Formally Defining Coordination Processes to Support Contract Negotiation

Aaron G. Cass Hyungwon Lee Barbara Staudt Lerner Leon J. Osterweil

Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts
Lederle Graduate Research Center
Amherst, MA 01003

{acass, hlee, blerner, ljo}@cs.umass.edu

Abstract

The literature and practice of negotiation and auctions, especially in the burgeoning area of electronic commerce, demonstrate that there is a wide and rapidly growing variety of negotiation and auction processes both in use and proposed. We believe that in the future automated online auctions will become a fundamental building block for contract negotiation carried out electronically. To avoid loss of money or bad decision making, it is important for organizations to have high confidence in the software involved in these activities.

We are writing a range of negotiation processes in Little-JIL, an agent coordination language that addresses goals of expressiveness, analyzability, and executability. With Little-JIL, we can express processes involving the coordination of the multiple participants involved in contract negotiation and do so in a syntax that allows an intuitive understanding to non-programmers. Furthermore, the language has a formal semantics that allows us to prove certain properties of the negotiation processes. Finally, the language also has executable semantics allowing us to directly execute the validated processes. We believe that this combination of capabilities makes Little-JIL a valuable language for the definition of distributed, multi-agent processes for which we want high assurance.

1. Introduction

The literature and practice of negotiation and auctions, especially in the burgeoning area of electronic commerce, demonstrate that there is a wide and rapidly growing variety of negotiation and auction processes both in use and proposed. Currently, most online auctions, such as eBay, are used primarily by hobbyists. We believe that in the future online auctions will not simply be tools for bargain hunters and collectors, but they will become a fundamental building block for contract negotiation carried out electronically. Furthermore, we expect many buying and selling decisions to be based on automated negotiation and auctions where all parties involved, buyers, sellers, and auctioneers, will be automated. To avoid loss of money or bad decision making, it is important that we have high confidence in the software involved in these activities. Contract negotiation will always remain a highly decentralized activity as the participants will always be separate processes communicating to achieve a goal. To gain high confidence about these requires us to understand the distributed nature of the domain and to define the acceptable boundaries within which each operates. An important aspect to understand, then, is the coordination of these distributed activities.

Submitted to ACM Conference on Electronic Commerce (EC-99), November 1999, Denver, Colorado.

We believe that if automated contract negotiation is to take on a large role in the economy, contracting processes must have the following properties: 1) they must be **correct**, not containing errors such as deadlocks, or improper handling of exceptional conditions, 2) they must be **reasonable**, allowing agents adequate time to bid for tasks, and awarding work based upon consideration of all bids received, for example, 3) they must be **robust**, being resistant to the negative impacts of agents who do not meet their commitments, either inadvertently or maliciously, 4) they must be **fast**, executing quickly in both absolute time, and relative to the time it takes to execute the task that has been negotiated for. These desiderata seem fairly and profitably viewed as requirements to be met by actual contract negotiation processes.

Existing negotiation processes found in the literature are generally described informally, using natural language. Their semantics are precisely defined through their implementations, but the implementations typically undergo minimal analysis and testing to ascertain their correctness and performance characteristics. Although the negotiation processes could be captured in many notations, a high level notation that provides abstractions to define the coordination of multiple agents is required to express the processes of interest. A notation with well-defined, well-formed semantics can facilitate verification of desirable properties using software analysis techniques. In addition, if the notation is executable, we can be assured that the properties understood informally and proven through analysis are not lost as the process is translated to an executable language.

For developers to reuse negotiation processes effectively, it is necessary that the reusable structure of negotiation be cleanly separated from domain-specific behavior. Domain-specific behavior should be captured in the domain-specific agents while reusable negotiation behavior should be captured in reusable processes. Once such reusable processes are defined, it is necessary for them to be collected and organized in a manner that facilitates identification of suitable processes and comparison of similar processes. Again, we believe that efforts to develop a library of negotiation processes will be more fruitful if the informal descriptions of the processes are supplemented with a definition written in a language that facilitates communication of the processes in a more precise manner than natural language allows, is simple enough to understand to allow informal reasoning, yet is executable and based upon a well-defined semantics that enables automated analysis tools to prove certain properties.

We are writing a range of negotiation processes in Little-JIL, an agent coordination language that addresses all these goals of expressiveness, analyzability, and executability. This experience has demonstrated that it is relatively straightforward to reuse common parts of negotiation processes and to discern various key distinguishing features of different negotiation processes informally by simple inspection. Moreover, there are numerous properties that are not as easy to extract, but are key distinguishing features that should and could be proven using formal analysis.

2. Related Work

In the area of multi-agent systems, there is much work that builds on Smith's Contract Net [1]. Contract Net is a protocol used to decentralize task allocation in a distributed, multi-agent system. Instead of requiring a centralized controller that allocates tasks to subordinates, Contract Net allows each agent to contract out tasks via a negotiation mechanism. Smith defines a set of message types that use the following fixed negotiation process: a manager announces a task to be performed, agents decide whether to bid to perform the task, then the manager chooses the best agent (using its own ranking criteria). Because the process is fixed in the implementation, there is no formal description (aside from the code) from which to perform any formal analysis about the suitability of this process for a particular application. The fixed process also limits the ability to change or reconfigure the process quickly and reliably.

Sandholm [2] extends Contract Net by developing an economic model based on marginal cost. This economic model is the basis for all task allocation decisions, allowing cooperative and competitive agents to work together. This work formally defines the allocation policy but does not formally define the allowable sequences of events — thus while it is possible to reason about allocation decisions, it is still not possible to reason about the negotiation process.

Sandholm and Lesser [3] later describe a Contract Net-derived process using a state-transition diagram approach. States in the diagram are not labeled, but represent the overall states of the process, while the transitions represent the agent actions (for example, "contractee proposes"). Their intent is to classify negotiations, so the descriptive nature of the state transition diagram approach is appropriate. The approach describes how the state of the process progresses in response to actions, but does not prescribe the order of those actions. Their notation allows reasoning over sequences of actions, but it is purely a modeling notation and is not directly executable.

Wurman, Wellman, and Walsh's [4] AuctionBot work introduces a parameterized taxonomy of auctions. By specifying the values (from a pre-defined allowable set) for different auction parameters (such as Bid Rules and Clearing Schedule), a user of the AuctionBot system can describe an auction that can be executed in the system. In a similar spirit, Kumar and Feldman [5, 6] describe the design of a class hierarchy that supports the construction of a variety of auctions operating with different rules. Both AuctionBot and the work by Kumar and Feldman support some classical auction models, such as sealed bid, English, and Dutch auctions, but also support the creation of new auction types by providing highly composable and customizable components. This offers a great deal of flexibility in the creation of auctions, but may leave the creator uncertain as to whether the combination of components and customizations used result in an auction with the desired properties.

We build upon definitions of negotiation and auctions described in the earlier work, but we focus our attention on describing the processes with a notation that facilitates both informal and formal analysis to validate that the described processes have the desired properties. Rather than focus on the functionality and variety of the processes, we are interested in studying their implementations from a software engineering perspective to assure their quality before trusting them with carrying out contract negotiation.

3. Overview

We are developing a library of contracting processes written in Little-JIL[7][8], an agent coordination language that supports specification, execution, and analysis of processes involving multiple agents. Little-JIL is a graphical language in which processes are decomposed hierarchically into steps. The steps are connected to each other with control flow and data flow edges. Each step is assigned to an agent to perform. In the context of contract negotiation, an agent might be an entity selling goods or services or buying goods or services, while the process itself defines the coordination of the agents involved in the negotiation. The collection of steps assigned to an agent defines the interface that the agent must satisfy to participate in the negotiation. Properties can be proven about the behavior of a negotiation process by reasoning about the data flow and control flow connections among the steps and analyzing the ordering and concurrency of their execution.

One key feature of Little-JIL that simplifies reasoning about distributed proceses is that the process interleaves the activities of the agents thereby making the synchronization between the distributed processes explicit. In a more traditional language, each agent's activities would be described in a separate syntactic unit. Interactions among the threads of activities belonging to individual agents appear as message passing and synchronization of the threads. In Little-JIL, these are represented directly with control flow and data flow edges. In Little-JIL, agents still operate in separate threads and the synchronization and message passing is still required, but it is provided by Little-JIL's interpreter rather than being defined in the code implementing the agents. The result is simpler process code, increasing our ability to both formally and informally analyze the code.

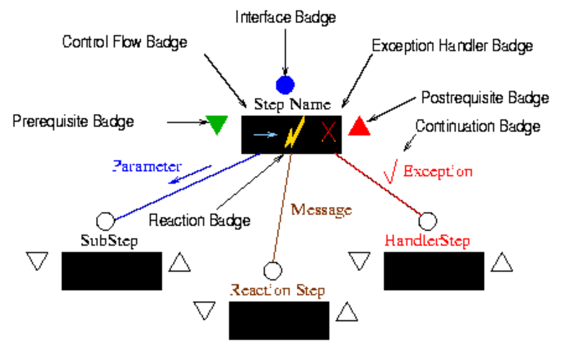

Figure 1: Little-JIL Syntax

3.1. Overview of Little-JIL

A Little-JIL process is represented as a hierarchical decomposition of steps. The graphical syntax of a Little-JIL step is shown in Figure 1.

This figure shows the various badges that make up a step, as well a step's possible connections to other steps. The interface badge at the top is a circle by which this step is connected to its parent. The circle is filled if there are local definitions associated with this step, and is empty otherwise. The interface includes the declaration of the agent who is to carry out the step, resource requirements of the step, variable declarations, exceptions that may be thrown, and messages that may be sent.

Below the circle is the step name. To the left is a triangle called the prerequisite badge. The prerequisite is a step that must be successfully completed for this step to begin execution. If the prerequisite is not completed successfully, an exception is thrown and the step is not allowed to execute. The badge appears filled if the step has a prerequisite step, and an edge may be shown that connects this step to its prerequisite (not shown). On the right is another similarly filled triangle called the postrequisite badge. The postrequisite step begins execution immediately after the step completes execution and must also successfully complete for the parent to be notified of the step's completion. If the postrequisite does not complete successfully, an exception is thrown.

Within the box below the step name are three more badges. From left to right, they are the control flow badge, reaction badge, and exception handler badge. Substeps are connected to the control flow badge. The edge connecting a step to a substep is annotated with parameter passing information. The control flow badge indicates the order in which the substeps may be executed. Little-JIL defines four control flow badges as shown in Figure 2. A sequential step executes its substeps in order from left to right, beginning the next substep only after the preceding substep completes successfully. A parallel step allows the substeps to be executed concurrently. A sequential

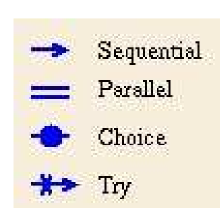

Figure 2: Control flow

or parallel step requires all of its substeps to be performed. A choice step allows the agent performing the step to choose which single substep to execute. A try step identifies alternative ways of performing the step but hardwires the order in which the alternatives should be tried from left to right. A choice or try step requires exactly one of its substeps to be performed successfully. A step with no control flow badge is a leaf step and is directly executed by an agent. A step whose name is in italics is a reference step and is defined elsewhere in the process.

The lightning bolt in the middle is the reaction badge to which reaction steps are attached. A reaction identifies a broadcast message that it responds to.

Figure 3: Exception Continuation Badges

The X is the exception handler badge to which exception handlers are attached. An exception handler identifies the exception that it is handling, optionally a step to perform to handle the exception, and a continuation badge to indicate what to do after completion of the handler step. There are four continuation badges as shown in Figure 3. The continue badge indicates that the execution of the step should continue. For sequential and parallel steps, this is as if the substep that threw the exception completed successfully. For choice and try steps, this allows an agent to perform a different alternative. The restart badge indicates that the entire step should be restarted from the beginning. The complete badge indicates that the entire step should be considered successful. The rethrow badge indicates that the entire step should be considered unsuccessful and the exception should be thrown again to the step's parent.

The control flow, reaction, and exception badges are hidden if there are no child steps, reactions, or handlers, respectively.

It is possible to attach timeouts to steps. Timeouts come in two varieties. A *deadline* timeout, represented with a clock icon, denotes an absolute time by which a step must be complete, such as June 1, 1999, midnight. An *interval* timeout, represented with an hourglass, denotes a duration of time that is measured from one event to another. Typically, an interval timer begins counting down when the associated step is started. For instance, an agent may be given one hour to complete an activity.

Each step in a Little-JIL process has an execution agent. Either one is declared locally as its agent or one is inherited from the parent. When a Little-JIL process is executed, the step is assigned to the agent to perform. The agents carry out the activities associated with the leaves of the process and report back to the Little-JIL interpreter upon completion of a step. The process thus controls the interleaving of agent activities by assigning different activities to different agents, using control flow edges to synchronize the agents and data flow edges and shared memory to communicate information among the agents. The synchronization primitives and communication primitives required to carry out the high-level synchronization and communication specified in the process are provided by the interpreter, relieving the process programmer from providing this error-prone code.

3.2. Benefits of Using Little-JIL

We hypothesize that with negotiation processes appropriately described in Little-JIL, we will be able to:

Intuitively understand processes — because Little-JIL is a concise visual language that makes communication among distributed processes straightforward, people will be able to get an intuitive grasp of the negotiations described using Little-JIL. In particular, non-programmers will be able to understand the processes, at least at this intuitive level.

Compare negotiation processes — by formally describing several negotiation processes in the same formalism, we will be able to compare the processes more easily.

Analyze negotiation processes — with a formal foundation, we can pose questions about the negotiation processes to be validated. We can formally prove that certain processes avoid deadlocks or race conditions, for example.

Separate higher level concerns from domain-specific concerns — by formally describing a set of negotiation processes separate from domains of applicability, we will provide the foundation on which domain-specific negotiations can be described. The aspects of the processes that are domain-independent will be captured in Little-JIL, while the domain-specific aspects can be captured separately, either in separate processes or encapsulated within agent behavior.

Reuse portions of negotiation processes — because we separate the agents doing the work from the work to be done, we will be able to reuse the descriptions of the work to be done in situations where different agents are doing the work. The steps and their organization into a process can be reused while changing the agents that are performing those steps.

Execute negotiation processes — because Little-JIL has an interpreter, negotiations described in Little-JIL can be directly executed. Furthermore, this ensures that any properties proven using formal analysis are guaranteed to hold during process execution.

4. Sample Auction and Negotiation Processes

In order for a buyer and seller to reach an agreement, they must first explore whether they wish to work together, announce their intent to work together, and then negotiate the exact terms with which they will work together. We are thus exploring a variety of *contracting* processes that contain within them *auctioning* or *bidding* subprocesses and *negotiation* subprocesses. These subprocesses can be defined and composed in a variety of ways resulting in different contracting situations. For example, we can model sealed bid contracts such as the government typically uses, as well as open-cry auctions more typical of short duration, face-to-face auctions. In both cases, incorporating a time limit on the bidding or auctioning activities is critical and Little-JIL allows us to do so. Furthermore, formalizing behavior when negotiations fail is also important to represent. For example, what should we do if nobody bids? What should we do if the winning bidder fails to deliver the service negotiated for? Handling of these exceptional conditions can also be formalized and reasoned about in Little-JIL.

Negotiation naturally involves multiple parties: the agents who are requesting services as well as the agents who can perform those services. Agents are autonomous and execute in a loosely-coupled distributed environment. Each agent performs decision-making activities: deciding which tasks to request of others, deciding whether or not they are capable of performing a task, considering how winning a bid impacts their ability to perform other current and future tasks. The agents have limited knowledge of each other and how others make decisions. The role of a negotiation process and modeling of agents is to provide coordination and communication among the agents during negotiation, not micromanaging the decision-making activities.

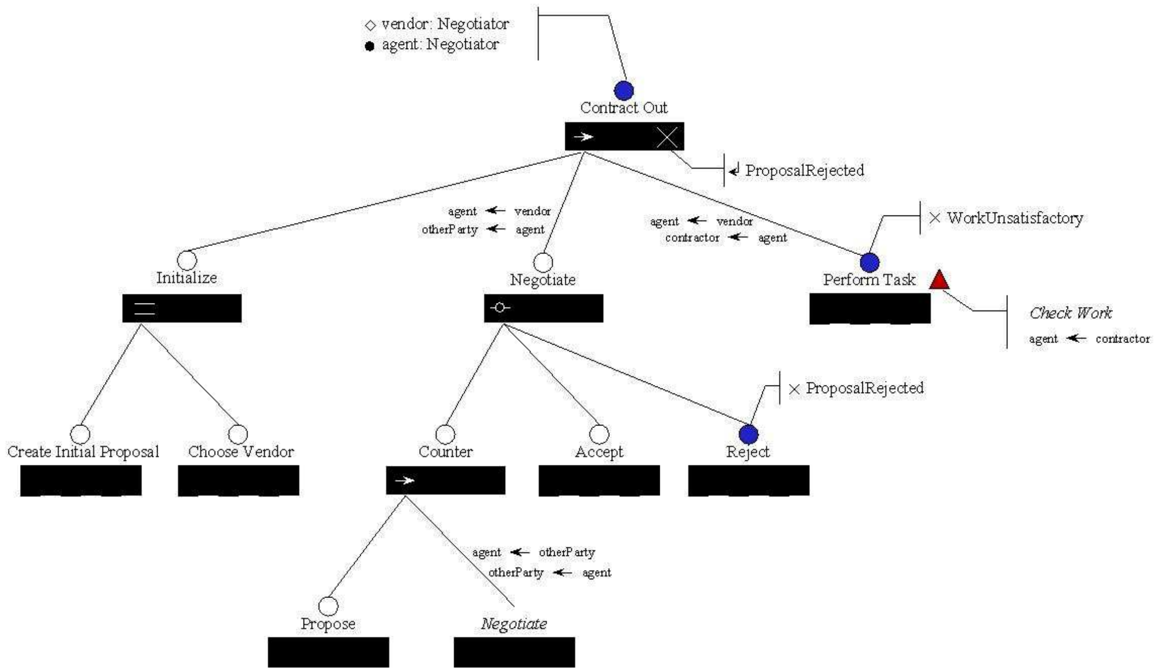

Figure 4: A Simple Negotiation Process

At the lowest level, all negotiations involve communication of a desired task to be performed and a bid for performing that task. The bid may include information such as price, delivery time/date, and perhaps a modification to the task itself. The manner in which an agent decides what bid to make can vary considerably from agent to agent and typically relies extensively on knowledge known only by the agent, such as its ability to procure supplies to perform the task. Similarly, the manner in which an agent decides which bid to accept can vary widely. In some situations, minimizing time of delivery may be most critical, in others minimizing cost may be. In general, however, the decision-making can involve quite complex considerations including the reputation of the agent providing the bid.

4.1. Basic Negotiation Building Block

The role of the process is to provide rules governing the negotiation, a communication mechanism among the agents to carry out the negotiation, and, in some cases, restrictions on how the agents make decisions. For example, a negotiation process might guarantee that only the four lowest cost bidders be considered. A basic building block of all negotiations is the bargaining activity of making proposals and counterproposals, ultimately ending in an acceptance or rejection of a proposal. A Little-JIL process defining a very simple negotiation process incorporating this basic building block is shown in Figure 4.

The top step, `Contract Out`, is a sequential step in which the agent desiring a task to be performed first creates a description of the task and, potentially in parallel, selects an agent to perform the service. These parallel activities are expressed in the `Initialize` step. Then the `Negotiate` step can begin by sending the proposal to the agent that can perform the service. This agent can now negotiate, by choosing to either return a counterproposal via the `Counter` step, accept the proposal as offered via the `Accept` step, or completely reject the job via the

Reject step. On rejection, an exception handler at Contract Out indicates that the step should be restarted. This allows the contractor to revise the proposal itself, select a new vendor to negotiate with, or potentially both. Once the contractor and vendor have reached an agreement, the vendor performs the task. Upon completion, the contractor examines the work to determine if it was done properly, possibly throwing a WorkUnsatisfactory exception that is rethrown by Contract Out because there is no handler for that exception. Performing the task, checking its quality, and deciding what action to take if the results are unsatisfactory is

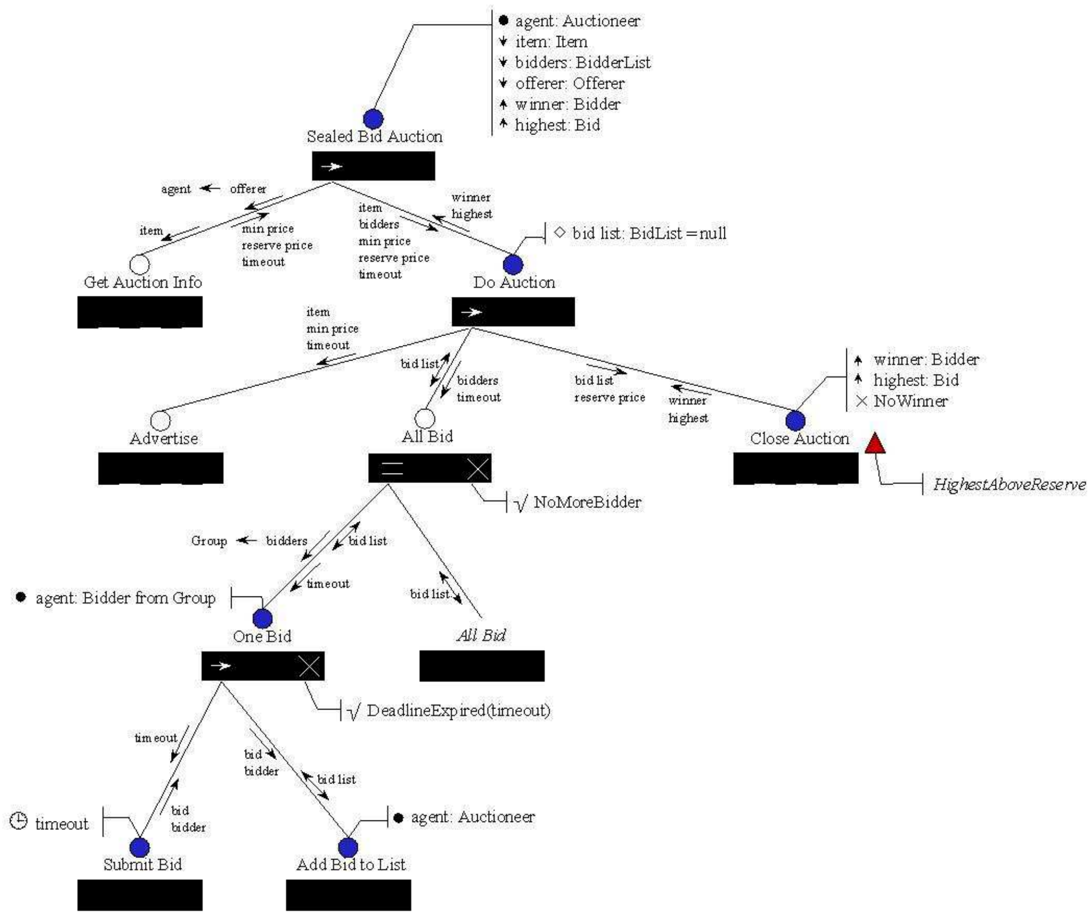

Figure 5: Sealed Bid Auction

outside the scope of the negotiation process, per se, and quite specific to the application. We have found this simple bargaining building block to be quite reusable.

The interleaving of agent activities can be seen in this process in the recursive use of the Negotiate step. The first time that the negotiate step is encountered, the agent is the vendor and who is given the initial proposal to consider. If the vendor produces a counter proposal, the Negotiate step is instantiated recursively. This time, however, the agent is the one that created the original proposal. The otherParty variable is used simply to allow the necessary swapping on the recursive negotiations.

4.2. Sealed Bid Auction

The process in Figure 4 selected a vendor using a method that was not specified in the process. Instead it was represented by the leaf step `Choose Vendor`. One potential mechanism by which a vendor can be chosen is through a bidding process such as the mechanism used by the government in selecting a contractor. We can create a more elaborate contract negotiation process by substituting a bidding process for the `Choose Vendor` step, leaving the remainder of the process untouched. Figure 5 shows a sealed bid auction process as a bidding process. This process starts with obtaining some information characterizing a particular sealed bid auction, such as reserve price, minimum starting price, and deadline for closing the auction, from the offerer who desires a task to be performed. The `Do Auction` step consists of sequential agent activity in which the bidders place bids and then the winner is chosen after the deadline arrives.

The `All Bid` step assigns the bidding activity to each bidder in parallel by assigning the One `Bid` step to an agent and then recursively invoking itself. When there are no more bidders available, that is, when the the `One Bid` step fails to acquire a resource which is to be used as the execution agent for the step, the `NoMoreBidder` exception is thrown and the last recursive step aborts, limiting the parallelism to the number of available bidders. This mechanism is called resource-bounded parallelism and is one of the Little-JIL programming idioms. The `One Bid` step decomposes into a bid submission performed by the bidder followed by a sending of a bid to the auctioneer. The communication occurs as a result of data flow out of the `Submit Bid` step and then in to the `Add Bid to List` step, which is performed by the auctioneer. The interpreter automatically provides the interprocess communication between the bidder and auctioneer. The `All Bid` step is complete when all of its substeps complete. This occurs when all of the potential bidding agents make a bid or the deadline by which bids are due passes, whichever comes first. If a step with a deadline is still executing when the deadline arrives, the step is terminated and the `DeadlineExpired` exception is thrown to the parent. In the `Submit Bid` step, the expiration of the deadline `timeout` results in the termination of the `One Bid` step as indicated by the completion badge on the `DeadlineExpired` handler.

After all bidding activity is complete, the `Close Auction` step starts. The `Close Auction` step is responsible for determining who the winning bidder is. Upon the completion of the `Close Auction` step, the postrequisite `HighestAboveReserve` is checked to determine if the highest bid outbids the reserve price. If the postrequisite is true, the winning bidder and the highest bid are passed to the outside of this process program via control flow edges to the `Sealed Bid Auction` step. The winning bidder becomes the vendor in the remainder of the negotiation process. Otherwise, the `NoWinner` exception is thrown, which is propagated to the parent of the `Sealed Bid Auction` step. The parent in contract negotiation might decide to modify the proposal to attract a different collection of bidders, for example.

The offerer, bidders, and auctioneer would almost certainly be separate (operating system) processes running on separate computers. Most representations of a sealed bid auction would represent these separate threads with separate syntactic entities. With such a representation, it is necessary to examine multiple syntactic entities to see how they interact. Little-JIL allows the interactions to be interleaved. For example, if we want to understand when the bids are examined, it is simple for us to see that bids are examined during the `Close Auction` subprocess, which does not begin until the `All Bid` subprocess is complete. Delaying the examination of bids until all bids have been received is a key property of sealed bid auctions. We can easily inspect the process to assure ourselves that this property exists. We could also use formal flow-based analysis tools, such as Flavers [9], to verify the property formally.

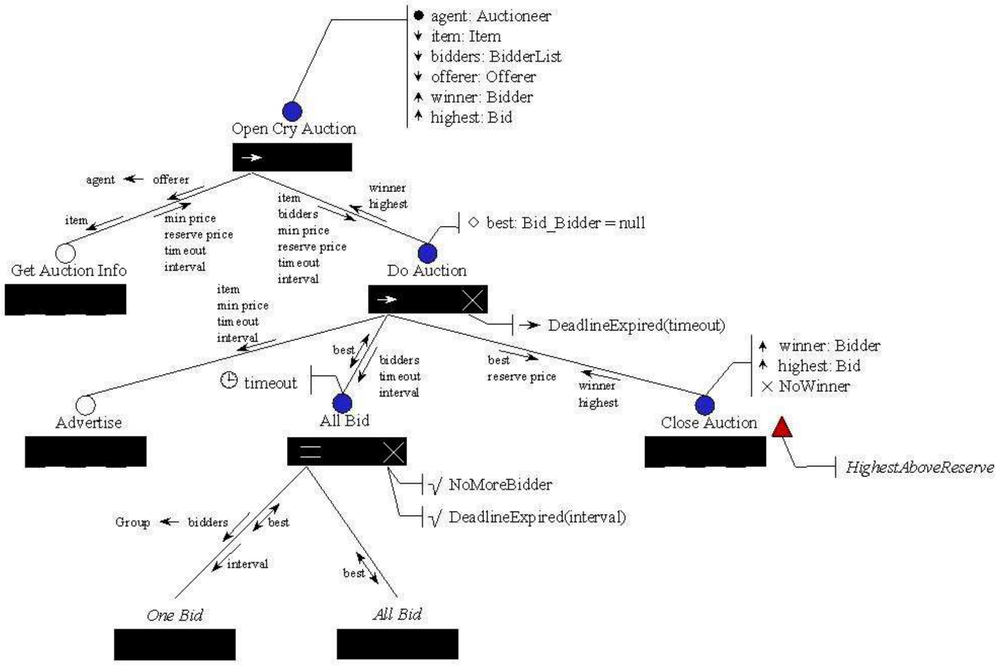

Figure 6: Open-Cry Auction

4.3. Open-Cry Auction

Figures 6 and 7 show an open-cry auction process known also as the English auction. An open-cry auction differs from a sealed bid auction in that it is more interactive. Whenever a bid is made, it is immediately compared to the current best bid. All bidders are kept informed of the current best bid. The auction ends either after a designated amount of time has elapsed since the start of the auction or the length of time between successive bids exceeds some limit.

We can reuse much of the previous sealed bid auction to describe the open-cry auction. In fact, the changes are isolated to the `One Bid` step and the throwing and handling of the deadline `timeout`. In the sealed bid auction, the offerer simply gathered the bids and examined all of them when the auction closed. With an open-cry auction, the offerer must examine the bids as they arrive. Figure 7 shows the `One Bid` step which includes this incremental evaluation activity. After the bidder submits a bid with the `Submit Bid` step, the bid is immediately evaluated. Following its evaluation, the `One Bid` step is recursively invoked allowing the same bidder to bid again. Evaluation of the bid involves updating the current best bid incrementally if the new bid is better. The current best bid is kept in shared memory used by all agents so that the new best bid value will be visible to all agents.

Timeouts are also more complex in this process. An open-cry auction may have a deadline at which it ends, but more typically bidding continues until there is no bidding activity for some time interval. This is represented as an interval timeout in the process. This value is all a shared data item so that all bidders see the same interval timeout. Whenever a new bid is received, the auctioneer resets the interval timer allowing the auction to continue further. Due to the recursive nature of the `One Bid` step, `One Bid` never completes until an exception is thrown that is not handled locally. The only possible exceptions in this process are the interval timer and the deadline expiring. When one of these timeouts occurs, the bidding activity will cease.

Another interesting aspect of this process is the seemingly redundant check of whether a submitted bid exceeds the current best bid. This check is made as a postrequisite of the `Submit Bid` step and as a prerequisite of the `Update Best Bid/Bidder` step. As a postrequisite on `Submit Bid`, it ensures that a bid that is known to not be the high bid when it is submitted is not even sent to the auctioneer. This prevents a bidder from maliciously keeping a bid open forever without ever raising the bid. Once a valid bid is submitted, the interval timer is

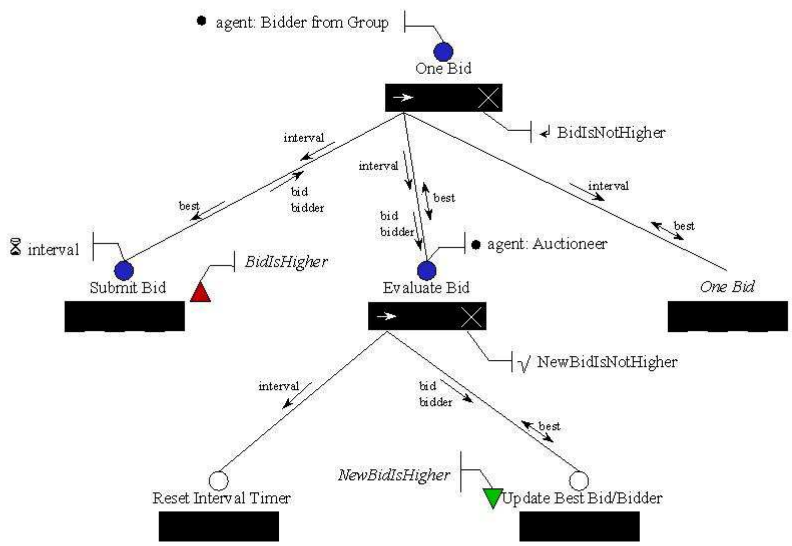

Figure 7: Bidding in Open-Cry Auction

immediately reset so that the auction does not close prematurely while a bid is being evaluated. Since the auctioneer is collecting bids from multiple sources, it is possible that by the time the auctioneer considers a bid, it is no longer the best bid. In that case, the prerequisite will fail and the best bid will not be updated.

Once again, if the offerer, bidders, and auctioneer are represented with separate syntactic entities, we need to examine each of them to determine if they jointly have the desired behavior. One desirable property is that each submitted bid should be evaluated by the auctioneer as it is received instead of deferring evaluation until the bidding ceases. It is straightforward to see that this property holds here, in contrast to the sealed bid process, since evaluation occurs immediately following the submission, while still within the bidding subprocess. It is the concise visual syntax of Little-JIL and the interleaving of the agent activities within a single graph that make this ordering apparent.

5. Evaluation

As was shown in the previous section, we can describe various types of negotiation processes using Little-JIL. With this approach, we are able to intuitively understand processes. Because Little-JIL is a visual language with a concise syntax, the processes in the previous section are easy to understand at an intuitive level once one becomes familiar with Little-JIL. In practice, we have discovered that people learn to read the syntax in a relatively short amount of time.

We are also investigating the use of flow analysis tools to support formal analysis of Little-JIL programs. Tools, such as Flavers[9], allow the specification of a property in terms of sequences of events. The property is compared to a flow graph representation of the program to determine if the sequence of events occurs on all, some, or no executions of the program. Flavers has been used successfully in the analysis of concurrent Ada and Java programs[10]. Since control flow and data flow are explicitly represented in Little-JIL, many of these analyses are straightforward and can be easily determined using informal analysis. The problem becomes more complex when we consider how exceptions can change the flow of execution and also in understanding the interleavings of children of parallel steps. For example, consider validating the property of the sealed bid auction process that a bid cannot be submitted after the interval timer has expired. This requires validating that no instance of `Submit Bid` completes after the timer expires, including instances active when the timer expires as well as those that are about to start due to recursive calls to `One Bid`, and also considering all parallel instantiations made by different bidding agents. As the amount of parallelism and exception handling increases, reasoning about properties will become increasingly difficult and the need for analysis tools will become more urgent.

As with AuctionBot and Kumar and Feldman's work, we have found it very useful to separate higher level concerns from domain-specific concerns. In the Little-JIL processes we've shown here, we have left undefined the domain specific concerns such as the criteria to be used to choose a vendor to perform a task, the criteria to be used to make a bidding decision, and how to decide when to accept or reject a proposal. From this higher level, we can compare the processes based on the interactions between the agents and the allowable sequences of events. From this starting point, we can continue this work to define some of these domain specifics either in Little-JIL or in the agents that the Little-JIL programs use. We have also found that a significant portion of the negotiation processes can be reused in multiple situations. We plan to continue to create and use reusable pieces to make development of new negotiation processes fast and easy.

6. Conclusions and Future Work

We are continuing to develop some aspects of Little-JIL. In particular, the timeout mechanism is still being refined. In addition, we are investigating the addition of a data flow mechanism to the language that would allow direct communication between concurrent steps. Currently such communication takes place through the use of shared memory. Shared memory is essentially an implicit communication mechanism which is more difficult to reason about than explicit communication through data flow.

We are continuing to expand the collection of auction processes defined in Little-JIL. We are interested in seeing how well the existing framework can be preserved across different auction types. Thus far, we have been intrigued by how little process modification has been required to represent quite different auction processes.

Ultimately our goal is to define and validate a collection of auction processes. Other researchers are doing an excellent job at defining architectures that allow composition of auction components. We are interested in building upon these ideas to construct auction processes that support reuse and composition and to do so within a notation that facilitates informal and formal reasoning about the correctness of the auction processes. As these processes become more complex and customization is carried out by non-experts, the ability to automate the reasoning with formal analysis tools will become essential. As the importance of auction processes increases in society, high assurance will become more critical to the well-being of our economy.

7. Acknowledgements

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, or the U.S. Government.

The definition of the Little-JIL language has been a team activity led by Sandy Wise and including Eric McCall, Stan Sutton, and Rodion Podorozhny as major contributors. We also thank Victor Lesser for his help in securing references to negotiation research in multi-agent systems.

8. References

1. Smith, R.G., *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.* IEEE Transactions on Computers, 1980. **C-29**(12): p. 1104-1113.

2. Sandholm, T. *An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations*. in *Eleventh National Conference on Artificial Intelligence*. 1993. Washington, DC.

3. Sandholm, T. and V. Lesser. *Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework.* in *First International Conference on Multi-Agent Systems (ICMAS-95)*. 1995. San Francisco.

4. Wurman, P.R., M.P. Wellman, and W.E. Walsh. *The Michigan Internet AuctionBot: A Configurable Auction Server for Human and Software Agents*. in *Second International Conference of Autonomous Agents*. 1998. Minneapolis, MN.

5. Kumar, M. and S.I. Feldman, *Business negotiations on the Internet*, . 1998, IBM Institute for Advanced Commerce.

6. Kumar, M. and S.I. Feldman, *Internet Auctions*, . 1998, IBM Institute for Advanced Commerce.

7. Wise, A., *Little-JIL 1.0 Language Report*, . 1998, Department of Computer Science, University of Massachusetts at Amherst, ftp://ftp.cs.umass.edu/pub/techrept/techreport/1998/UM-CS-1998-024.ps.

8. Jensen, D., *et al. Coordinating Agent Activities in Knowledge Discovery Processes*. in *International Joint Conference on Work Activities Coordination and Collaboration (WACC'99)*. 1999. San Francisco, CA.

9. Dwyer, M.B. and L.A. Clarke. *A Flexible Architecture for Building Data Flow Analyzers*. in *18th International Conference on Software Engineering*. 1996.

10. Naumovich, G., G.S. Avrunin, and L.A. Clarke. *Data Flow Analysis for Checking Properties of Concurrent Java Programs*. in *The 21st International Conference on Software Engineering*. 1999. Los Angeles.