Automated Negotiation from Declarative Contract Descriptions

Daniel M. Reeves and **Benjamin N. Grosof** and **Michael P. Wellman** and **Hoi Y. Chan**

University of Michigan Artificial Intelligence Laboratory
1101 Beal Avenue, Ann Arbor, MI 48109-2110 USA
{dreeves, wellman}@umich.edu
http://ai.eecs.umich.edu/people/{dreeves, wellman}/

IBM T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532 USA
{grosof, hychan}@us.ibm.com
http://www.research.ibm.com/people/g/grosof/

Abstract

We give a new, implemented approach for automating the negotiation of business contracts. We use our previous work on developing a declarative language for expressing and reasoning about contracts and negotiations. Here we newly extend it to include a knowledge base of rules about negotiation structures and auctions. This work addresses three important research questions. First, how can we represent information to allow automatic inference of negotiation structures? Second, how can we automate negotiations in a way that will closely drive a realistic automated platform (the Michigan Internet AuctionBot)? Third, how can we use auction results to form a final contract? We use our work on Courteous Logic Programs, a form of logic-based knowledge representation, as a way to express fully-specified, executable contracts and extend this to also express partially-specified contracts that are in the midst of being negotiated. In our current prototype, we have developed concepts and vocabulary to reason about several aspects of the negotiation process: (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 (the Michigan Internet AuctionBot), and (4) special-case rules for specific domains, including rules from potential buyers and sellers about capabilities, constraints, and preferences. By performing inferencing on the rule sets and interfacing to our auction server, our prototype is able to automatically configure a set of auctions, the results of which will "fill in the blanks" of a partial contract. We use an upcoming Trading Agent Competition as an example domain and are able to automatically generate all the auctions used in the competition (and other possible configurations) starting from a formal description of the competition domain. The result of this project is an extended approach which allows both the automation of the negotiation process, includes conducting of auctions, and produces contracts with are themselves executable using rule-based techniques.

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 primarily with negotiation, and specifically with the process by which an automated negotiation mechanism can be configured to support a particular contracting episode. We present 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 make a sharp distinction between the definition of the negotiation mechanism, and the actual negotiation strategies to be employed by participating agents. Our concern here is with the former, though of course in designing a mechanism one must consider the private evaluation and decision making performed by each of the negotiating parties.

The contribution in this work is a system which bridges the gaps between the discovery and negotiation phases above, and between negotiation and execution. We call the current prototype of this system *ContractBot*. 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 mechanism. Then, by monitoring the individual auction results, it generates the final, executable contract.

Overview of Problem and Approach

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 to a useful degree in the current state of the art.

Most contracting contexts lie somewhere in between, where an identifiable set of issues are to be determined through negotiation. 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*—a medium for expressing the contract terms resulting from a negotiation.

In this project, we focus on the automatic configuration of negotiations based on a contract and show how the negotiation results can be used to construct a final, "filled-in" contract. Sections "Auction-Based Negotiation" and "Courteous Logic Programs as KR" provide background on auction-based negotiation and the rule language we use to express contracts. Section "Contracting Framework" 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 "Courteous Logic Programs for Configuring Auctions" we discuss in detail how the language is used to infer parameters for configuring the negotiation—that is, parameters for a set of auctions—focusing on the upcoming Trading Agent Competition (Wellman & Wurman 1999) as an example domain (Section "Domain Specific Rules: Trading Agent Competition"). Finally, in Section "Prototype Implementation," we discuss the details of our ContractBot prototype. It processes the contract description (rules for describing possible components and their attributes) along with meta-level rules about the negotiation and about individual auctions. It combines all this with rules from buyers and sellers about their constraints and preferences over the possible negotiation structures. Based on inferencing by a rule engine, it generates the appropriate auctions and determines the auction parameters. When transactions happen in the auctions, it generates the corresponding rules and produces a final contract.

Contracting Language

In developing a shared contracting language, we are concerned with the three stages of contracting: discovery, negotiation, and execution. This breadth of scope is one argument for adopting a declarative approach, with a relatively expressive knowledge representation (KR). "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 and human understandability.

Traditionally, of course, contracts are specified in legally enforceable natural language ("legalese"), as in a typical mortgage agreement. This has great expressive power—but often, correspondingly great ambiguity, and is thus very difficult to automate.[1] At the other extreme are automated languages for restricted domains; in these, most of the meaning is implicit in the automated representation. This is the current state of Electronic Data Interchange (EDI). We are in the sparsely occupied middle ground, aiming for considerable expressive power but also considerable automatability.

Our point of departure for our KR is pure logic programs (in the knowledge-representation-theory sense, not Prolog). (Baral & Gelfond (Baral & Gelfond 1994) provide a helpful review.) Logic programs are not only declarative and relatively powerful expressively, but also practical, relatively computationally efficient, and widely deployed.

Our KR builds on prior work (Reeves *et al.* 1999) representing business rules in Courteous Logic Programs (CLPs) (Grosof 1997; Grosof, Labrou, & Chan 1999), described in more detail in Section "Courteous Logic Programs as KR." To express executable contracts, these 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.

As part of our approach, we extend this KR with features specific to negotiation. Foremost among these is the ability to specify *partial* agreements, with associated negotiable parameters. A partial agreement can be viewed as a contract template. Some of its parameters may be bound to particular values while others may be left open. In our current prototype, we focus on rules that express aspects of how these parameters are actually negotiated—i.e., rules for configuring the negotiation mechanism (set of auctions)—but also generate rules for the final contract based on the negotiation results.

Negotiable Parameters

Once we have this contracting language, our next step will be to use it to establish the automated negotiation process. As noted above, a key element of this is to identify the negotiable parameters. The contract template effectively defines these parameters by specifying what the contract will be for any instantiation of parameter values.

The problem then, is to enable the contract language to allow descriptions of contract templates. In addition, we require auxiliary specification of possible values for parameters, and dependencies and constraints among them. Given this specification of what can be negotiated, we require a policy to determine what is actually to be included in the given negotiation episode (rather than assigned a default value, or left open for subsequent resolution).

[1] Even if a natural language contract is completely unambiguous, it would require a vast amount of background and domain knowledge to automate.

This answers the question of *what* is to be negotiated; the remaining question is *how*. In general, there are many ways to structure a negotiation process to resolve multiple parameters. We focus on processes mediated by auctions. As we describe below, the problem then becomes one of *configuring* appropriate auctions to manage the negotiation.

Auction Configuration

To support the configuration of auctions based on rules about the contract and about the negotiation, we have created three general-purpose sets of rules, Auction-Configuration, Auction-Space, and AuctionBot-Mapping (see subsections under "Courteous Logic Programs for Configuring Auctions" and corresponding appendices) which provide background knowledge about the configuration of auctions. Auction-Configuration encodes control-level knowledge about the process of generating a suite of auctions to support negotiation of multiple parameters. It also encodes knowledge for aggregating agent preferences in determining the set of auctions to create. Auction-Space is modeled on our current parameterization of auction design space (Wurman, Walsh, & Wellman 1998) (discussed in Section "Auction-Space"). It lays out the set of auction parameters, specifying their domains, and default values, as well as constraints and other rules about how they influence each other. Additionally, it clusters sets of parameters based on well-known auction types such as Continuous Double Auctions (CDA)[2] or English.[3] The AuctionBot-Mapping ruleset maps the auction-space parameterization to the AuctionBot. The mapping is not at all straightforward since we have significantly changed our view of the parameterization of auction-space but our implementation on AuctionBot has not kept up due to backward compatibility constraints and has become rather convoluted.

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 described above, are used to infer the actual auction parameters for a suite of auctions that will implement the chosen negotiation structure. We discuss an example in Section "Domain-specific Rules: Trading Agent Competition" which implements the creation of the auctions for the ICMAS Trading Agent Competition (Wellman & Wurman 1999) as well as choosing between multiple possible configurations for the competition.

Composing Final Contracts

Once ContractBot configures and generates the suite of auctions, it monitors the auctions, waiting for transactions. Each transaction generates a fact specifying what component was transacted, what the values were for each of its attributes, who the buyer and seller were, and the price and quantity. The partial contract contains rules that make use of such transaction facts once they are filled in. The portion of the contract template that combines with the transaction facts we call the proto-contract. 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. More about the proto-contract and forming executable final contracts is discussed in Section "Contracting Framework." Section "Domain-specific Rules: Trading Agent Competition" discusses an example of generating final contracts in the Trading Agent Competition domain.

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 and variants that resolve multiple issues at once. This can range from the simple approach of running independent one-dimensional auctions for all of the parameters of interest, to more complicated approaches that directly manage higher-order interactions among the parameters.

Auctions are rapidly proliferating on the Internet.[4] Although typical online auctions support simple negotiation services, researchers have begun to deploy mechanisms with advanced features. For example, our own Michigan Internet AuctionBot supports a high degree of configurability (Wurman, Wellman, & Walsh 1998) (http://auction.eecs.umich.edu/), and IBM's auction system supports one-sided sales auctions integrated with other commerce facilities (Kumar & Feldman 1998).

Although multidimensional mechanisms are more complicated, and not yet widely available, we expect that they will eventually provide an important medium for automated negotiation. For example, *combinatorial* auctions allow bidders to express offers for combinations of goods, and determines an allocation that attempts to maximize overall revenue. We are aware of one prototype system currently supporting combinatorial auctions over the Internet (Sandholm to appear). *Multiattribute* auctions, typically employed in procurement, allow specification of offers referring to multiple attributes of a single good (Branco 1997).

Whether a multiattribute auction, a combinatorial auction, or an array of one- or zero-dimensional auctions[5] 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, arrival and departure times), 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

[2]Stock markets are examples of CDAs. See Friedman and Rusts's book on double auctions (Friedman & Rust 1993).

[3]Consumer auctions on the Internet (like eBay) are mostly variants of English auctions.

[4]As of this writing, eBay alone has over 4 million currently running auctions.

[5]A zero-dimensional auction is one which determines only price. A one-dimensional auction determines price and quantity.

contract options, then they could be handled either by separate or combined auctions. Whether they should be combined depends on how *complementary* the negotiating agents perceive them to be.

- The *complexity* of auction configurations, for both the mechanism infrastructure and participating agents. Dimensionality plays a large role in complexity tradeoffs.

In Sections "Auction-Configuration" and "Domain-specific Rules: Trading Agent Competition" we discuss and give examples of some of the limited support that the current ContractBot provides for reasoning about some of the above criteria.

Courteous Logic Programs as KR

The KR we are using to represent contracts is **Courteous** Logic Programs. Courteous LPs expressively generalize *ordinary* LPs by adding the capability to conveniently express prioritized conflict handling, i.e., where some rules are subject to override by higher-priority conflicting rules. For example, some rules may be overridden by other rules that are special-case exceptions, more-recent updates, or from higher-authority sources. Courteous LPs facilitate specifying sets of rules by merging and updating and accumulation, in a style closer (than ordinary LPs) to natural language descriptions. Priorities are represented via a fact comparing rule labels: $overrides(rule1, rule2)$ means that $rule1$ has higher priority than $rule2$. If $rule1$ and $rule2$ conflict, then $rule1$ will win the conflict. See Section "Courteous Logic Programs for Configuring Auctions" for examples of labeled rules and prioritizations.

Courteous LPs have several virtues semantically and computationally. A Courteous LP is guaranteed to have a consistent, as well as unique, set of conclusions. Priorities and merging behave intuitively. Execution (inferencing) of courteous LPs is fast: only relatively low computational overhead is imposed by the conflict handling.

Our work on representing contracts via Courteous LPs builds on prior work at IBM representing business rules. The implementation we are using is a Java library called CommonRules available from IBM (com).

Contracting Framework

We now describe the process of automatically turning a partially specified contract into a fully specified, executable one. The partial contract, or contract template, is 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. Our previous work (Reeves *et al.* 1999) presents a language based on Courteous Logic Programs for representing contract templates and shows that this language is sufficiently general to support the negotiation of any aspect of an executable contract. As shown in Figure 1, the contract template consists partially of rules that will implement the final agreement (called the "proto-contract"), as well as rules that describe the components of the contract left to be determined. The proto-contract refers to facts and conditions regarding, for example, mechanics of the deal (payment and delivery) or ancillary agreements such as return policies (see Section "Courteous Logic Programs as KR"). 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 as part of the inferencing to establish an appropriate negotiation mechanism. For example, time constraints on delivery may dictate final clearing times for auctions. Figure 2, shows in more detail how a partial contract, including its proto-contract, is mapped to a final contract using general rules about auction configuration (see Section "Courteous Logic Programs for Configuring Auctions").

The rules in the partial contract that describe the negotiable aspects have two purposes. The first is to describe the hierarchy of components and attributes of a contract. In our previous work we present a richer component description ontology which allows reasoning about orthogonality and separability of pieces of a contract, as well as arbitrarily nested hierarchies of components and attributes. This is aimed at more sophisticated multidimensional negotiation mechanisms that we plan to support in AuctionBot in the future. For the purposes of our current prototype, we only support partitioning the contract into separable components, each of which may have a set of attributes. However, we also allow the contract template to express reasoning about various ways to split up a component into separable components. The final partitioning of the contract into a set of components may be influenced by potential buyers and sellers who submit rules that specify their constraints and preferences among the alternative negotiation structures.

The second purpose for rules about the components of a contract is to specify high-level knowledge about how the individual components should be negotiated. The rule engine can then infer the necessary parameters for configuring the negotiation mechanism. This is currently a set of single-dimensional auctions for each included component, but could also by a multidimensional mechanism that would determine attributes of a component simultaneously, or allow bidders to directly express preferences of subsets of components.

The key step in this process—interfacing the discovery phase (the partial contract) with the negotiation phase (set of auctions)—is then the configuration of the negotiation mechanism based on the inferencing done from the contract template and rules submitted by buyers and sellers. 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 the final contract as a function of the proto-contract and the auction results. (and the focus of the prototype discussed in Section "Prototype Implementation") is then the configuration of the negotiation mechanism based on the inferencing done from the contract template. After the set of auctions are configured and run and the negotiation phase is complete, high-priority facts are added to

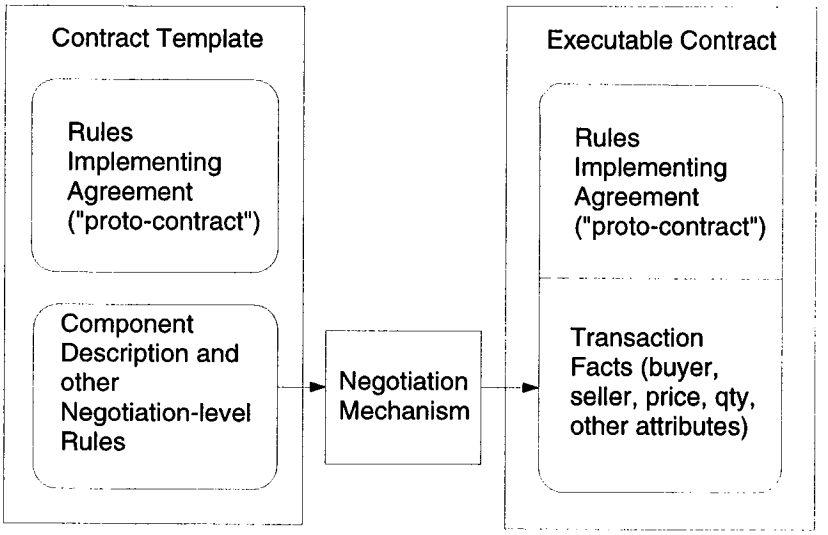

Figure 1: Overall contracting process, partial to complete contract.

the contract, yielding a set of rules that fully implements the negotiated agreement.

In the next two sections we describe our implementation of the overall contracting process. Namely, (1) the configuration of the negotiation mechanism based on rules describing possible partitionings of the contract into components, rules from buyers and sellers influencing the choice of negotiation structure, and rules about how individual components should be negotiated; (2) the generation of an executable contract by combining rules generated for each auction transaction with the proto-contract from the contract template.

Courteous Logic Programs for Configuring Auctions

In this section we discuss the logic programming aspects of inferring the parameters for the negotiation mechanism. The primary set of rules for this inferencing comes from the contract template (see Section "Domain-specific Rules: Trading Agent Competition" for a detailed example of a contract template) but we also have three sets of rules that serve as background knowledge about the space of possible negotiation mechanisms. The first is Auction-Configuration (Section "Auction-Configuration") which is used for determining which set of auctions to create based on agent preferences and constraints. It implements the method of setting up a multidimensional negotiation by creating an array of single-dimensional auctions—one for each combination of attribute values. Additionally, it contains miscellaneous, low-level rules used in the configuration of auctions. The next ruleset is Auction-Space (Section "Auction-Space") which provides basic knowledge about our parameterization of the space of possible auction mechanisms, as well as defaults for auction parameters and constraints among them. It also contains various heuristics for setting auction parameters and aggregation of sets of parameters into "auction types". The last ruleset (Section "AuctionBot-Mapping") maps this general auction knowledge to a specific auction server—the Michigan Internet AuctionBot.

Together, the rules from the contract template (and rules from buyers and sellers) and the auction mechanism background knowledge enable the inferencing engine to reach a set of conclusions that is sufficient to configure the negotiation.

Auction-Configuration

The Auction-Configuration ruleset (see Appendix "Auction Configuration Ruleset" implements the technique of simulating a multiattribute auction by holding an array of single-dimensional auctions—one for every point in attribute-space. 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, it infers a score for each of several possible components[6] by counting the total number of buyers and sellers interested in them, unless there are no buyers or no sellers, in which case the score is zero:

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

[6]See Section "Domain-specific Rules: Trading Agent Competition" for examples of alternative components for a contract.

The numBuyers and numSellers rules are determined based on buyers and sellers who submit rules specifying their interest in buying or selling certain components of the contract.

Also included in Auction-Configuration are rules governing the priorities of other rules, as in the example above. There are currently several levels of rule priorities: lowest, verylow, low, medium (abbreviated as "m"), high, veryhigh, and highest. The "lowest" labels are only used in Auction-Space to catch any unassigned parameters from other rule sets. The higher priority labels are used any time an exception is needed to a standard rule. An example of this occurs in the Hotel section of the Trading Agent Competition rules, included in Appendix "Trading Agent Competition Contract Template." "Highest" rules are used for constraints.[7] It is from these priority rules that we know, for example that the rule labels in the example 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 last section in Auction-Configuration simply specifies that only one value may be inferred for each auction parameter. Which value to infer (when there are multiple possible) is determined by the conflict resolution rules (see Section "Courteous Logic Programs as KR").

Auction-Space

The first thing that Auction-Space (see Appendix "Auction-Space Ruleset") specifies are the domains of each of the auction parameters, as well as defaults for each of them. The parameterization is based initially on AuctionBot but extended and improved in more recent work (Wurman, Walsh, & Wellman 1998). 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, 1).
<lowest>
  auction(multipleSellers, 0).
<lowest>
  auction(tiebreaking, fifo).
```

In the next section of Auction-Space, we specify *conditional* default parameters—that is, what certain parameters should default to, given that certain other parameters have already been inferred. For example, if we know that an auction has a single seller then, by default, it should have multiple buyers, and vice versa.

```
<verylow>
auction(?ID, multipleBuyers, 1)
<-
auction(?ID, multipleSellers, 0).

<verylow>
auction(?ID, multipleSellers, 1)
<-
auction(?ID, multipleBuyers, 0).
```

Next are hard constraints between parameters. For example, if there is bidding rule that says one must meet the current quote, then this implies that one need not *beat* the quote.

```
<highest>
auction(?ID, beatQuote, 0) <-
  auction(?ID, meetQuote, 1).
```

Notice that constraints are similar to conditional defaults except that constraints have overriding priority, while conditional defaults are just that—defaults that will be overridden by values inferred elsewhere.

The remaining rules rules all involve negotiationType/2 predicates in the body. NegotiationType is used in a contract to specify meta-level information for the negotiation mechanism. Auction-Space maps such knowledge to specific auction parameters, which correspond loosely to the parameters in AuctionBot, but that mapping is completed in AuctionBot-Mapping (Section "AuctionBot-Mapping"). Note that negotiationTypes can infer other negotiationTypes but that the inferencing must trickle down to auction predicates eventually. For example, negotiationType(continuous) implies, among others, negotiationType(continuousClears) which in turn implies auction(quoteMode, bid). Following are the rules for inferring continuous quotes and clears:

```
negotiationType(?ID,
                continuousQuotes)
AND
negotiationType(?ID,
                continuousClears)
<-
negotiationType(?ID,
                continuous).

auction(?ID, quoteMode, bid)
<-
negotiationType(?ID,
               continuousClears).

auction(?ID, intClearMode, bid)
<-
negotiationType(?ID,
               continuousQuotes).
```

One particularly useful feature of Auctionbot-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 that define an auction as a CDA—chronological matching, continuous quotes (bid-ask) and clears, double-sided, and discrete goods.

[7] They may also be used outside of any particular rule set—for example, when creating a batch of auctions based on the same rule set but for which one or two parameters should change for each auction.

```
<m>
auction(?ID, matchingFunction,
        earliestTime)
AND
negotiationType(?ID, continuous)
AND
negotiationType(?ID, double)
AND
auction(?ID, divisible, 0)
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 don't close until ten minutes of inactivity have passed.

```
<ebay>
auction(?ID, matchingFunction,
        mthPrice)
AND
auction(?ID, multipleBuyers, 1)
AND
auction(?ID, multipleSellers, 0)
AND
auction(?ID, divisible, 0)
AND
negotiationType(?ID, revealAll)
AND
auction(?ID, bidRules,
        [bidQty1, bidWithdrawr,
         bidQuoteb, bidPrev2,
         bidPrev5])
AND
negotiationType(?ID,
                allNotifications)
AND
negotiationType(?ID,
                continuousQuotes)
AND
auction(?ID, quotePolicy,
        askOnly)
AND
auction(?ID, quoteIncrement, 1)
AND
auction(?ID, intClearMode, none)
AND
auction(?ID, finalClearMode,
        fixed)
AND
auction(?ID, matchingFunction,
        mthPrice)
<-
negotiationType(?ID, ebay).

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

<amazon>
auction(?ID, finalClearMode,
        inactivity)
AND
auction(?ID,
    finalClearInactivityInterval,
        600)
      /* 10 minutes inactivity */
<-
negotiationType(?ID, amazon).

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

This hierarchy could be extended further by making an eBay auction a special case of a standard English auction.

Auctionbot-Mapping

The Auctionbot-Mapping (see Appendix "AuctionBot Ruleset") is a set of rules for inferring AuctionBot parameters from the improved and generalized parameterization given in Auction-Space.

The reason this is necessary is that we have continued to improve, clean up, and generalize our parameterization of auction design space (Wurman, Wellman, & Walsh to appear) but the AuctionBot has not kept up, due to backward-compatibility constraints. For example, in the Auction-Space rule set, we have added to the auction parameterization by introducing an additional auction parameter, `matchingFunction` (either price-based—Mth, $M+1$st—or based on time of bid). AuctionBot does not recognize this parameter but it is used in AuctionBot-Mapping to derive the superfluous "auction type" parameter which the AuctionBot does currently need. In this way, all other rule sets can ignore "auction type" and instead use the interface provided in Auction-Space—one that uses such existing parameters as quote and clear mode, along with the added parameter `matchingFunction`. When the AuctionBot parameters are changed to reflect this new parameterization, Auction-Space will already support it, and the rules to infer the old "auction type" can (optionally) be deleted.

Following is a rule that maps the fundamental parameters defining a Vickrey auction to the deprecated AuctionBot "type" parameter value corresponding to Vickrey auctions.

```
<m>  /* Vickrey, sealed-bid */
auctionbot(?ID, type, 3)
<-
auction(?ID, matchingFunction,
        mPlusFirstPrice)
AND
auction(?ID, finalClearMode,
        synchronized)
AND
auction(?ID, multipleSellers, 0)
AND
```

```
auction(?ID, quoteMode, 0)
AND
auction(?ID, intClearMode, 0).
```

Other auction types such as CDA and Chronological Match can be inferred similarly. Note that CDA is a special case of Chronological Match. This is elegantly captured in CLP.

```
<chronmatch>
auctionbot(type, 4)
<-
auction(matchingFunction,
        earliestTime).

<cda>
auctionbot(type, 5)
<-
auction(matchingFunction,
        earliestTime)
AND
auction(intClearMode, bid).

/* special case */
overrides(cda, chronmatch).
```

Domain-specific Rules: Widget Example

In this section we present a simple example of a subset of a contract from which our prototype is able to automatically configure the appropriate auctions. Section "Domain-specific Rules: Trading Agent Competition" describes a more elaborate domain. In this example, there is only one component of the contract (a widget) and it has only one attribute (quality) with two possible values (regular and deluxe). (There are no alternative negotiation structures.) This information is represented with the following rules:

```
/* Possible Values: */

value(quality, regular).
value(quality, deluxe).

/* Specify the components and
   their attributes: */

component(widget).
attribute(widget, quality).
```

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

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

A widget is a multiattribute negotiable but the current AuctionBot only supports single-dimensional auctions (negotiating price and quantity). A brute-force method for implementing a multiattribute auction is to simply create an array of single-dimensional auctions, one for each point in attribute-value space. The following rule enumerates all the points in attribute-value space for all components—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: (This rule is actually unnecessary in ContractBot because inferring valueTuples is done automatically as part of Auction-Configuration.)

```
valueTuple(?Component,
           dotOp(?Quality,nil))
<-
value(?Component, quality,
      ?Quality).
```

Note that the `dotOp/2` predicate is used to represent a list (dotted pair) since the current implementation of CLP does not support lists explicitly. 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 AuctionbotMapping and Auction-Space to generate the full set of auction parameters for widget auctions. (In this case, most of the parameters will be default values specified in Auction-Space.)

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

At this point, we have inferred all of the auction parameters for widgets and we have enumerated the valueTuples for all the auctions we need to create. We now combine those steps to explicitly create the auctions and have each of the auctions created get its parameters from the parameters we 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. This tells us the component that each auction belongs to. (Inferring the set of auctions from the valueTuples is also done automatically in the Auction-Configuration rule-set, as well as inheritance of parameters from parent components.)

```
makeAuction(dotOp(?Component,
                  ?Values))
AND
parent(dotOp(?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).
```

Domain-specific Rules: Trading Agent Competition

In July 2000 in Boston at the International Conference on Multiagent Systems, the University of Michigan is hosting a trading agent competition in which participants will write agents to do automated trading in a set of auctions on the AuctionBot. The auctions will simulate the domain of a travel agent assembling trips for its customers. The goods that agents will be shopping for are flights (defined by day and destination—out or back), hotels (defined by day and quality), and entertainment tickets (defined by day and type of event). Each of the types of goods are sold in a different kind of auction. Flights are sold at randomly fluctuating fixed prices. Hotels are sold in an ascending English auction. Agents buy and sell entertainment tickets in a continuous double auction much like trading securities in a stock exchange.

The design of the Trading Agent Competition (TAC) game describes in detail how the goods are partitioned into a set of auctions and exactly what the parameters of those auctions are. Appendix "Trading Agent Competition Contract Template" is a rule set that generates that partitioning (among a space of possible partitionings) and auction configurations based on a higher-level description of the TAC game. It also includes rules from buyers and sellers, indicating how they would like the contract to be partitioned into components. As given, the TAC contract template and buyer/sellers rules will infer the same components described above—the partitioning used in the actual TAC competition—but simple changes to buyer or seller rules will infer alternative structures for hypothetical TAC negotiation mechanisms. For example, we currently have rules from multiple buyers expressing willingness to buy bundled travel packages (flight, hotel, and entertainment bundled into one good). Adding a rule from a single seller expressing willingness to sell such a good would result in a repartitioning of the set of auctions to include complete travel packages.

The first thing the TAC contract template specifies is a proto-contract. As described in Section "Contracting Framework," 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. As mentioned in Section "Composing Final Contracts," we show a typical rule for a proto-contract that we have included in the TAC example, namely, inferring the total amount that a given agent owes another agent after the negotiation:

```
pay(?Agent1, ?Agent2, ?Amt) <-
   setof(?Pay12,
         transact(?Agent1,
                  ?Agent2,
                  ?Component,
                  ?AVList,
                  ?Pay12, ?Qty),
         ?Pay12List) AND
   setof(?Pay21,
         transact(?Agent2,
                  ?Agent1,
                  ?Component,
                  ?AVList,
                  ?Pay21, ?Qty),
         ?Pay21List) AND
   sum(?Pay12List, ?Pay12Total)
   AND
   sum(?Pay21List, ?Pay21Total)
   AND
   is(?Amt, minusOp(?Pay12Total,
                    ?Pay21Total)).
```

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.[8] Following is a part of the utility calculation which says that a client's utility is a function of whether they were able to procure a trip, how many days deviation from their ideal travel dates they were, and their bonuses for staying in the nice hotel and seeing the entertainment they wanted:

```
<high> clientUtility(?Client, 0)
     <- feasibleTrip(?Client, 0).
<m>
clientUtility(?Client, ?U) <-
  feasibleTrip(?Client, 1) AND
  travelPenalty(?Client, ?TP) AND
  hotelBonus(?Client, ?HB) AND
  funBonus(?Client, ?FB) AND
  is(?U, 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 by transaction facts generated by the ContractBot as it monitors the auction results. We can now infer a travel agent's utility in the competition by summing the utilities of its clients and subtracting its expenses:

```
utility(?TravelAgent, ?U) <-
  setof(?Client,
        clientOf(?Client,
                 ?TravelAgent),
        ?ClientList) AND
  map(clientUtility, ?ClientList,
      ?ClientUtilities) AND
  sum(?ClientUtilities,
      ?Profit) AND
  expenses(?TravelAgent,
           ?Expenses) AND
  is(?U, ?Profit - ?Expenses).
```

Note that determining the expenses for an agent is a variation on the pay predicate given above. (See Appendix

[8] 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—i.e., to determine the winner of the competition.

"Trading Agent Competition Contract Template" for details.)

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:

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

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

```
attribute(flight, type).
attribute(flight, day).
  value(flight, day, ?Val)
    <- value(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 `value/2` predicates.

By using the description of the possible components of the contract, along with other rules about how to split the components into a set of auctions, the procedure determines the groups of auctions to create. There are also rules that help infer what the parameters of those auctions should be. For example, the following rules specify that hotels should be auctioned "eBay style," with the exception that buyers can bid for multiple quantities, and in fact submit entire discrete demand schedules—a list of quantities demanded for each of a set of prices.

```
<m> negotiationType(hotel, ebay).
<high> auction(hotel, bidRules,
  [pqPoints, noWithdraw,
   beatQuote, beatPreviousBid]).
```

The TAC contract template has sections for several components besides flights, hotels, and entertainment. It also has round trip flights (parameterized by arrival day and departure day), hotel blocks (which have a type as before, plus first and last night for a contiguous range of rooms), flight/hotel bundles (with attributes for day in, day out, and hotel type), entertainment packages (which bundle a set of entertainment tickets for a trip), and finally, complete travel packages (parameterized by arrival, departure, type of hotel, and when each type of entertainment is to be seen, including never).

By stating relationships between these components and incorporating rules from buyers and sellers, we can reason about alternative negotiation structures for TAC. The relationships we encode in the current example are mutual exclusivity rules about which components an agent would never be interested in simultaneously. For example, individual one-way flights and flight/hotel bundles are considered mutually exclusive:

```
mutex_head <- component(flight)
  AND component(flighthotel).
```

And the travel package component subsumes all other possible components:

```
mutex_head <-
  component(travelpackage) AND
  component(?X)
  mutex_given
  notEquals(?X, travelpackage).
```

Prototype Implementation

In this section we discuss our implementation of the overall contracting process described in Section "Contracting Framework." Figure 2 depicts the overall process of turning a contract template along with rules from buyers and sellers into a final contract, and thus an executed deal. At the heart of this process are the three sets of background knowledge discussed in Section "Courteous Logic Programs for Configuring Auctions"—Auction-Configuration, Auction-Space, and AuctionBot-Mapping. ContractBot.clp wraps these rulebases together along with a file of miscellaneous utilities (util.clp) and the Prolog (XSB) queries that drive the inferencing.

The inferencing engine itself is actually a series a Perl scripts that guide the input to ContractBot and the background knowledge through the inferencing engines. 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 then combined by another script 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 to interact with the AuctionBot. For example, to generate the list of auctions to be created, contractBot.clp makes the following query:

```
:- setof(ID, makeAuction(ID), L),
   writelist(L), nl.
```

This simply 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 point in attribute space for every component inferred. Components, in turn, are inferred from the contract template and from buyer and seller rules.

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 Mathe-

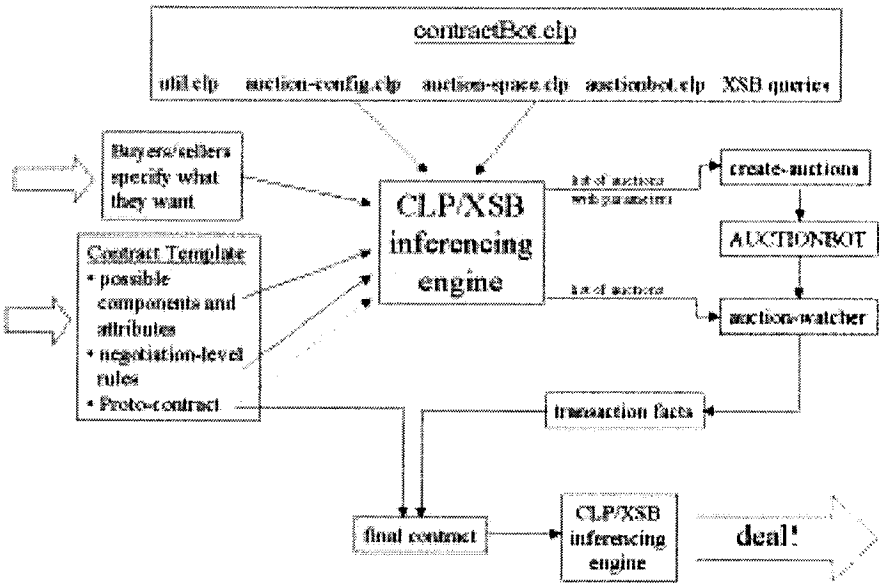

Figure 2: How ContractBot uses its auction knowledge to turn a partial contract into a complete, executable contract.

matica implementation of the AuctionBot's API[9] 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 "Contracting Framework" and "Domain-specific Rules: Trading Agent Competition") 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 inferencing engine to execute the terms of the deal.

Section "Domain-specific Rules: Widget Example" presents a simple example of a partial contract that can be used as input to our prototype. In Appendix "Trading Agent Competition Contract Template" we provide a more elaborate example which generates the set of auctions used in an upcoming Trading Agent Competition. The domain for the competition involves three components—flights, hotels, and entertainment—each of which has two attributes, type and day. Our prototype is able to reproduce the same set of auctions actually used in the competition, but using only a high-level description of the goods to be negotiated and the nature of the negotiations.

Related Work

This work builds on a project at IBM T. J. Watson Research called Business Rules for Electronic Commerce (BREC). (http://www.research.ibm.com/rules/) Its goal is to support the encoding of business rules using Courteous Logic Programs (Grosof 1997). This is the basis for the representation of business contracts that we are using here. Note that this work differs from existing work under similar names. Notably, Tuomas Sandholm's Contract Net and other work in distributed AI and industrial engineering describe mechanisms for subcontracting among agents in order to divide work in accomplishing a task. By contrast, our approach is to support an automated negotiation mechanism for agents to decide upon agreeable terms of a contract, which can then be executed electronically.

Multidimensional negotiation is the other aspect of this work which has an existing literature. As discussed in Section "Auction-Based Negotiation," combinatorial auctions allow bidders to make offers for combinations (bundles) of goods. We are aware of one prototype that supports combinatorial auctions on the Internet (Sandholm to appear). Multiattribute auctions have typically been used for procurement (one buyer, many sellers). They allow bidders to express willingness to buy over an entire space of attributes of a single good (Branco 1997).

Conclusion and Future Work

This paper builds on our framework for the overall contracting process (Reeves *et al.* 1999) by implementing a system for automatically configuring a negotiation mechanism based on a formal description of a partial contract and interpreting the results of the negotiation to form a complete and executable contract. We present an infrastructure

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

for configuring negotiations and carrying out the resulting contracts. The background knowledge supporting this infrastructure is embodied in three CLP rule sets: Auction-Configuration, Auction-Space, and Auctionbot-Mapping. Auction-Configuration supports reasoning about alternative negotiation structures and how to split contract into an array of auctions. Auction-Space implements a cleaner, more general parameterization of the auction design space, imposes constraints and conditional defaults on the parameters, and infers auction parameters from higher-level knowledge about a negotiation. AuctionBot-Mapping maps the Auction-Space parameterization to the existing set of AuctionBot parameters.

Our prototype can generate sets of auctions corresponding to a multicomponent, multiattribute negotiation, and supports reasoning about alternative ways to decompose a contract into components and attributes. A simple example of this is discussed and we also use the prototype to generate the auctions for the upcoming Trading Agent Competition. We also discuss additional uses for this prototype as a general way to programmatically create sets of auctions, not just in the contracting context.

One piece of future work on ContractBot 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. This will also entail further work on the contracting infrastructure, such as richer mechanisms for aggregating agent preferences in configuring negotiations.

We would also like to extend AuctionBot and thus our ontology in ContractBot to support richer negotiation mechanisms than the current naive approach to handling multiple attributes of a component—creating an array of auctions, one for every combination of attribute values. This is not tractable and needs to be augmented with multiattribute and/or combinatorial auctions.

Finally, we would like to use the knowledge bases we are developing to drive the back end of human interfaces for auction creation—such as the CGI interface to AuctionBot. For this we need to add rules to infer an order for asking the user auction parameters, and the CLP engine will need to be run after each input so that constraints and defaults can be propagated.

References

Baral, C., and Gelfond, M. 1994. Logic programming and knowledge representation. *Journal of Logic Programming* 19,20:73–148.

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

IBM CommonRules. http://www.research.ibm.com/rules/commonrules-overview.html.

Friedman, D., and Rust, J., eds. 1993. *The Double Auction Market.* Addison-Wesley.

Grosof, B. N.; Labrou, Y.; and Chan, H. Y. 1999. A declarative approach to business rules in contracts: Courteous logic programs in XML. In *ACM Special Interest Group on E-Commerce (EC99)*, 68–77.

Grosof, B. N. 1997. 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).* P.O. Box 137, Blackpool, Lancashire, FY2 9UN, UK. http://www.demon.co.uk/ar/PAAM97: Practical Application Company Ltd.

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

Reeves, D. M.; Grosof, B. N.; Wellman, M. P.; and Chan, H. Y. 1999. Toward a declarative language for negotiating executable contracts. In *AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce (AIEC-99).*

Sandholm, T. to appear. Approaches to winner determination in combinatorial auctions. *Decision Support Systems.*

Wellman, M. P., and Wurman, P. R. 1999. A trading agent competition for the research community. In *IJCAI-99 Workshop on Agent-Mediated Electronic Trading.* See also, http://tac.eecs.umich.edu/.

Wurman, P. R.; Walsh, W. E.; and Wellman, M. P. 1998. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems* 24:17–27.

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

Wurman, P. R.; Wellman, M. P.; and Walsh, W. E. to appear. A parameterization of the auction design space. *Games and Economic Behavior.*

Auction Configuration Ruleset

See http://ai.eecs.umich.edu/people/dreeves/autonego/auction-config.clp

Auction-Space Ruleset

See http://ai.eecs.umich.edu/people/dreeves/autonego/auction-space.clp

AuctionBot Ruleset

See http://ai.eecs.umich.edu/people/dreeves/autonego/auctionbot.clp

Trading Agent Competition Contract Template

See http://ai.eecs.umich.edu/people/dreeves/autonego/tac.clp