisfactory for the purposes of our prototype.

The Editor

The editor will be a structure-oriented WYSIWYG SGML editor respecting and using the markup-tags in the SGML documents produced by the text generator. There are commercial SGML editors of this type. Notice, however, the arrow from the *Theories* box to the *Editor* in Figure 1. This is meant to suggest that the editor would, ideally, not only enforce the *syntactic* restrictions of the type of document being edited, but also the *semantic* restrictions contained in the theory created for the document. That is, the editor should prevent the user from violating semantic constraints or dependencies between the text segments used in the document. For example, if the document contains a column of numbers with their total at the bottom, the user should not be able to change the numbers without also updating the total. The user should be able to override these restrictions and edit text freely if he chooses to do so, but the system should be able to check the semantic integrity of the document at the level used by the reasoner during document construction.

Rather then ignoring semantic constraints by freely editing text, the user is expected to assert additional *facts* into the database used by the reasoner and then generate a new version of the document compatible with this additional information. Ideally, the system would allow a document to be incrementally modified in this fashion, without requiring a completely new version of the document to be generated.

Implementing an editor of this type is not a trivial task; it would require that we implement a more tightly-coupled editor providing services at each of the levels of document abstraction, from the logical level to the drawing of characters on the screen. We do not have the resources to take on a programming task of this scale in our research group, so we will have to be content with an SGML WYSIWYG editor; the syntactic restrictions of the document type will be enforced by the editor, but the user will remain responsible for ensuring that semantic constraints are respected.

As is usual for document assembly systems, our system will not use AI methods for generating fluent natural language. Rather, as will be discussed below, the system constructs documents out of “boilerplate” text building blocks. This approach is well-understood and easy to implement, but has the disadvantage that documents produced are

often awkwardly formulated. This is especially true for a language like German, where, for example, the gender of a noun can have consequences throughout a sentence. The proper suffix of adjectives, e.g., depends on the gender of the noun being modified. The usual way to circumvent this problem is to "beautify" the document using an editor in the final phase before printing. Because our system constructs documents using a logical level of document description, we can offer an alternative approach: the term structure of the document can be viewed as an *outline* of the document, rather than as a representation of the concrete choice of words or sentences in the final document. The outline describes *what* the various sections of the document should say, and their interrelationships, but does not determine exactly how the meaning of each section is to be expressed. According to this perspective, the only function of the boilerplate text segments is to express the intent of each segment in some canonical, if awkward, form. Thus, we could substitute an *explanation* of the purpose and intend of each building block for the boilerplate text and leave it up the user to formulate the actual sentences or paragraphs.

Our system will support both approaches: there will be boilerplate text segments which the user can apply (and then edit) if desired, but we also envision an outline processor mode for the editor. As is usual for outline processors, the user will be able to browse through the outline by expanding and collapsing nodes. In addition, there may be a comment associated with each node explaining what the node should express.

Representing "Boilerplate" in Logic

Figure 2 is an English translation of the header of a hypothetical petition for divorce. As the form of such petitions is fairly standardized and routine, this is surely a suitable application for a form book. In fact, there is a comprehensive form book in Germany for preparing applications for divorce [Verspermann 83]. The first step in preparing a form book is to identify the structural elements of the particular document type. Figure 3 shows some of the elements of this header, such as the address of the desired court, the occupation and name of the applicant, and so on. The primary problem at this stage is deciding which parts of the document belong to the background, i.e. to every instance of this document type, and which parts are to be abstracted out of this particular document so as to be allowed to vary from document to document. That is, the parameters or fields of the boilerplate text segments need to be selected. In our example, e.g., we have decided to leave the "District Court - Family Court -" and "PETITION F" parts of the document constant, as we are only interested in generating petitions for divorce, not a broader class of petitions.

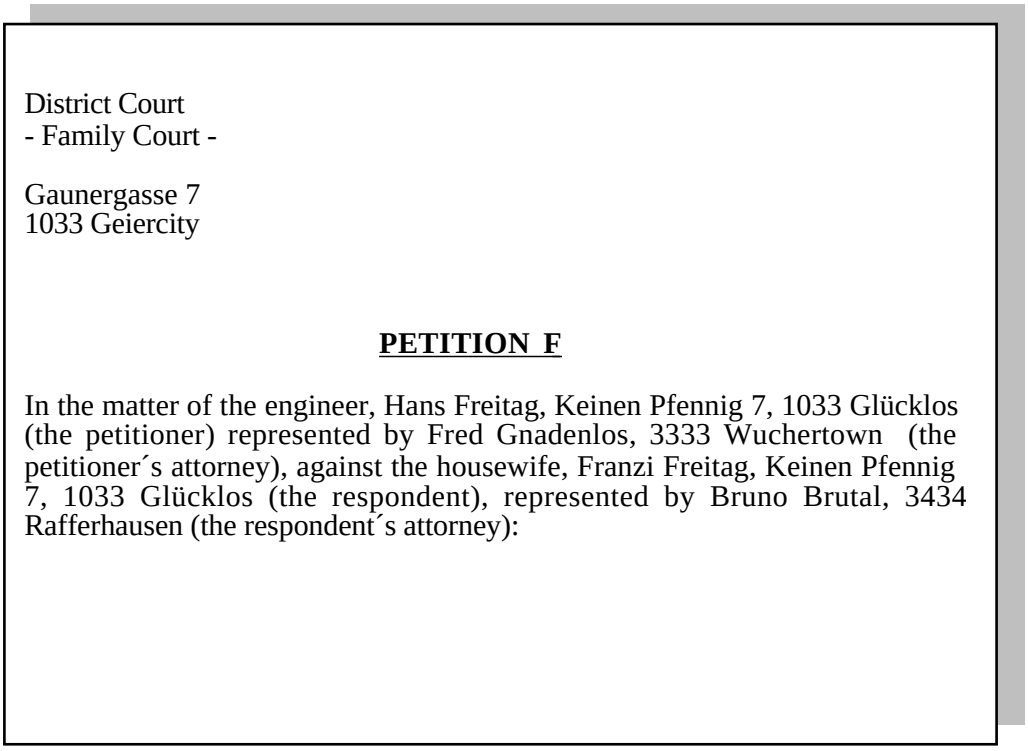

Figure 2. Petition F

Once these elements have been selected and identified, we could then use SGML to define the syntactic structure of such a header:

```
<!ELEMENT header -- (court, petitioner, pattorney,
                     respondent, rattorney)>
<!ELEMENT court -- (street, city)>
<!ELEMENT street -- CDATA>
<!ELEMENT city -- CDATA>
<!ELEMENT petitioner -- (occupation, name, address)>
<!ELEMENT occupation -- CDATA>
<!ELEMENT name -- CDATA>
<!ELEMENT address -- (street, city)>
<!ELEMENT pattorney -- (name, city)>
<!ELEMENT respondent -- (occupation, name, address)>
<!ELEMENT rattorney -- (name, city)>
```

It is not necessary to understand the details of this SGML code. Basically, these element declarations define a grammar for the headers of divorce applications. A `court`, e.g., is defined to be `street` followed by a `city`. `CDATA` is just a SGML key word meaning that arbitrary character strings can appear at that point in the document.

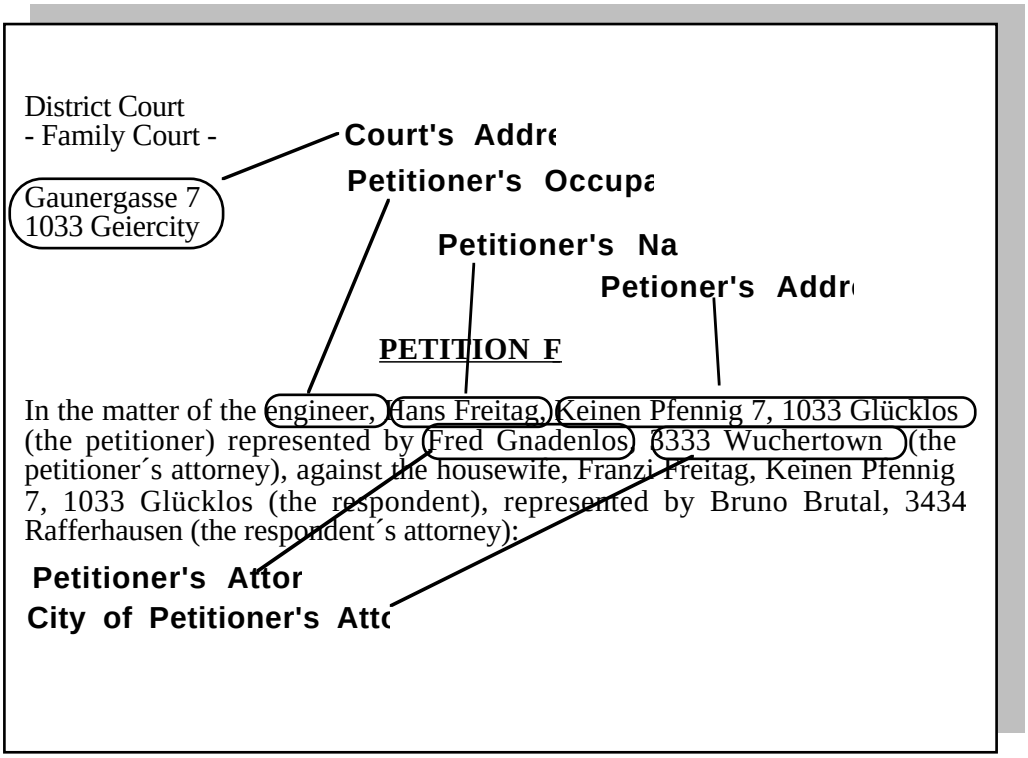

Figure 3. Some structural elements of a petition.

Using these SGML elements, we could represent the above example petition in SGML as follows:

```
<header>
<court> <street>Gaunerstrasse 7</street>
        <city>1033 Geierkirchen</city> </court>
<petitioner><occupation>engineer</occupation>
            <name>Hans Freitag</name>
            <address><street>Keinen Pfennig 7</street>
                     <city>1033 Glücklos</city></address>
</petitioner>
<pattorney><name>Fred Gnadenlos</name>
           <city>3333 Wucherstadt</city></pattorney>
<respondent><occupation>housewife</occupation>
            <name>Franzi Freitag</name>
            <address><street>Keinen Pfennig 7</street>
                     <city>1033 Glücklos</city></address>
</respondent>
<rattorney><name>Bruno Brutal</name>
           <city>3434 Rafferhausen</city></rattorney>
</header>
```

Although it may be somewhat difficult to read at first, the structure of this SGML document is quite simple; each piece of text is marked with a *start tag* and and *end tag* denoting the beginning and end of a particular element in the document. These tags are nested according to the grammar provided in the set of SGML elements given previously. An SGML parser would be able to check that this particular document indeed satisfies the grammar for headers. Notice that none of the background text common to all divorce petitions appears in this SGML version of the petition. The background text must be provided by the program which maps SGML documents of type *header* into the particular formatting language being used, such as TeX.

SGML allows us to describe the syntactic structure of a class of documents and to create (by hand) instances of these document types but it does not adequately support the construction of libraries of boilerplate text. Certainly, the header document type can be viewed as a form document, where we use instances of the type to "fill in the blanks". But what if, e.g., the attorney for the petitioner, Fred Gnadenlos, would like to create a customized version of the form in which the relevant information about his law practice always appears in the proper location? Using SGML alone, we would have to redefine the grammar for headers, removing the element for the petitioner's attorney. The information about Fred's practice would then be included into the mapping program which translates instances of this new document type into instructions in the formatting language, along with all the other background information, such as the "PETITION F" title. This approach would also suffer from the negative consequence that instances of this new document type are no longer instances of the general document type for headers.

We take another approach. Rather than creating a large number of SGML document types for each use of some class of documents, the SGML document type definition should be broad enough to encompass all legitimate instances of the class of documents of interest, whether created by hand, or assembled from predefined boilerplate text or some combination of both. Continuing with our example, the SGML definition should be liberal or broad enough so that all divorce petitions that would be acceptable by the courts can be defined as instances of the type, irrespective of the particular case or lawyer. Drawing the line between which parts of the document should belong to the background, and therefore factored out of the document type definition, and which parts should be allowed to vary from document to document, is a decision that needs to be addressed when designing the particular document assembly application.

How then do we propose to represent boilerplate text? We leave SGML now and turn to logic. Briefly, we suggest representing document types as *sorts* in a many-sorted predicate logic. Document instances of these types are then represented as terms in this logic. Finally, boilerplate text segments are represented as *functions* on such terms. Let us reconstruct the SGML code above using this approach and then create a boilerplate text segment. We require a notation for defining sorts. Let us use a syntax similar to the functional programming language ML's syntax for defining data types [Harper 88]. If `string` is a pervasive sort, then the header sort can be defined as follows:

```
sort street = street of string;
sort city = city of string;
sort name = name of string;
sort court = court of street * city;
sort address = address of street * city;
sort occupation = occupation of string;
sort petitioner = petitioner of occupation * name * address;
sort respondent = respondent of occupation * name * address;
sort pattorney = pattorney of name * city;
sort rattorney = rattorney of name * city;
sort header = header of court * petitioner * pattorney *
                        respondent * rattorney;
```

Using these sorts, we can represent our example header with the following term:

```
header1 =
    header(court(street "Gaunergasse 7",
                 city "1033 Geirercity),
           petitioner(occupation "engineer",
                      name "Hans Freitag",
                      address(street "Keinen Pfennig 7",
                              city "1033 Glücklos")),
           pattorney(name "Fred Gnadenlos",
```

```
                city "3333 Wucherstadt"),
        respondent(occupation "housewife",
                   name "Franzi Freitag",
                   address(street "Keinen Pfennig 7",
                           city "1033 Glücklos")),
        rattorney(name "Bruno Brutal",
                  city "3434 Rafferhausen"));
```

The relationship between this representation and the previous SGML version should be clear. Obviously, it is not difficult to translate such terms into an SGML equivalent. To create a piece of boilerplate text for Fred's practice from this header, we use functional abstraction. Using a typed lambda calculus notation for defining functions, we can represent Fred's header as follows:

```
FredsHeader =
    lambda c : court, p : petitioner,
           r : respondent, ra : rattorney .
    header(c,p,pattorney(name "Fred Gnadenlos",
                         city "3333 Wucherstadt"),r,ra);
```

Thus, with this technique we can create parameterized abstractions of particular terms (text segments) without having to modify the grammar of the document type. When applied to a particular (`court, petitioner, respondent, rattorney`) tuple, `FredsHeader` constructs an ordinary header.

Without any further features, we could use the approach described here to achieve the functionality of the typical document assembly system. A functional programming language, such as Standard ML, could be used to define document types and build a library of boilerplate text segments represented as ML functions. Documents would be generated by function application. Of course, the terms constructed would need to be translated into instructions in some text formatting language, perhaps by first translating them into SGML. The principal difficulty would be user-friendliness and designing a method for prompting the user for just the information required to fill in the "blanks". Lazy evaluation, together with a means of prompting the user for the values of undefined variables might be an interesting approach to try.

Such a functional approach to document assembly, however, would not be significantly more useful than the current generation of commercial document assembly systems. We are seeking additional functionality, not just alternative implementation approaches. Our goal is an approach to document assembly which complements our theory construction view of legal reasoning. A legal document reflects a particular analysis of the facts and relevant law. In commercially available document assembly systems, a particular interpretation of the law is "burned-into" the procedural code for assembling documents of a particular type. As there is little opportunity for the lawyer to reinterpret the law without reprogramming the document assembly application, there is a danger that lawyers, because of the economic realities of law practice, will neglect their responsibility to critically analyze and apply the law when using form documents or document assembly systems of the currently available type.

Multiple Context Reasoning

In our theory construction view of legal reasoning, the task of an attorney analyzing a case is to search through an ill-defined space of interpretations of the law and facts to find a view which maximizes his client's interests. Each constellation of interpretations defines a *context.* If contexts are represented using logic, as we propose, then a context is a set of logical formulae, usually called the *assumptions* of the context. Obviously, then, alternative contexts may have a number of assumptions in common. Because of the

monotonicity of standard logic, these contexts will also have implied or derived formulas in common. The principal job of an assumption-based truth maintenance system (ATMS) is to cache inferences made in one context, so that these inferences may be carried over to related contexts when we switch the focus of our attention to another context, in our case another interpretation of the, in our case and law.

What does this have to do with document assembly? Every legal document is subject to implicit assumptions about the interpretation of the facts and law upon which it depends. We propose to make these assumptions explicit, so as to allow the document to be incrementally modified when we shift our attention to another context, to another set of assumptions.

Returning to the example of the last section, suppose we are not sure about the address of the respondent. Let us assume, for whatever reason, that the husband and wife are still living together. We can make the dependency of our header on this assumption explicit with, e.g., the following context of formulae:

```
header(FredsHeader(C,P,R,RA)) <-
    court(C),
    petitioner(P),
    respondent(R),
    rattorney(RA).

court(…).
petitioner(petitioner(occupation "engineer",
                      name "Hans Freitag",
                      address(street "Keinen Pfennig 7",
                              city "1033 Glücklos"))).
respondent(respondent(occupation "housewife",
                      name "Franzi Freitag",
                      A)) <- petitioner(petitioner(_,_,A)).
rattorney(…).
```

Here I am using a Prolog-like notation for predicate logic. Uppercase letters denote universally quantified logical variables. The names of the various term constructors for the elements of a header have been "overloaded." The symbol `petitioner`, e.g., is used both as a functor and as a predicate. The Horn clause for `respondent` states that the address of the respondent is the same as the address of the petitioner. To generate the header in this context, we search for a solution to the goal `header(X).` In our example, at least, a Horn-clause backward-chaining prover such as Prolog could easily deduce a solution. The ATMS reason maintenance system makes it easy to retrieve this solution in all contexts which include these assumptions, which means it is unnecessary to expended resources recomputing this header each time when exploring related contexts in search of satisfactory formulations of other parts of the complete document.

Default Reasoning

In the last section, we showed how to make document assumptions explicit and discussed how to use an ATMS to facilitate the exploration of alternative interpretations of the underlying law and facts related to the document. We also demonstrated a kind *default reasoning*. Not knowing Franzi's address, we assumed --- by default -- that her address was the same as her husband's. However, *we* did the default reasoning, not the system.

Poole has developed an approach to default reasoning which fits in very well with our theory construction approach to legal reasoning [Poole 88]. According to his view, default reasoning does not require a special *nonmonotonic logic*. Indeed, he does not view default reasoning as a form of deduction at all. Rather, he considers default reasoning to be a form of theory construction or *abduction* from a set of assumptions or *hypotheses* that the user is willing to consider. Deduction plays an important, but subordinate role in this framework. In principal, the deductive component of the approach may use any logic. Poole however uses standard predicate logic for this purpose.

Due to space considerations it is not possible here to describe Poole's framework in any great detail. Rather, let me try to give a feeling for the approach by continuing with our example document assembly application. Suppose that our attorney friend Fred Gnadenlos usually appears before the family court in Geirercity. He would like this court to be used in his petitions unless he explicitly requests otherwise. It might be thought that we could just create a new piece of boilerplate text in which this court is used, just as we created Fred´s custom header above. Certainly we could do this, but this step alone would not achieve the functionality we are seeking. We would like our choice of the Geierkirchen court to appear in the set of assumptions supporting any documents generated using this assumption. We wouldn´t want Fred to forget that he only assumed the Geierkirchen court was appropriate. So let us try another approach. Let us define the following *default rule*, using Poole's THEORIST language for default reasoning [Poole 88]:

```
default d1 : court(court(street "Gaunergasse 7",
                         city "1033 Geirercity")).
```

Again, here I am using the symbol `court` both as a predicate and as a constructor function. Such defaults are just syntactic sugar for *formula schemata*. This particular schema is not particularly interesting, as it has no schema parameters. Thus there is exactly one instance of the schema:

```
d1 -> court(court(street "Gaunergasse 7",
                  city "1033 Geierkirchen"))
```

This material implication is intended to mean that if the default `d1` is applicable, then the applicable court is the usual family court in Geierkirchen. The pragmatic effect of this default is to suggest trying contexts in which this instance of the default and `d1` are assumed. To *cancel* the default, Fred need only assert the following premises or, in Poole's terminology, *facts*.

```
fact ¬d1.
fact court(court …).
```

Such "facts" are considered to be universally true in all contexts. Now, any context containing `d1` is inconsistent with these premises and would no longer be considered. As a consequence, the default court would not be used in subsequent drafts of the petition for this particular case.

Conclusion

What has been accomplished? A design of an integrated system for legal reasoning and legal document assembly has been sketched out. The system is an example of Fiedler's CAD approach to legal expert systems and applies a variety of recent AI technologies, including de Kleer's ATMS and Poole's framework for default reasoning.

Work on a prototype implementation of the system has begun, but there is still a great deal of programming remaining to be done. The prototype is being implemented in Standard ML under Unix as part of our `qwertz` project. The principal goals of `qwertz` are to reconstruct, evaluate and improve AI methods for planning and configuration tasks. The document assembly system described here is to be the first "application" of the methods developed in `qwertz`. I say "application", in quotes, as we are primarily concerned with furthering the state of AI methods, not with constructing text assembly systems. Its purpose is to provide a test bed for the methods were are developing.

One of our student assistants, Hartmut Bewernick, is developing an application of our approach in the field of German divorce law. The examples in this paper are drawn from his "knowledge base".

The system as it has been presented here requires a fairly powerful theorem prover for many-sorted predicate logic with equality and functions defined using lambda expressions. We are designing a Natural Deduction theorem prover with a programmable control component and the features required. However there are several problems which have to be addressed before the prover can be considered practical for this application domain. The most serious concern existential quantifiers. The domain of documents constructed out of recursively defined sorts is infinite, so it is unfeasible to iterate through the elements of the domain in search for documents satisfying certain predicates. Resolution-based refutation provers avoid this problem nicely using unification. We are attracted to Natural Deduction, however, because of an interesting relation between it and assumption-based reason maintenance. These issues will have to wait for another publication.

References

[Allen 57] L. E. Allen; *Symbolic Logic: A Razor-Edged Tool for Drafting and Interpreting Legal Documents* ; Yale Law Journal ; Vol. 66; 1957.

[Bryan 88] M. Bryan; *SGML, An Author´s Guide to the Standard Generalized Markup Language* ; Addison-Wesley; 1988.

[Doyle 79] J. Doyle; *A Truth Maintenance System* ; Artificial Intelligence; Volume 12; 1979.

[Gordon 87] T. F. Gordon; *Some Problems with Prolog as a Knowledge Representation Language for Legal Expert Systems* ; in Yearbook of Law, Computers and Technology; ed. C. Arnold; Butterworths; 1987.

[Gordon 89] T. F. Gordon; *Issue Spotting in a System for Searching Interpretation Spaces* ; International Conference on AI and Law (ICAIL) ; Vancouver; 1989.

[Fiedler 85] H. Fiedler; *Expert Systems as a Tool for Drafting Legal Decisions* ; II Convegno Internazionale Logica, Informatica, Diritto; Florence; 1985.

[Gardner 87] A. L. Gardner; *An Artificial Intelligence Approach to Legal Reasoning* ; MIT Press; 1987.

[Harper 88] R. Harper, Robin Milner, Mads Tofte; *The Definition of Standard ML, Version 2* ; LFCS Report Series; No. ESC-LFCS-88-62; University of Edinburgh; August, 1988.

[de Kleer 86] J. de Kleer; *An Assumption-Based TMS;* Artificial Intelligence; Volume 28; 1986.

[Kowalewski 86] D. L. Kowalewski, J. Schneeberger; *KOKON: Wissensbasierte Konfigurierung von Verträgen* ; GI-16. Jahrestagung; Springer-Verlag; Berlin; 1986.

[McCarty 81] L. T. McCarty and N.S. Sridharan; *A Computational Theory of Legal Argument* ; Rutgers Technical Report, LRP-TR-13; 1981.

[McCarty 88] L. T. McCarty; *Clausal Intuitionistic Logic, I. Fixed-Point Semantics* ; Journal of Logic Programming; Vol. 5; 1988.

[Poole 88] D. Poole; *A Logical Framework for Default Reasoning* ; Artificial Intelligence; Volume 36; 1988.

[Schweichhart 88] K. Schweichhart; *Das Argument Construction Set* ; Arbeitspapiere der GMD; No. 348; November, 1988.

[Sergot 86] M. J. Sergot, F. Sadi, R. Kowalski, R. A. Kriwaczek, P. Hammond and H. T. Cory; *The British Nationality Act as a Logic Program* ; Communications of the ACM; Vol. 29; 1986.

[Sprowl 80] J. Sprowl; *Automated Assembly of Legal Documents* ; in Computer Science and Law, An Advanced Course; ed. B. Niblett; Cambridge University Press; 1980.

[Vespermann 83] H. J. Vespermann; *Scheidungs- und Scheidungsverbundverfahren* ; Beck; Munich; 1983.