A Multi-Agent Legal Argument Generator

Mark Allen[1] Trevor Bench-Capon[2] and Geof Staniford[3]
LIAL - Legal Informatics at Liverpool

Abstract

One of the most significant series of experiments in AI and Law is the investigation of case based reasoning carried out in the HYPO, CABERET and CATO projects. It is important to understand what has been achieved in these experiments, and this requires that the techniques be applied to a variety of domains. The techniques are not, however, straightforward to apply. In order to provide an experimental environment for investigating the techniques further, we have re-implemented the central argument generation process as a set of agents written in JAVA using IBM's Aglet agent building framework. By using this environment it will be possible to explore the general applicability of the techniques. In this paper we give a summary of an algorithmic description of argument building in HYPO and CATO, and describe our implementation of the algorithm.

1. Introduction

In the foreword to their excellent book on the series of MYCIN experiments (Buchanan and Shortliffe 1984), the authors wrote:

> "In the present state of AI, it is all too easy to move on to the next system without devoting sufficient energies to trying to understand what has been wrought, and to doing so in such a way that adds to the explicit body of science." (p xiv).

One factor underlying the rapid assimilation of MYCIN ideas into the community was the existence of the expert system shell, EMYCIN, which allowed for easy application of the ideas to a variety of domains, permitting a greater understanding and a sounder basis for generalisation. But in general, their criticism remains true today. AI progresses through experiments, but even successful and important experiments need to be thoroughly understood and made easily replicable if they are to become part of the received wisdom of the subject. AI and Law provides some instructive examples. There have been several important and widely known experiments, not all of which have been sufficiently understood and exploited by the community.

In this respect two of the most famous examples provide an important contrast. Consider first the classic logic programming experiment of the British Nationality Act (BNA) (Sergot et al 1986). Working out "what has been wrought" - and what has not been wrought - here has been a major activity of logical programming inclined workers in AI and law ever since. HYPO, developed by Rissland and Ashley and best described in Ashley (1990), however, despite being the key program representing the CBR approach to AI and Law, has proved far more difficult for others to build on.

Partly this is because of intrinsic differences in the experiments: the BNA experiment was based on a rather simple hypothesis: that a legal expert system could be built by:

1) Representing the relevant legislation in executable logic;
2) Executing the representation using Prolog augmented by a "Query the user" (Sergot 1983) module.

Given the wide understanding of the formalism and the ready availability of the required environment, it was possible to reproduce the experiment in other domains, and reflect on the strengths and limitations of the approach. HYPO, on the other hand, applies some sophisticated and specific techniques, to a rather special representation. As a result, despite its influence, and the fact that its developers have refined it in separate directions (CABERET (Skalak and Rissland 1992), and CATO (Aleven 1997)) we have not seen the proliferation of HYPO-like developments that would help us to fully understand its strengths and limitations to the extent that we do with legal logic programs. One possible exception is the reconstruction of Prakken and Sartor (1998), which helps to explain the logic of the reasoning, but falls well short of providing

[1] Chester, College, Chester, UK.

[2] Department of Computer Science, The University of Liverpool, Liverpool, UK.

[3] Liverpool John Moores University, Liverpool, UK.

0-7695-0680-1/00 $10.00 © 2000 IEEE

tools to allow for experimentation in different domains.

In an attempt to improve this situation, one of the current authors provided in Bench-Capon (1997) a description of the underlying collection of algorithms used in HYPO (and the CATO extensions). This breaks HYPO's argument generation down into a series of components, each specified using pseudo-code. While this articulated the various mechanisms in HYPO, it does not immediately provide a readily usable tool for experimentation comparable to that supplied by Prolog for the logic programming approach. In this paper we therefore go a step further and describe a modular implementation of the HYPO/CATO algorithms, which we hope can be used to provide the required experimental vehicle.

Section 2 summarises the expression of HYPO/CATO as a set of algorithms. Section 3 describes the design of the implementation in terms of a multi-agent architecture. Section 4 gives some detail on the actual implementation. Section 5 offers some suggestions for future work.

2. Argument in HYPO

The aim of HYPO (and CATO) is to construct an argument which could be advanced concerning a new case, the various steps of which are supported by precedents taken from the case law of the domain (US Trade Secrets Law). In fact, these systems can produce arguments for either side, but in what follows I shall consider that the system is trying to argue for the plaintiff.

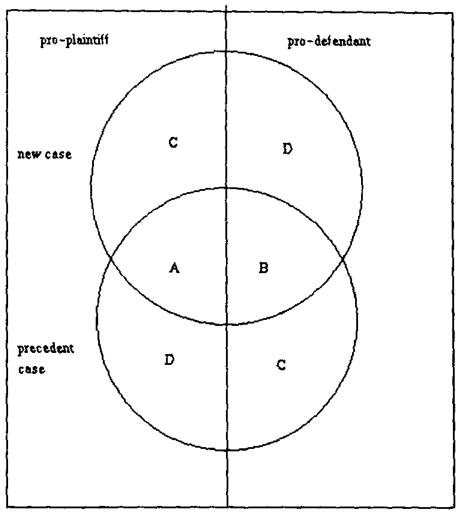
Cases are represented using factors, which are distinctions that can be drawn in a case and which favour one side or other. Factors can be one of three kinds. A binary factor is either true or false. A dimension has a value representing its strength and also has a *direction* (whether high or low values favour the plaintiff). Finally (introduced by CATO) there are abstract factors, which relates factors in a hierarchy, the children of a given abstract factor either supporting or militating against that abstract factor.

When past cases are compared with a current case, there are four possibilities, as shown in Figure 1.

Figure 1: classification of factors when comparing cases in HYPO

Depending on their classification according to the above scheme, factors have different effects:

- a-factors are pro-factors in common.
- b-factors are con-factors in common.
- c-factors make the new case *more* favourable.
- d-factors make the case *less* favourable.



Note that the new case is compared with the precedents on an individual basis so that, for example, an a-factor with respect to one case might be a c-factor with respect to another. The classification of the factors in this way is an essential input to the argument algorithm. Each of the various moves makes use of this classification. In HYPO's model an argument has three stages, called "plies".

1. *State Point.* The plaintiff analogises a precedent case to the current fact situation (CFS) and makes the claim that the court should find for them. The precedent must have some A-factors, and precedents can be ranked in order of support for the plaintiff according to the existence of other factors: The order is: AC, ABC, AB, A, ACD, ABCD, ABD, AD. (In general C factors strengthen the plaintiff's case, whereas D factors allow the defendant room for manoeuvre.
2. *Respond.* The defendant responds, either by citing a counter example (a past case at least as on-point, i.e. containing as many A and B factors, as that cited by the plaintiff) or by distinguishing the cited case, citing D factors. CATO also allows the distinctions to be emphasised, if appropriate, using the factor hierarchy.
3. *Rebut.* The plaintiff rebuts the response. This means distinguishing the counter example, (using C factors), showing any weaknesses are not fatal, (citing a case decided for the plaintiff where the D factors were present), and (in CATO), downplaying the significance of any distinctions and up playing any strengths, again using the factor hierarchy.

From this we can see that each of the plies contains a number of moves: in each case we have to establish whether the move can be made, and to find a suitable precedent case to licence the move. A full specification of all the moves is given in Bench-Capon (1997).

3. The Program

It was decided to implement the system using a multi-agent architecture. This was motivated mainly by a desire to use a highly modular design, to make experimentation with more, different, or even fewer, components readily possible. It also allows us to take advantage of any scope for parallelisation that may present itself.

We began modelling the system by identifying and assigning roles to each agent that would take part in the argument. This gave us:

- A *Solicitor*. This agent interfaces with the user, organises the argument, creates and communicates with other agents.
- A *Judge*. This agent accepts and displays parts of the argument. It also ensures that the argument follows the specified rules.
- Various "expert" agents. There is one such agent for each possible move in the argument, as identified in Bench-Capon (1997). For example there are *state point* and *counter example* experts. These build their own part of the argument and send it to the judge.
- A *Case Librarian*. This agent guards the case base and provides access to the cases. When presented with a new case, it creates a *contact agent*.
- The *Contact Agent* handles communications with the experts. This agent also carries out the classification and sorting. If the case base is large this agent may spawn a team of agents to classify the cases in parallel.
- A *Context Register*. This agent maintains a list of active agents in the local context. It also has a list of other context registers on other servers. The list of active servers is maintained through a cgi program an a web server. A context server's address is published as a property in its local context.
- *Finder agents*. Created by the solicitor to search out the *case librarian*.

The role model for the system is shown in Figure 2. Each Expert Agent is started by the *solicitor* and passed the data relating to the case under consideration. It then constructs its argument and send it to the judge. Where possible the experts act in parallel. For example the stages of the response of *distinguish* and *counter example* can be done in parallel.

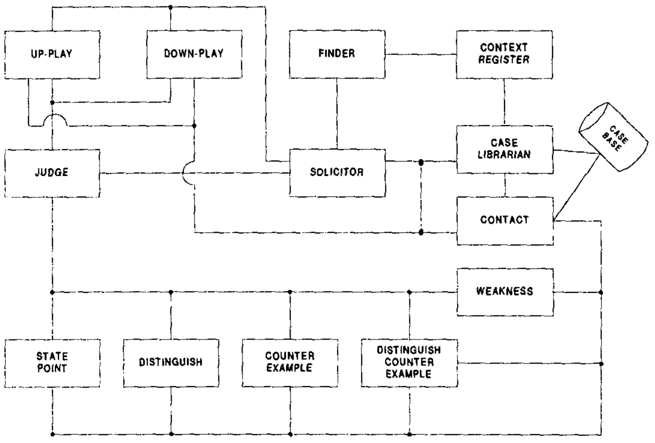


Figure 2 System Role Model

The system is written in Java using IBM's Aglet[4] agent framework (Lange and Oshima (1998)). This provides a class library supplying the framework for basic agent actions such as movement from one server to another, message passing, serialisation of data, and security (based on the standard Java model[5]). Java is the current standard language for agent creation. Used, for example, by Zeus[6], Jade[7] or JatLite[8]. This is because Java provides platform independence, secure execution, dynamic class loading, multi-thread programming, object serialisation, and reflection. The drawbacks include execution speed, and the lack of resource control – Agents can use up processor cycles and memory without limit. Also there is no reference protection – an agent cannot control what access another agent has to its public methods.

4. Example Agent

To give a flavour of the details of the implementation, in this section we will describe the coding of one of our expert agents. The agent we shall discuss is the *distinguish factors* expert. The role of this agent is to find any factors which can be used to distinguish the precedent from the case

[4] Aglet is a cross between Agent and Applet

[5] *Java Security* J. S. Fritzinger and M. Mueller Sun Micro Systems Inc http://java.sun.com/docs/white/index.html

[6]Zeus http://www.labs.bt.com/projects/agents/zeus/index.htm

[7] Jade http://sharon.cselt.it/projects/jade/

[8] JatLite http://java.stanford.edu/

under consideration. Essentially these are pro-plaintiff factors present in the precedent and absent from the current case, or pro-defendant factors present in the current case but absent from the precedent (what we call *d-fators*). This agent will also emphasis the distinctions if appropriate.

This agent is created by the *solicitor* agent. It extends the Expert class, which contains the base information required by all expert agents, including the addresses of the *judge* and *contact* agents and the new case. This data is passed through the `onCreation` method.

```
public void
onCreation(Object init)
   {
     Object[]
info=(Object[])init;

judge=(AgletProxy)info[0];

contact=(AgletProxy)info[1];
     newCase=(Case)info[2];
   }
```

The `init` object is cast into an array of objects. Each element of the array is then cast into the appropriate type. An `AgletProxy` represents the address of an agent.

The factor distinction algorithm looks for weaknesses represented by 'D' classifications.

```
If case is AD, ABD or ACD
Write citation for cited case
Write "is distinguished because:"
```

The first job is to get the cited case. This was passed to the *judge* by the *state point* expert.

```
   citedCase=(Case)
judge.sendMessage(new
Message("CITED_CASE"));
if(citedCase.tv.numD==0)
{
     sb.append("\nNo
Distinctions");
 }
else
{
sb.append("\n"+citedCase.cas
eName+" is distinguished
because:");
 }
```

The number of 'D' classified factors is stored in the `transient values` object of the case. If this is zero then no distinctions can be drawn and the move is ended. If there are weaknesses then we need to retrieve the "meta values" of the case type from the `contact` – the agent that holds the classified and sorted cases.

```
metaValues=(MetaValues)
contact.sendMessage(new
Message("META_VALUES"));
```

This object maintains information about aspects of the cases. For example how many factors there are, which factors are dimensions and which are abstract and the text relating to each factor. Now we loop through each factor in the case to check if it is 'D' classified.

```
  for(int
i=0;i<metaValues.numberFactors;i++
)
 {

if(citedCase.tv.classD.charAt(i)==
Case.SET)
{
if(metaValues.outcomes.charAt(i)==
Case.DEFENDANT)
{
  sb.append("\nin new case"
+metaValues.factorText[i]+". Not
so  in"+citedCase.caseName);
sb.append(emphasiseDistinction(i))
;
}
else
{
  sb.append("\nin
"+citedCase.caseName+"
"+metaValues.factorText[i]+". not
so in "+newCase.caseName);
sb.append(emphasiseDistinction(i))
;
}
}
}
```

The classD array in the transient values object records whether each factor is 'D' classified or not. If it is we check from metaValues whether the factor is pro-defendant or pro-plaintiff and then build the argument accordingly. This is a direct translation of the FD algorithm.

```
If D-factor is a d-factor
  Write "in" NC
  Write D-factor
  Write "Not so in"
  Write cited case
  Do ES
```

Once a 'D' factor is found and added to the argument string the "emphasiseDistinction" method is called with the factor number.

```
sb.append
(emphasiseDistinction(i));
```

This method implements the Emphasise Distinction Algorithm by searching for the highest abstract factor supported by the factor we wish to emphasise and which is not supported by any factor in NC.

The argument is appended into the StringBuffer "sb". When all binary factors have been completed the dimensions are distinguished (if possible). Then a Message object is created and the argument text set as a message argument. This is then sent to the *judge* for display.

```
sb.append(distinguishDimensions())
;
Message message=new
Message("DISTINGUISH_FACTORS");
```

```
message.setArg("TEXT",sb.toString(
));
judge.sendMessage(message);
```
On completion the agent destroys itself. Communication errors are handled with a *try and catch block*. If the message fails to arrive the Expert first establishes whether the Judge still lives. If it does it resends – if it does not it destroys itself, since if the Judge has failed then the argument will need to be restarted from the beginning.

5. Conclusions and Further Work

In this paper we have described the design and implementation of a system capable of generating arguments in the style of HYPO/CATO. The intention is to use this as an experimental vehicle to apply this style of argumentation generation to other domains. We believe that it is important to do this if we are to understand how generally applicable these techniques are, and to come to a better appreciation of the strengths and weaknesses of the approach.

We therefore intend to use the program in two new domains. The first domain will be a legal domain, namely the interpretation of the phrase "arising out of, and in the course, of employment" central to decisions on compensation for Industrial Injuries. As well as being interesting in its own right, this will enable comparison with another important case based reasoning system, Branting's GREBE (Branting 1989), which also operated in this domain. Secondly we would like to apply it also to a non-legal domain. Currently we have not fixed on a domain for this, but job interviews might be an interesting possibility.

When we have used the system in these two applications we will be in a position to assess the usability of the system for building HYPO style systems in different domains. If this is acceptable we would hope to make the tool available to other researchers. In this way a body of experimental data will be accumulated which we would expect to provide a good basis for coming to a verdict on the effectiveness and applicability of the techniques.

References

Aleven, V., (1997). *Teaching Case-Based Argumentation Through a Model and Examples* PhD Dissertation University of Pittsburgh.

Ashley, K.D., (1990). *Modelling Legal Argument: Reasoning with Cases and Hypotheticals.* MIT Press: Cambridge, Mass.

Bench-Capon, T.J.M., (1997). Arguing with Cases. In *Legal Knowledge Based Systems,* (Proceedings of the 10th JURIX conference), GNI, Nijmegen, pp 85-100.

Branting, L.K., (1989). *Representing and Reusing Explanations of Legal Precedents,* in the Second International Conference on AI and Law, ACM Press: New York, pp103-110.

Buchanan, B.G., and Shortliffe, E.H., (1984). *Rule Based Expert Systems,* Addison Wesley: Reading, Mass.

Lange, D.B., and Oshima, M., (1998) Programming and Deploying Java Mobile Agents with Aglets Addison-Wesley.

Prakken, H., and Sartor, G., (1998). Modelling Reasoning with Precedents in a Formal Dialogue Game. *Artificial Intelligence and Law,* Vol 6 Nos 2-4, pp 231-287.

Sergot, M.J. (1983). A query-the-user facility for logic programming. *In Integrated Interactive Computer Systems* (Degano, P., Sandewall, E., Eds). North-Holland, Amsterdam.

Sergot, M.J., Sadri, F., Kowalski, R.A., Kriwaczek, F., Hammond, P., Cory, H.T. (1986) The British Nationality Act as a Logic Program. *Communications of the ACM* 29, 5 (May 1986), pp 370-386.

Skalak, D.B., and Rissland, E.L., (1992). Arguments and Cases: An Inevitable Intertwiining. Artificial Intelligence and Law, Vol1 No 1, pp 3-42.