Automated Negotiation from Declarative Contract Descriptions

Daniel M. Reeves Michael P. Wellman
University of Michigan Artificial Intelligence Lab.
1101 Beal Avenue, Ann Arbor, MI 48109 USA
ai.eecs.umich.edu/people/{dreeves,wellman}/
{dreeves,wellman}@umich.edu

Benjamin N. Grosof
MIT Sloan School of Management
50 Memorial Drive, Rm E53-317
Cambridge, MA 02142 USA
www.mit.edu/~bgrosof/
bgrosof@mit.edu

ABSTRACT

Our approach for automating the negotiation of business contracts proceeds in three broad steps. First, determine the structure of the negotiation process by applying general knowledge about auctions and domain-specific knowledge about the contract subject along with preferences from potential buyers and sellers. Second, translate the determined negotiation structure into an operational specification for an auction platform. Third, map the negotiation results to a final contract. We have implemented a prototype which supports these steps, employing a declarative specification (in Courteous Logic Programs) of (1) high-level knowledge about alternative negotiation structures, (2) general-case rules about auction parameters, (3) rules to map the auction parameters to a specific auction platform, and (4) special-case rules for subject domains. We demonstrate the flexibility of this approach by automatically generating several alternative negotiation structures for a previous domain: travel-shopping in a trading agent competition.

1. INTRODUCTION

One form of commerce that can benefit substantially from automation is *contracting*, where agents form binding, agreeable terms, and then execute these terms. The overall contracting process comprises several stages, including broadly:

1. *Discovery.* Agents find potential contracting partners.
2. *Negotiation.* Contract terms are determined through a communication process.
3. *Execution.* Transactions and other contract provisions are executed.

In this work we are concerned with bridging these three stages, and primarily with the process by which an automated negotiation mechanism can be configured to support a particular contracting episode. We begin by presenting a shared language with which agents can define the scope and content of a negotiation, and reach a common understanding of the negotiation rules and the contract implications of negotiation actions. Note that we are concerned here with the definition of negotiation mechanisms and not the negotiation strategies employed by participating agents, though in designing a mechanism one must consider the private evaluation and decision making performed by each of the negotiating parties.

Our prototype system for automated contracting is called *ContractBot*.[1] By starting from a formal description of a partial contract—describing the space of possible negotiation outcomes—ContractBot automatically generates configuration parameters for a negotiation mediator (auction) platform. Then, by monitoring the individual auction results, it generates a final, executable contract.

Section 2 gives an overview of our approach to automated contracting. Section 3 provides background on auction-based negotiation. Section 2.3 frames the overall process of automated contract negotiation and shows how rules generated during the negotiation process can be combined with the partial contract to form an executable final contract. In Section 4, we discuss in detail how the language is used to infer parameters for configuring the negotiation—that is, parameters for a set of auctions-. We focus on a Trading Agent Competition [15] as an example domain (Sections 5 and 6). Finally, in Section 7, we discuss details of our prototype.

2. CONTRACTBOT FRAMEWORK

The central question in configuring a contract negotiation is, "What is to be negotiated?" In any contracting context, some features of the potential contract must be regarded as fixed, with others to be determined through the contracting process. At one extreme, the contract is fully specified, except for a single issue, such as price. In that case, the negotiation can be implemented using simple auction mechanisms of the sort one sees for specified goods on the Internet. The other extreme, where nothing is fixed, is too ill-structured to consider automating in the current state of the art.

Most contracting contexts lie in between, where an identifiable set of issues is to be determined through negotiation.

AGENTS'01, May 28-June 1, 2001, Montréal, Quebec, Canada.

[1] A complete version of the prototype, including rulesets required for the examples described in this paper, is available at `http://ai.eecs.umich.edu/people/dreeves/contractbot/`.

Naturally, there is a tradeoff between flexibility in considering issues negotiable and complexity of the negotiation process. But regardless of how this tradeoff is resolved, we require a means to specify these issues so that we can automatically configure the negotiation mechanisms that will resolve them. That is, we require a *contracting language.*

Our approach uses a form of logic-based knowledge representation to represent contracts and extends this language to express and reason about *partial contracts.* The partial contract, or contract template, describes possible negotiable parameters and how they are interrelated, along with meta-level rules about the negotiation and about individual auctions. It combines all this with rules from agents about their constraints and preferences over the possible negotiation structures. From implications of the rules, it generates the appropriate auctions and determines the auction parameters. Transactions in the auctions generate additional rules, which produce results for the final contract. As part of this new framework, our approach allows for reuse of the information in multiple stages of the contracting process.

2.1 Contracting Language

That our language must support all three stages of contracting (discovery, negotiation, and execution) is one argument for adopting a declarative approach. "Declarative" here means that the semantics say which conclusions are entailed by a given set of premises, without dependence on procedural or control aspects of inference algorithms. In addition to flexibility, such an approach promotes standardization, human understandability, and information reuse.

Traditionally, contracts are specified in legally enforceable natural language ("legalese"), as in a typical mortgage agreement. At the other extreme are automated languages for restricted domains (e.g., Electronic Data Interchange). In these, most of the meaning is implicit in the automated representation. We are in the sparsely occupied middle ground, aiming for considerable expressive power but also considerable automatability.

We adopt as our underlying representation of contracts sets of business rules expressed as *Courteous Logic Programs* (CLPs) [7, 8]—a language well-suited to expressing business contracts. The rules must specify the goods and services to be provided, along with applicable terms and conditions. Such terms include customer service agreements, delivery schedules, conditions for returns, usage restrictions, and other issues relevant to the good or service provided.

2.2 Configuring Auctions

ContractBot configures auctions based on rules about the contract and about the negotiation, as well as general background knowledge about auction configuration itself. This background knowledge (described in detail in Section 4) includes rules defining:

1. The process of generating suites of auctions for negotiation of multiple parameters, and for aggregating agent preferences about which auctions to generate.

2. Behavioral elements of individual auctions [17], and relationships among these elements.

3. Means of specifying these behaviors for a particular auction platform.

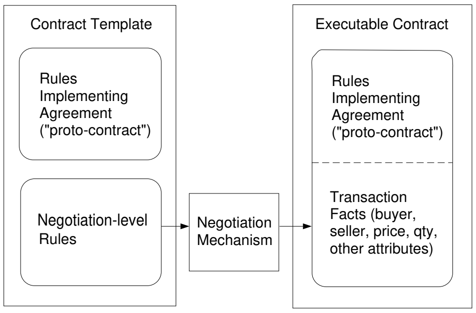

Figure 1: Overall contracting process, partial to complete contract.

To configure a set of auctions for a particular domain, we incorporate additional rules from the contract template and from potential buyers and sellers. These rules, combined with the background knowledge about auction configuration, are used to infer the actual auction parameters for a suite of auctions that will implement the chosen negotiation structure.

2.3 Overall Contracting Process

The contract template is essentially a declarative description of the space of possible negotiation outcomes, with additional rules for influencing the structure of the negotiation and how it will be configured. As shown in Figure 1, the contract template comprises rules that will implement the final agreement (the proto-contract), along with rules describing the contract issues to be negotiated (the "negotiation-level rules").

The proto-contract refers to facts and conditions regarding mechanics of the deal (e.g., payment and delivery) and ancillary agreements such as return policies or provisions for failure of one party. It is the part of the contract that carries over unchanged into the final contract, and which combines with the facts output by the negotiation mechanism to result in an executable ruleset that implements the agreement. We point out, however, that this distinction need not be sharp. In fact, an important advantage of our rule-based representation language is the ability to re-use rules and reason about them on different levels. Rules in the proto-contract that implement some aspect of the final deal may also be used in determining an appropriate negotiation mechanism. For example, time constraints on delivery may dictate auction final clearing times.

The negotiation-level rules address both questions of *what* is to be negotiated, and *how.* Rules describing ways that the contract issues may be partitioned into separable components define the space of possible *goods* to be negotiated at auction. Rules referring to policies for negotiating individual components define the auction behaviors for the corresponding goods.

The key step in this process—bridging the discovery phase (the partial contract) and the negotiation phase (set of auctions)—is the configuration of the negotiation mechanism based on inference from the contract template and rules submitted by agents.

After ContractBot configures and generates the suite of auctions, it monitors the auctions, waiting for transactions. Each transaction generates a fact specifying which aspect of the contract the transaction pertains to, who the buyer and seller were, and the price and quantity. The proto-contract contains rules that make use of such transaction facts once they are filled in. A typical rule in the proto-contract might be to say that the amount paid by agent X to agent Y is the sum of the prices in all transactions in which X bought from Y minus the sum of transactions in which Y bought from X.

Finally, after the set of auctions are configured and run and the negotiation phase is complete, we can automatically enter the execution phase by generating the final contract as a function of the proto-contract and the auction results. The actual execution of final contracts is the subject of previous work [8].

3. AUCTION-BASED NEGOTIATION

Mechanisms for determining price and other terms of an exchange are called *auctions*. Although the most familiar auction types resolve only price, it is possible to define *multidimensional* generalizations that resolve multiple issues at once. This can range from a simple approach of running independent one-dimensional auctions, to more complicated approaches that directly manage higher-order interactions among the parameters.

Auctions have proved a popular medium for Internet commerce.[2] Although typical consumer-oriented online auctions support simple negotiation services, business-to-business dynamic trade applications have begun to exhibit advanced capabilities previously found in research systems. For example, some commercial auction platforms support the configurability characteristic of the Michigan Internet AuctionBot [16],[3] and several integrate auctions with other commerce facilities, as demonstrated by an IBM auction prototype [9].

Although multidimensional mechanisms are more complicated, and not yet widely available, we expect that they will eventually provide an important medium for automated negotiation. In particular, *combinatorial* auctions allow bidders to express offers for combinations of goods, and determine an allocation that attempts to maximize overall surplus. We are aware of one prototype system that offered general combinatorial auctions over the Internet [13]. *Multiattribute* auctions, typically employed in procurement, allow specification of offers referring to multiple attributes of a single good [3].

Whether a multiattribute auction, a combinatorial auction, or an array of one- or zero-dimensional auctions[4] is appropriate depends on several factors. Although a full discussion is beyond the scope of this paper, we observe that these factors can bear on any of:

- The *coherence* of auction configurations. For example, if some attributes are inseparable (say, size and color of a good), then it makes no sense to treat them as separate goods in a combinatorial auction.

- The *expected performance* of auction configurations. For example, if parameters represent distinct and separable contract options, then they could be handled either by separate or combined auctions. If the options are considered by negotiating agents to be substitutable, then separate auctions will likely work well [10]. If they are complementary, then combining them may prove advantageous.

- The *complexity* of auction configurations, for both the mechanism infrastructure and participating agents.

In Sections 4.1 and 5 we give examples of the support that the current ContractBot provides for reasoning about the above criteria and choosing among alternative negotiation mechanisms.

4. COURTEOUS LOGIC PROGRAMS FOR GENERATING AUCTIONS

In addition to instance-specific rules from the contract template, our configuration process employs three sets of rules encoding background knowledge about the space of possible negotiation mechanisms.

4.1 Auction Configuration

The *Auction-Configuration* ruleset defines the process of collecting all possible attribute combinations for negotiable components, and generating auctions for each point in this space.[5] It generates `valueTuple` predicates for every combination of attribute values, noting which component each belongs to. It then creates an auction for each of the value tuples, and the parameters for those auctions inherit from the parameters for the parent component. In addition to determining the set of auctions for a particular component, Auction-Configuration helps determine how to partition the negotiation into components. For example, as part of the inference for determining priorities for each of several possible components we compute a "user interest score" which is the total number of users interested in the component, or zero if there is not at least one buyer and one seller:

```
<m> userInterestScore(?Component, ?N) <-
      numBuyers(?Component, ?NB) AND
      numSellers(?Component, ?NS) AND
      ?N is ?NB + ?NS.
<high> userInterestScore(?Component, 0) <-
         numSellers(?Component, 0).
<high> userInterestScore(?Component, 0) <-
         numBuyers(?Component, 0).
```

In the rule example above, the arrow ("`<-`") indicates "if" and the "?" prefix indicates a logical variable. Rule labels are specified with angle brackets (`< >`). The syntax is described in detail in the documentation for IBM CommonRules [1], the implementation of Courteous Logic Programs we employ. Also included in Auction-Configuration are rules governing the priorities of other rules. It is from these priority rules that we know, for example that the rule labels in the excerpt above have priority such that setting a score to zero when there are no buyers or sellers overrides setting the score to the sum of the number of buyers and number of sellers (the rule label "`m`" refers to medium or default priority which is less than "`high`"). Conflict resolution in CLPs is discussed in detail in previous work [7].

[2] As of this writing, eBay alone has over five million concurrently running auctions.

[3] `http://auction.eecs.umich.edu`

[4] A zero-dimensional auction is one that determines only price. A one-dimensional auction determines price and quantity.

[5] More advanced auction types (Section 3) will allow for alternative ways to handle multiple attributes.

4.2 Auction Space

Auction-Space provides basic knowledge about the parametrization of the space of possible auction mechanisms, as well as defaults for auction parameters and constraints among them. The parametrization is motivated by that employed by the AuctionBot [16] and by an extended and generalized parametrization in more recent work [17]. The default values for parameters are labeled as lowest priority rules so that parameters inferred based on specific aspects of a negotiation will take precedence. For example, the following rules specify that by default, any auction should have multiple buyers and one seller, and that ties for winning bids should be broken by first-in/first-out.

```
<lowest> auction(multipleBuyers, true).
<lowest> auction(multipleSellers, false).
<lowest> auction(tiebreaking, fifo).
```

We also specify *conditional* default parameters. For example, if we know that an auction has a single buyer then, by default, it should have multiple sellers. (Note that conditional defaults have higher priority than defaults.)

```
<verylow> auction(?ID, multipleSellers, true) <-
             auction(?ID, multipleBuyers, false).
```

Constraints are similar to conditional defaults except that they have overriding priority. For example, if there is a bidding rule that says one must beat the current quote, then this implies that the bid must *meet* the quote (i.e., a greater-than rule implies a greater-than-or-equal rule).

```
<highest> auction(?ID, meetQuote, true) <-
              auction(?ID, beatQuote, true).
```

The `negotiationType` predicate is used in a contract to aggregate auction parameters and specify how to negotiate particular components. Auction-Space maps such directives to more specific auction features. Note that negotiationTypes can entail other negotiationTypes but that the inference must propagate conclusions to `auction` predicates eventually. For example, `negotiationType(continuous)` implies, among other things, `negotiationType(continuousClears)` which in turn implies `auction(quoteMode,bid)`.

One particularly useful feature of Auction-Space is that it encodes several well-known auction types. For example, specifying a negotiation type of "CDA" is all that is necessary to infer all the characteristics of a Continuous Double Auction [5]—chronological matching, continuous quotes (bid-ask) and clears, double-sided, and discrete goods.

```
<m> auction(?ID, matchingFunction, earliestTime)
          AND negotiationType(?ID, continuous)
          AND negotiationType(?ID, double)
          AND auction(?ID, divisible, false)
          AND auction(?ID, quoteMode, bidAndAsk)
     <- negotiationType(?ID, cda).
```

The conflict resolution that CLP provides is also useful here. For example, it allows specifying that an "Amazon-style" auction is just like "eBay-style" *except* that Amazon auctions do not close until ten minutes of inactivity have passed.[6]

[6]Incidentally, this difference turns out to have a marked effect on agent strategies. [12]

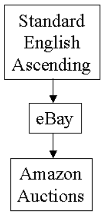

Figure 2: An example of an inheritance hierarchy encoded in the Auction-Space ruleset.

```
<ebay> auction(?ID, matchingFunction, mthPrice) AND
         auction(?ID, multipleBuyers, true) AND
         auction(?ID, multipleSellers, false) AND
         negotiationType(?ID, revealAll) AND
         ...
         auction(?ID, finalClearMode, fixed)
    <- negotiationType(?ID, ebay).

negotiationType(?ID, ebay) <-
  negotiationType(?ID, amazon).

<amazon> auction(?ID, finalClearMode, inactivity)
AND auction(?ID, finalClearInactivityInterval, 600)
   <- negotiationType(?ID, amazon).

overrides(amazon, ebay).  /* Amazon rule is an
                             exception */
```

In our implementation we include additional rules for the case of the standard English auction as a more general type, and include eBay as a special case. The result is a three-level hierarchy, shown in Figure 2.

4.3 Auction Server Mapping

Based on parameters concluded from Auction-Space, we need to drive a specific auction server. *AuctionBot-Mapping* is a particular set of rules for deriving AuctionBot parameters from the generalized and extended parametrization provided by Auction-Space. Separating the server-specific ruleset allows for great flexibility in inferring negotiation mechanisms, allows us to cope elegantly with shortcomings in the AuctionBot parametrization, and facilitates connection of ContractBot to alternative auction platforms.

4.4 Domain-specific Rules: Widget Example

Consider a simple contract involving only one component (a widget) with only one attribute (quality) with two possible values (regular and deluxe).

```
component(widget).
attribute(widget, quality).
possValue(quality, regular).
possValue(quality, deluxe).
```

The possible values are not tied to the widget component because in general they might apply to more than one component in the contract. The following general rule creates `possValue/3` rules for each component based on the general `possValue/2` rules and the components that have been declared:

```
possValue(?Component, quality, ?Q) <-
  component(?Component) AND possValue(quality, ?Q).
```

Negotiating for a widget means determining multiple attributes (price, quantity, and quality) but the current AuctionBot supports only single-dimensional auctions (negotiating price and quantity). One (rather brute-force) method for dealing with this is to simply create an array of single-dimensional auctions, one for each point in attribute-value space.[7] The following rule enumerates all the points in attribute-value space for all components.[8] In this simple example, only two points will be enumerated, one for each possible value of the single attribute of the single component, a widget:

```
valueTuple(?Component, [?Quality]) <-
   possValue(?Component, quality, ?Quality).
```

Note that "[?Quality]" represents a list with a single element (a variable representing a value for the "quality" attribute). The above rule creates a `valueTuple/2` fact for every possible way to assign values to the attributes of a component.

Next, we provide general information about the negotiation of widgets. These facts are used by Auction-Space to generate the full set of auction parameters for widget auctions. (In this case, most of the parameters will be default values.)

```
negotiationType(widget, continuous).
negotiationType(widget, double).
negotiationType(widget, revealAll).
```

At this point, we have inferred the auction parameters for widgets and we have enumerated the valueTuples for the auctions we need to create. We now combine those steps to explicitly create the auctions and have each of them inherit its parameters from those derived for widgets in general.

For every valueTuple, we infer a `makeAuction/1` fact which takes a list (thought of as an ID) and tells our prototype to create an actual auction. We also infer a `parent/2` fact for every valueTuple.[9] This tells us the component that each auction belongs to.

```
makeAuction([?Component, ?Values]) AND
       parent([?Component,?Values], ?Component)
<- valueTuple(?Component, ?Values).
```

Finally, we specify the auction parameters for each created auction—simply the parameters that we derived in general for the component that the auction belongs to (its parent).

```
auction(?ID, ?Attr, ?Val) <-
    parent(?ID, ?Component) AND
    auction(?Component, ?Attr, ?Val).
```

[7]Note that this places quite a burden on agents—for example, if they would value either a regular or a deluxe widget and place bids for both, they assume the risk of being stuck with redundant items.

[8]The rules for enumerating points in attribute-value space are generalized and included in the background knowledge encoded in Auction-Configuration. This specific rule is presented for illustration and would not actually be necessary in specifying a contract template with ContractBot.

[9]Inferring the set of auctions from the valueTuples, as well as inheritance of parameters from parent components, is done automatically in the Auction-Configuration ruleset. So the remaining rules also would be unnecessary in a contract template with our implementation.

When the inference concludes, there will be two auctions created: "widget: quality= regular" and "widget: quality= deluxe". For each auction, 27 distinct auction parameters will be inferred via Auction-Space and AuctionBot-Mapping. Note that since these auctions derived from the same parent component (widget), their parameter values are identical.

5. THE TAC DOMAIN

In July 2000 at the International Conference on Multiagent Systems,[10] the University of Michigan hosted a *trading agent competition* (TAC), in which software agents developed by participating teams competed in a challenging market game [15]. In TAC, agents aim to assemble travel packages for designated clients, buying and selling travel resources through various types of auctions implemented by the Michigan Internet AuctionBot. TAC features three basic travel goods: flights (defined by day and destination—inbound or outbound), hotels (defined by day and quality), and entertainment tickets (defined by day and type of event). Each type of good is mediated by a different kind of auction. Flights sell at randomly fluctuating prices, dictated by a fixed-price seller. Hotels are sold in a variant of an ascending English auction. Agents buy and sell entertainment tickets in a continuous double auction, much like trading securities in a stock exchange.

The TAC example Contract Template included with ContractBot is a ruleset that generates the partitioning (among a space of possible partitionings) of a travel package contract into the goods described above. It also generates, from a high-level description in the contract template, the auction configurations used in the competition. In Section 6 we demonstrate a key result: how alternative structures for the negotiation can be derived, based on rules plausibly contributed by parties interested in the negotiation.

5.1 Contract Template: Proto-Contract about Payments and Utilities

The first thing the TAC contract template specifies is a proto-contract. As described in Section 2.3, the proto-contract is the subset of the contract template that, when combined with the rules coming out of the negotiation mechanism, form the final, executable contract. A typical rule for a proto-contract (see Section 2.3) that we have included in the TAC example is inferring the total amount that a given agent owes another agent after the negotiation, by aggregating transaction facts from completed auctions:

```
transact(?Agent1, ?Agent2, ?Component, ?AVList,
         ?Pay12, ?Qty).
```

More specific to TAC, we include rules in the proto-contract to infer the utility that a travel agent receives from its transactions, according to the definition of the TAC game.[11] As in the payment example, this is a straightforward computation as a function of the transaction facts.

Following is a part of the utility calculation specifying that a client's utility is a function of whether it was able

[10]and again in 2001 at the ACM Conference on Electronic Commerce

[11]A utility calculation would probably not make sense in a proto-contract in the real world, but in the TAC game, the utility is used externally to evaluate agents' performance.

to procure a trip, its deviation from its ideal travel dates, and its bonuses for staying in the nice hotel and seeing the entertainment it wanted:

```
<high> clientUtility(?Client, 0) <-
         feasibleTrip(?Client, false).
<m> clientUtility(?Client, ?U) <-
       feasibleTrip(?Client, true) AND
       travelPenalty(?Client, ?TP) AND
       hotelBonus(?Client, ?HB) AND
       funBonus(?Client, ?FB) AND
       ?U is 1000 - 100 * ?TP + ?HB + ?FB.
```

Note that although the complete ruleset for utility calculation is not given, all of the above predicates can be inferred from transaction facts generated by ContractBot as it monitors the auction results. The TAC example contract template then includes rules to infer a travel agent's utility in the competition by summing the utilities of its clients and subtracting its expenses.

5.2 Contract Template: Possible Components and Attributes

The first thing the TAC contract template specifies after the proto-contract is the possible values for the attributes of the goods. For example, the following facts set the possible types of entertainment events:

```
possValue(entertainment, type, baseball).
possValue(entertainment, type, symphony).
possValue(entertainment, type, theatre).
```

After specifying the domains for the attributes, there are several sections of rules corresponding to possible components of the TAC domain. These give the attributes of each of the components, as well as specify negotiation-level rules. For example, the following rules specify that flights have two attributes—type (inbound or outbound) and day.

```
attribute(flight, type).
attribute(flight, day).
  possValue(flight, day, ?Val) <-
     possValue(day, ?Val).
```

Note that the possible values for flight types were enumerated in separate rules. The possible values for flight days are inferred from the globally defined `day` values, declared with `possValue/2` predicates (similar to quality values for widgets in Section 4.4).

Another possible component is the bundle of two flights into a round-trip.

```
attribute(roundflight, dayin).
attribute(roundflight, dayout).
<m> auction(roundflight, ?Param, ?Val) <-
       auction(flight, ?Param, ?Val).
```

Two attributes for `roundflight` components are specified, and they inherit their domains from the globally defined domain for possible days. The individual auction parameters for round-trip flights are inherited from those inferred for one-way flights.

Figure 3 illustrates some of the possible components specified in the TAC contract template and the next section shows how ContractBot chooses between these alternative configurations.

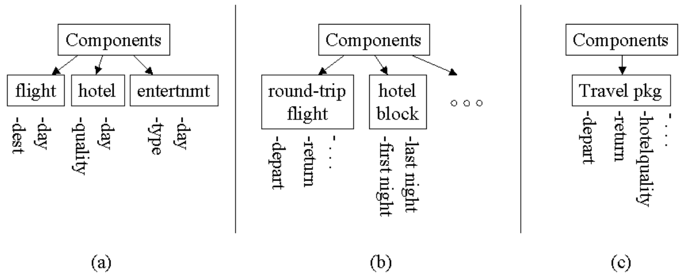

Figure 3: Some alternative ways to structure the TAC negotiation: (a) the actual TAC configuration, (b) bundling round-trip flights and blocks of hotel rooms, and (c) bundling everything into a comprehensive travel package.

6. ALTERNATIVE NEGOTIATION STRUCTURES: TAC EXAMPLE

One of the most interesting features of ContractBot is its ability to reason about alternative negotiation structures by stating relationships between the possible components and incorporating rules from participating agents.

The example TAC Contract Template includes buyer, seller, and auctioneer rules which are consistent with the scenario in the actual TAC game. There are travel agents interested in buying any of the atomic components, and who would also be interested in buying various bundles, such as round trip flights and blocks of hotel rooms. The travel agents are also interested in selling entertainment tickets as well as buying them. The only other sellers are the airline, who is only willing to sell one-way flights, and the hotels, who will sell rooms either individually or in blocks. The auctioneer prefers fewer auctions and more users. It makes this explicit by specifying a `perAuctionCost` and a `perUserCredit`. ContractBot will choose a consistent set of components that minimizes

$$\#users * \texttt{perUserCredit} - \#auctions * \texttt{perAuctionCost}.$$

Although the hotel seller is willing to sell either blocks of hotels or individual nights, having a high `perAuctionCost` leads to the structure shown in Figure 3 (a). This is because the `hotelblock` component has more attributes (`firstnight` and `lastnight`) than `hotel` (which has only one "night" attribute), resulting in more auctions and therefore a higher cost. Section 4.1 discusses some of the component scoring rules that drive this inference. As described in the widget example (Section 4.4), the Auction-Configuration ruleset will infer multiple auctions for each good. In the TAC example, assuming four days, there will be $4 * 2 = 8$ flight auctions—one for every combination of day and type (inbound or outbound)—and similarly for hotels which have two types (good or bad). Entertainment tickets have three types (baseball, symphony, or theatre) and so $4 * 3 = 12$ auctions are created.

Note that the buyer/seller rules include several travelers who are interested in buying complete travel packages (see Figure 3 (c)). This structure is not inferred, however, because there are no agents willing to sell travel packages (as well as the prohibitive `perAuctionCost`). By adding rules such as

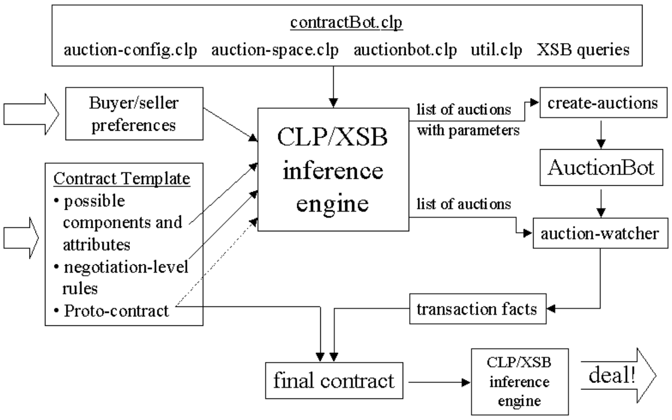

Figure 4: How ContractBot uses its auction knowledge to turn a partial contract into a complete, executable contract. The large arrows represent inputs and outputs of the system. The final stage of executing a final contract is the subject of previous work [8].

```
seller(hypotheticalSeller,travelpackage).
perAuctionCost(0).
```

ContractBot will instead infer a single component with six attributes (arrive, depart, hoteltype, ent1day, ent2day, ent3day). The sizes of the domains of these attributes are 4, 4, 2, 5, 5, and 5, respectively, and would require 392 auctions.[12]

Without a `perAuctionCost`, the following rules are the minimal set necessary to choose a bundling of flights into round trips, hotel rooms into contiguous blocks, and entertainment into packages of three events.

```
buyer(traveler1, roundflight).
buyer(traveler2, hotelblock).
buyer(traveler1, entpackage).
seller(airline1, roundflight).
seller(hotel1, hotelblock).
seller(agent3, entpackage).
```

These rules will (trivially) infer the structure shown in Figure 3 (b). Various other partitionings of the contract into bundles can be inferred similarly by adding or removing potential buyers and sellers for the various possible components and adjusting the cost of additional auctions and credit for additional buyers and sellers.

7. PROTOTYPE IMPLEMENTATION

Figure 4 depicts the overall process of turning a contract template along with rules from agents into a final contract, and then an executed deal. At the heart of this process are the three sets of background knowledge discussed in Section 4—Auction-Configuration, Auction-Space, and AuctionBot-Mapping. ContractBot.clp wraps these rulebases together along with miscellaneous utilities (util.clp) and Prolog (XSB) [2] queries that drive the inference.

The inference engine itself is actually written as a series of Perl scripts that flow a set of input rules and the background knowledge through CLP and Prolog. The main ContractBot executable accepts arbitrary CLP rules (generally the contract template and buyer/seller rules) on standard input and combines these rules with the background knowledge specified in contractBot.clp. This conglomeration of CLP input is fed into the *Courteous Compiler*, a component of IBM CommonRules which compiles CLP into ordinary Prolog. This Prolog code is combined with the queries specified in contractBot.clp and fed into the XSB Prolog engine.

It is these queries that generate the output that the following modules need in order to interact with the AuctionBot. For example, to generate the list of auctions to be created, contractBot.clp makes a query which writes a list to standard output containing all the auction IDs for which there is a `makeAuction` fact entailed by the knowledge base. These facts are generated by Auction-Configuration for every point in attribute space for every component inferred. Components, in turn, are inferred from the contract template and from rules from agents.

The output of the Prolog queries amounts to a list of auctions and parameter values for each auction. The list of auctions and parameter settings are fed to the *create-auctions* module which connects to the AuctionBot via the Mathematica implementation of the *AuctionBot Agent Communication Protocol*[13] and creates the auctions. The list of auctions is also sent to the *auction-watcher* module which monitors the specified auctions and composes the corresponding transaction facts (see Sections 2.3 and 5) whenever a transaction occurs on AuctionBot in an auction relevant to the contract. Finally, the transaction facts are concatenated with the proto-contract from the original contract template to form an executable contract which can itself be fed through an inference engine to execute the terms of the deal [8].

8. DISCUSSION AND FUTURE WORK

We have given a new approach to automatically bridging the three stages of contracting: discovery, negotiation, and execution. In particular, this encompasses three new contributions: (1) we use a declarative contracting language to represent information pertinent to all three aspects of contracting, (2) our approach relates all 3 aspects to each other and enables the same information to be used in multiple stages, and (3) we have shown the flexibility of this approach by implementing a prototype (ContractBot) and generating alternative negotiation structures for a travel domain (TAC).

Our approach addresses several research questions regarding practical automation of the contracting process. First, how can we represent information to allow automatic inference of negotiation structures? Second, how can we automatically specify negotiations in a way that will closely drive a realistic automated platform? Third, how can we use auction results to form a final contract?

In our prototype, we use Courteous Logic Programs to represent (1) partial contracts, (2) additional rules about the negotiation process from buyers and sellers, (3) background knowledge about how to structure negotiation mechanisms and configure individual auctions, and (4) final, executable

[12]Less than 4000 since infeasible packages—e.g., departure before arrival—are not inferred.

[13]Mathematica was chosen for its clean implementation of the protocol and its convenient LISP-like handling of the auction and parameter lists.

contracts. Using rules in this language, from the contract template and from potential buyers and sellers, we first infer the basic structure of the negotiation mechanism by applying background knowledge encoded in Auction-Configuration. We then use the Auction-Space knowledge base to infer a general set of parameters for each of the auctions supporting the negotiation. The AuctionBot-Mapping ruleset translates this into an operational specification for the Michigan Internet AuctionBot. Finally, we combine rules generated by auction transactions with rules in the proto-contract of the contract template, to form a final contract which itself is executable using rule-based techniques.

Our prototype can generate sets of auctions corresponding to a multicomponent, multiattribute negotiation, and supports reasoning about alternative ways to decompose a contract into multiattribute components. In future work, we would like to extend AuctionBot as well as our ontology in ContractBot to support richer negotiation mechanisms. We currently handle multiple attributes of a component by creating an array of auctions, one for every combination of attribute values. This is not tractable for larger numbers of attributes and needs to be augmented with multiattribute and/or combinatorial auctions. As we add additional negotiation mechanisms to AuctionBot, we will be able to add more sophisticated background knowledge about how to optimally structure a negotiation according to the criteria discussed in Section 3. To extend our ability to handle the execution phase of contracting, we will generalize our knowledge representation to express *Situated* Courteous LPs. Situated logic programs [6] use beliefs to drive procedural APIs.

One piece of future work outside of ContractBot itself will involve writing agents that participate in the infrastructure we've developed. This is an extremely rich area for analyzing complex agent strategies since an agent using ContractBot must not only know how to bid intelligently in a vast space of negotiation mechanisms, but also intelligently contribute rules to influence which negotiation mechanism is chosen. Further issues for future work include meshing more closely with other aspects of contracts, e.g., transactions, payments, negotiation and communication protocols [4] [11], and supplier selection [14].

Acknowledgments

This work was supported by IBM's University Partnership Program while the third author was at IBM. Hoi Chan of IBM played a key role in development of CommonRules. We would also like to thank Bill Walsh, Terence Kelly, David Parkes, Ed Durfee, Bill Birmingham, and Rachel Rose for helpful discussions and comments in earlier stages of this work.

9. REFERENCES

[1] IBM CommonRules. `http://www.research.ibm.com/rules/commonrules-overview.html`.

[2] The XSB Programming System. `http://xsb.sourceforge.net/`.

[3] F. Branco. The design of multidimensional auctions. *RAND Journal of Economics*, 28:63–81, 1997.

[4] A. Dan, D. Dias, T. Nguyen, M. Sachs, H. Shaikh, R. King, and S. Duri. The coyote project: Framework for multi-party e-commerce. In *Seventh Delos Workshop on Electronic Commerce, Lecture Notes in Computer Science, Vol. 1513*. Springer-Verlag, 1998.

[5] D. Friedman and J. Rust, editors. *The Double Auction Market*. Addison-Wesley, 1993.

[6] B. N. Grosof. Building Commercial Agents: An IBM Research Perspective. In *Proceedings of the Second International Conference and Exhibition on Practical Applications of Intelligent Agents and Multi-Agent Technology (PAAM97)*, April 1997.

[7] B. N. Grosof. Prioritized conflict handling for logic programs. In J. Maluszynski, editor, *Logic Programming: Proceedings of the International Symposium (ILPS-97)*, pages 197–211, Cambridge, MA, USA, 1997. MIT Press.

[8] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: Courteous logic programs in XML. In *ACM Conference on Electronic Commerce*, pages 68–77, Denver, 1999.

[9] M. Kumar and S. I. Feldman. Internet auctions. In *Third USENIX Workshop on Electronic Commerce*, pages 49–60, Boston, 1998.

[10] P. Milgrom. Putting auction theory to work: The simultaneous ascending auction. *Journal of Political Economy*, 108, 2000.

[11] N. H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *18th International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[12] A. E. Roth and A. Ockenfels. Last minute bidding and the rules for ending second-price auctions: Theory and evidence from a natural experiment on the internet. Technical report, May 2000. `http://www.economics.harvard.edu/~aroth/alroth.html`.

[13] T. Sandholm. eMediator: A next generation electronic commerce server. In *Fourth International Conference on Autonomous Agents*, Barcelona, 2000.

[14] P. Szekely, B. Neches, D. P. Benjamin, J. Chen, and C. M. Rogers. Controlling supplier selection in an automated purchasing system. In *AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce (AIEC-99)*, Menlo Park, CA, USA, 1999.

[15] M. P. Wellman, P. R. Wurman, K. O'Malley, R. Bangera, S.-d. Lin, D. Reeves, and W. E. Walsh. Designing the market game for a trading agent competition. *IEEE Internet Computing*, 5(2), 2001.

[16] P. R. Wurman, M. P. Wellman, and W. E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents*, pages 301–308, Minneapolis, 1998.

[17] P. R. Wurman, M. P. Wellman, and W. E. Walsh. A parametrization of the auction design space. *Games and Economic Behavior*, 35, 2001.