TRUCE: Agent Coordination Through Concurrent Interpretation of Role-Based Protocols

Wilfred C. Jamison
Dep't of Electrical Engineering
and Computer Science
Syracuse University
Syracuse, NY 13244
wcjamiso@cat.syr.edu

Douglas Lea
Computer Science Department
SUNY Oswego
Oswego, NY 13126
dl@cs.oswego.edu

November 23, 1998

Abstract

Established protocols for coordination are essential for implementing joint-action activities among collaborating software agent. Most existing agents, however, are designed only to support static protocols, which limit their interaction domain to specific sets of agents. We develop an agent collaboration framework for open systems that enables an agent to expand its acquaintance set and to adapt to various coordination protocols dynamically. This is achieved through writing coordination scripts that are interpreted at collaboration time. We developed a role-based coordination language for writing these scripts, where the coordination mechanism used is the concurrent interpretation of a single script by the participants of a given collaboration. This interpretation defines the behavior of every agent. Thus. their individual interpretation may differ depending on the roles that were assigned to them. This new coordination language provides various coordination primitives in which the basic synchronization is achieved via distributed rendezvous points. In this paper, we present and demonstrate the elements of the TRUCE language.

1 Introduction

The current methodologies for designing interacting agents in a cooperative problem solving (CPS) environment are tight, in the sense that coordination protocols are often incorporated either within the agents themselves or the applications. The downside of this method is the high coupling between coordination and real computation, thus limiting the flexibility of the entire application. Moreover, such an application is more likely to require substantial rewriting when adapting to new collaborations. Thus, our main objective is to develop a collaboration framework that meets the following requirements (apart from the basic requirements of open distributed systems, such as security, reliability, response time, *etc.*):

Adaptive Coordination. Agents adapt to varying coordination protocols resulting from working with different groups of agents.

Heterogeneity. Agents work in an open system and therefore interact with different kinds of agents from different platforms.

Multiple Collaboration. Agents engage in several collaboration activities simultaneously, thus maximizing their use.

To achieve an adaptive and dynamic agent environment, we extract the coordination logic from the application at hand, and treat the corresponding coordination rules as subscribable services. In other words, we *separate* the coordination domain from the application domain where coordination is carried out by special agents called *coordinators*. As such, coordinators can offer higher-level coordination patterns that further generalize interaction behaviors. Coordination patterns are one way of attaining coordination logic reusability.

The separation approach is highly accepted in this research area. Gelernter and Carriero stated that using a separate coordination language leads to portability, which in turn promotes *reusability* [12]. This is because a coordination language is *not committed* to any base language. Consequently, the language can be used to link, or glue, different applications, written in different languages, together.

Coordination can be viewed as a set of coordination rules or *protocols*. Preprogrammed low-level fixed protocols such as TCP/IP do exist. Similarly, higher-level fixed protocols, such as blackboard systems, publish-subscribe systems, fault-tolerant services, and resource negotiation[18], are available. Dynamic coordination, however, implies *customizability* — that is, programmable coordination services that fit the needs of a particular group, and hence, requiring a special form of *protocol* programming. We do need to write coordination protocols, possibly more specialized and specific, that could best satisfy the needs of the problem being solved. Thus, we designed a general-purpose coordination language that will allow us to specify any arbitrary protocol.

TRUCE (Tasks and roles in a unified coordination environment) is a language framework based upon the environment that carries out coordination services. This environment is called ACACIA (Agency-based Collaboration Architecture for Cooperating Intelligent Agents). TRUCE, however, is not exclusive to ACACIA. In this paper, we shall be describing the TRUCE framework alone. The succeeding sections will present the essential concepts and is not intended for describing syntax and complete semantics. However, we provide an example to illustrate the use of this language.

2 Coordination Languages

There is a considerable number of coordination languages already introduced in this research area. These include languages based on *conversations, tuple-spaces, rules and events, logic systems* and others. Some examples are Linda [11], COOL [3], Finesse [4], CoLa [1] and PoLis [8]. TRUCE is a small and yet powerful scripting language that is designed to support various kinds of multi-party actions and protocols that are necessary for agent coordination in an open system. Its salient properties are listed below. Some of these cannot be found in existing languages.

- It simplifies delegation of coordination tasks to *coordinator agents*. These coordination tasks can be developed independently of the functional components of agents.
- It contains a variety of primitive coordination constructs.
- It enables simple expression of coordination constructs that often translate to complex underlying actions.
- It simplifies the task of programming in a heterogeneous environment.
- It provides dynamic capabilities that are useful for dynamic collaboration.

The most significant contribution of the TRUCE language is a coordination programming paradigm that is based on scripts (thus, protocols can be programmed dynamically). This paradigm introduces a new coordination mechanism that is based on concurrent interpretation of a single script.

3 Collaboration Group

Central to any collaboration framework, and to an open system in general, is the notion of a group (or collaboration group). A collaboration group is a set of possibly heterogeneous agents that are engaged together in a cooperative problem solving activity, and governed by a common set of coordination protocols. In our system, however, the members are replaced (that is, represented individually) by a homogeneous set of coordinators.

A collaboration group is defined by the composition of its members, the problem being solved, and the coordination protocols being used. The protocols do not define a single thread of control: control is distributed among the group members. This is necessary in an open system which operates asynchronously and in which agents may join and leave a group anytime.

Apart from the basic synchronization machinery, our coordination framework supports the following:

1. engagement in protocols that become known to agents dynamically in the course of their life-cycles;

2. separation of the identity of an agent from its *role(s)* in a protocol, allowing it to assume multiple roles in the same protocol and to engage simultaneously in multiple protocols;

3. accommodation for agents to join and leave collaboration groups and protocols dynamically;

4. encapsulation of coordination support as services whereby subscriptions are established when the support becomes necessary: details of coordination procedures are no longer the concern of the agents; and

5. availability of high-level supervisory capabilities without dependence upon fragile, centralized control mechanisms.

4 Coordination Protocols

Coordination among autonomous entities is easily implemented through the establishment of protocols. *Protocols* are rules or policies known to two or more parties who agree to comply with them. Protocols are created to address the set of dependencies imposed upon a collaborative task. Solutions to different problems mostly require different protocols. Some solutions, however, may use the same patterns of coordination for completely different problems. In such cases, coordination protocols can be shared and reused. We describe a familiar pattern below.

> A master agent sends the same message to a number of slave agents (mostly, by broadcasting). Each slave eventually sends its reply to the master. The master waits until it has collected all replies from the original set of slaves, at which point, it performs some processing on the messages received to come up with a single result.

One possible application that follows this protocol is *bidding* in which the master sends information about a sale item to the slaves. The latter reply with their price bids. After collecting all bids, the master chooses the winner (using some criteria). Another application is *redundant programming*. In this case, the master solves a problem using some methods. To evaluate the quality of its solution, it sends the same problem to a number of slaves that can solve the same problem, but using different methods. It compares its own solution with those returned by the slaves. Having collected them all, the master determines its confidence-level of its own solution.

Some protocols are simple constraints. For example, "Task B cannot proceed without finishing task A." is a constraint imposed upon task B. This example is an instance of *order dependencies*. Another example is "Task C gets its inputs only from task D.". By applying such a constraint, task C becomes *data dependent* on task D. Task D, on the other hand, remains unconstrained. Some constraints are state-based, that is, certain restrictions are imposed upon the possible states of an entity. For example, "Agent

A's buffer cannot exceed n items." is a constraint which A must observe. There are other types of dependencies. Interested readers can refer to Crowstron [10].

Some protocols may also come as guarded rules or triggers. These are rules or actions that are triggered whenever a particular condition is satisfied, for example, "When the temperature reaches 100 Farenheit, the boiler must be shut off.". The difference between a constraint and a trigger is that the former *imposes* a restriction, whereas the other *enables* a rule or an activity when certain conditions occur. Strictly speaking, a trigger is a form of constraint wherein the *application of rules* is constrained.

An agent may subscribe to a coordination service with different coordination protocols at various times. Since a collaboration group has no single locus of control (that is, each agent has its own thread), the only way that participants can follow an arbitrary protocol is by having a copy of a script that describes such protocol.

5 Concurrent Role-Based Interpretation

Concurrent Role-Based Interpretation constitutes TRUCE's primary mechanism for coordination. A script is interpreted by the collaboration group using multiple instances of the same interpreter. The collaboration group is given a script to guide its way to accomplishing its task. A copy of this script is given to each member of the group. Subsequently, they all interpret the script simultaneously.

Role-based interpretation is a form of *selective* interpretation wherein an interpreter does not have to execute every instruction in the script. The fraction of the script to be interpreted is determined from the interpreter's *view* of the script — which depends on its role assignment. We explain these concepts in the following sections.

5.1 Roles

In a system of interacting components, Lea and Marlow [16] used roles to describe a *view* of one or more components by providing an *interface* specification of the messages expected. Thus, a role abstracts all other things about the components. In the same way, a role in TRUCE hides various details of an agent who reveals only its capabilities by performing the steps (or instructions) specified for that role. Application agents are individually mapped to a particular role.

Collaboration is about tasks and their coordination. While the completion of these tasks is necessary, the identities of the providing agents are irrelevant to the process. Referencing roles instead of the actual agents, not only hides these information, but is sufficient to describe a collaboration. Such an abstraction enables different providers to play the same role at different times without affecting the overall coordination protocols. Agents may also subcontract their tasks to other agents (including those that are not members of the group). Such a strategy is hidden and unimportant to other roles, thus it can be adopted without introducing unwanted consequences.

Agents carry out their individual tasks based on the roles assigned to them. Once the *role assignments* (also called *role-binding* are established, all future references to the agents will occur through their role names. Knowledge about role assignment is hidden from other entities, including the users and other agents.

Roles may be viewed as a language mechanism for dynamically binding agents to a set of coordination protocols. A role is a profile in terms of expertise, skills and responsibilities. Any agent that fits the profile is a *prospect* for the role. Willing prospects become *candidates*. An agent may fit a number of profiles, suggesting that simultaneous role assignments are possible.

5.2 Role-Based Programming

Every instruction, called a *step*, in a script has a set of executors. These are roles delegated to carry out the instruction. There are four possible views when interpreting a step:

1. *executor* — that performs the action actively
2. *receiver* — that receives the target result of the action directly
3. *object* — that is being used or manipulated by the action
4. *observer* — that is not involved at all but can sense and use the results of the action

Unlike some other coordination languages, TRUCE is used to define an entire coordination plan in a single script (a script is called a *truce*). The actions of each collaborator in a collaboration group are all described in a *truce*. These actions or protocols are intermingled in a way that still reflects their execution order statically.

An action is specified using a *step*. A protocol specification is composed of one or more steps. Aside from the action itself, a step specifies these information: (1) the performer of the action, (2) the receiver of the action, and (3) the object of the action, and some optional specifiers or directives. Every role in a step has at least one of these roles.

- Performer. The role is the main executor of the step, which also means that it initiates the execution.
- Receiver. The role is the target of a message sent by the performer.
- Object. The role or some of its properties are referenced in the step.

A role that does not have any participation in a step but needs to wait until the step is completed is called a *spectator*: some tasks need to finish first before another can proceed. Primitive steps are the basic actions defined in the language. Every such step has an associated set of specifiers or directives. The 15 TRUCE primitive steps are listed in Table 1. User-defined steps can be formulated from existing steps through composition. A TRUCE step, optionally, can be given a user-defined name, or may

Primitive Steps	Description
truce	to declare and define a truce
cast	to declare a named set of roles
packet	to declare and define the format of a packet
protocol	to declare and define a set of event-based rules
set	to assign a value to a property
tell	to send information to another role
wait	to put a role in a waiting state
proceed	to tell a waiting role to resume its execution
retract	to undefine an existing property or rule
recover	to undo the retraction of a retracted name
trigger	to execute a truce, an episode or raise an event signal
step	to declare and define a named sequence of steps
if	to execute conditionally a sequence of steps
echo	to print out text on the standard output
break	to skip the current context

Table 1: TRUCE's Primitive Steps

declare names within the step body. The classification of names include *cast names, rule names, step names, packet names, property names* and *role names*.

A property is akin to a variable in traditional languages. A global property (prefixed by a dollar symbol) is one that can be accessed by all roles. Basic types includes *number, strings, boolean, role, date, time, list, packet*.

Every step has a *context* — a transient environment that consists of all names created within the step. A context also inherits the context of an immediately enclosing step, that is, when the step occurs inside another step. If collision occurs between a recently declared name and an inherited one (that is, same name), the former hides the latter until the current context is destroyed, which happens as soon as the interpretation of the step is completed.

Every role in a step defines a viewpoint, which implies that the semantics of a given step may differ depending on which role the step is being viewed from. For example, let the pseudocode below represent a step:

producer.tell consumer about object X.

In this example, tell is an action used to specify the transmission of a message from a source to a destination. The performer of the action is producer, while the receiver is consumer. The object is X. The semantics of tell from the point of view of the producer can be described as follows:

> The tell action specifies that a message is placed in a packet containing at least the address of the destination, and sent to the destination.

Whereas, the viewpoint of the *consumer* role is given below:

The tell action specifies that a message is expected to arrive, and therefore the receiver is suspended temporarily to wait for the message. When the packet finally arrives, the actual message is taken out of the packet and stored in a buffer, after which, the receiver resumes.

Thus, an interpreter for a *role-based* script behaves differently, depending on whose viewpoint it is performing the interpretation for.

5.3 Role Binding

TRUCE defines an arbitrarily large and unstructured *virtual agent space* with different agents coming from various places. Collaboration groups are formed from this agent space and coordinate themselves by interpreting a collaboration script. Hence, every agent becomes an independent interpreter. A role-based interpreter must assume one of the viewpoints (or role) in the step. It is therefore necessary to specify which agent has which role in every step. To do this, we maintain symbolic names or *id*s to represent every agent. We use these names to specify a collaborator's role in a step. The unique assignment of a symbolic name to an agent is called *role-binding*. Although we can perform role-binding before every step execution (thus allowing us to use different symbolic names every time), such method is not very efficient. Role-binding is done only once, – just before the first step is executed, – giving each collaborator only one symbolic name for the entire script. This symbolic name is used as the collaborator's *rolename*.

At any given step, a specific expertise/skill may be required for some of the roles in that step. The corresponding collaborators of those roles must therefore satisfy, directly or indirectly, the given requirement. In general, the requirements needed by a role, which a collaborator must be able to meet, is the sum of all expertise/skills required in all of the steps where that role appears. Role-binding is also the process of matching up the responsibilities of a role to an actual agent (a collaborator) that can and is willing to take these responsibilities. Furthermore, this binding serves as a *contract* between the collaborator and the collaboration group, obligating the former to fulfill all of its responsibilities. A collaborator may be bound to multiple roles only if it is able to meet the combined requirements of these roles. Roles and collaborators are bound dynamically, and therefore roles need not be bound to the same agent every time.

5.4 Concurrent Interpretation

We have seen the mechanics of role-based interpretation — how the semantics of a step is viewed by the interpreter based its role. However, role-based interpretation alone cannot accomplish the coordination process. In order to achieve the right coordination effect, a step must be interpreted from all of its viewpoints simultaneously, combining its semantics from all views. This process is called *concurrent interpretation*. By doing this, the coordination protocols embedded within a step are extracted and executed properly. Hence, concurrent interpretation is, in effect, the application of coordination protocols. We extend this idea by applying concurrent interpretation to all steps in a script. In so doing, the *coordination task* can be accomplished.

The combination of role-based and concurrent interpretation allows a coordination process to work without centralized control. It also avoids both performance and fault-tolerance problems associated with centralized mediation. Because each host has a copy of the complete script, the only communications necessary are those required by the steps. The key point is this: *Each agent knows exactly when to wait and to interact, and what to expect from others*. In a distributed environment, nothing can be assumed about the relative speeds of the interacting agents. Therefore, a *rendezvous* mechanism is used when two roles need to interact.

6 Example

In the area of *electronic commerce*, electronic *auctioning* is a very promising application of agents. We illustrate how a simple auction may be carried out and coordinated using TRUCE.

6.1 Description

The *fisherman's auction* protocol is an old method normally used by fishermen to sell their fish to the middlemen. A fisherman presents his catch to the public. When the go-signal is given, any interested middleman can *whisper* his bid to the fisherman or to a representative. A bidder can only whisper once. After the last bidder, the highest price and the winner are announced.

We extend the scenario, generalizing *fishermen* to *sellers*, where there can be multiple items for sale. In this setting, only one seller can sell an item at a time. A seller can also bid for other sellers' items.

6.2 Solution

We include a facilitator to serve as a neutral party. This agent manages the bids by collecting them and announcing the highest price. The bidding goes through several rounds until nobody else can beat the current highest price. The sellers take turns in a round-robin fashion. The whole session finishes when all sellers have sold (or at least attempted) to sell their items.

The solution is composed of two parts. First, we write a `truce` for a bidding session involving only one item from a seller. Second, we extend the solution to include repetition of the `truce` developed in part 1, changing only the seller and item parameters.

6.2.1 First Part.

Listing 1.0 gives us the complete solution codes for this part. We identify three roles in this coordination problem. These are facilitator, seller and some buyers. Also, we simplify the information domain, for brevity, by defining sale item as consisting of the following information: *item code, current price, description.* Some other information needed would include the *bid price* of each interested buyer and the winning bid. Thus, we declare three packets (lines 3-6). Packets are data structures that are used when passing messages.

Buyers are not allowed to communicate with each other. Communication only flows from the facilitator to the buyers and vice-versa. TRUCE enables us to define a communication topology for the group. Each type of topology has primitive communication operations. In this example, a *star* topology is ideal. This is specified in line 07 by using the *cast* step, which define a subgroup of the collaboration group. In a star topology, one of the roles become the center where communication can flow only between the center and all other roles. In this case, the facilitator is the center of the star.

There are 3 main activities that need to be accomplished.

- Declaration of item for sale
- bidding
- Declaration of the winning bid

6ft 180 lbs 54 eugene 40 yrs 9332319

For activity 1, we shall ask the seller to get the item information from the real collaborating agent (called its client). Then it relays the information to the facilitator (Line 32-34). Notice that a packet type must be specified explicitly (sell-item in this case) to determine the type of information being asked for. The symbol *_client* is reserved and pertains to the actual agent (thus, assigning values to it means these values are passed to the agent). Afterwards, the information is passed to the facilitator. The seller then waits until the facilitator notifies it with the outcome of the sale.

For activity 2, the item for sale is given to the buyers who in return would reply with their bids. The buyers take their bids from their corresponding collaborators. Later, the facilitator selects the highest bid and announces it to the bidders. It then calls for another round, giving them a chance to beat the current price. This procedure repeats until no bidder beats the current price.

In lines 10-16, the facilitator first relays the item information to all buyers using spread — a primitive operation for *star* in which the center broadcasts a message to the rest. This operation is followed by a block of TRUCE codes, called the callback actions. These are actions executed as soon as the receiver gets the message. In this example, a buyer executes the callback actions when it receives the item. The callback actions mainly describe the interaction between a buyer and the actual agent. The buyer gives the item information to the agent while the latter returns a bid. Necessarily, the facilitator collects all bids as shown in lines 18-26.

The property highest-price by the facilitator, which is initially set to zero, stores the current highest bid for the current round. The property winning-price stores the highest price that nobody else has beaten (hence, the winner). At this point, the facilitator collects the bids. *Collect* is another primitive operation in which the center receives messages from the other roles. The callback actions, which the facilitator executes whenever it receives a bid, tell us that the bid just received is compared with the current highest bid. The properties highest-price and winner (a role reference to the bidder of the highest price) are both updated depending on the result of the comparison. The predefined property *_sender* pertains to the role that sent the latest message (current bid, in this case). Notice that the specifier *synch* is set to true; This means that all members of the bidding-group waits until all bids have been collected and processed.

Now, we have to check whether another round is required. It happens when the current highest bid has changed. If so, a request for new bids from thebuyers is issued by the facilitator. Otherwise, the current highest price is declared as the winning price and the seller is notified.

Listing 1.0 Complete `truce` for Part 1

```
truce fisherman-auction { facilitator, seller, buyer[] } {
% Let's define the packets
   packet sell-item { number {item-code, current-price},
            text{descpn}},
         bid-item { number {item-code, asking-price} },
         winning-bid { number{price}, role{bidder} };
   cast bidding-group { facilitator, buyer} struct=star;
   episode auction {
      initially { facilitator.set {highest-price} 0;};
      step sell-and-bid {
         bidding-group.spread { sell-item{item-for-sale} } {
% when the buyers receive the item, they tell their agents
% about the item and then ask them for a bid price.
         set {_client} sell-item{item-for-sale};
         set {my-bid} bid-item{_client};
      }
% highest-price is initially set to zero;
      set { winning-price} {highest-price};
      bidding-group.collect { bid-item{my-bid} } synch=true
            timeout=5000 {
% For every replies received by the facilitator,
% it executes the following
         if { highest-price < my-bid@asking-price } {
            set {highest-price} my-bid@asking-price;
            set {winner} _sender;
         }
         if { winning-price < highest-price } {
```

```
            facilitator.set {winning-price} highest-price;
            sell-and-bid;
          }
        } % end of step
        seller.set {item-for-sale} sell-item {_client};
        seller.tell {facilitator} data=item-for-sale target=item-for-sale;
        seller.wait {facilitator};
        bidding-group.sell-and-bid;
% facilitator announces the winner
        facilitator.set {result} winning-bid{winning-price, winner};
        facilitator.tell {seller, buyer} data=result synch=true
              target=result;
        {seller, buyer}.set {_client} result;
      } % end-of-episode
} % end-of-truce
```

6.2.2 Second Part.

The second part of the solution considers the case where there are multiple sellers with multiple items to sell. We mentioned earlier that we shall use round-robin scheduling to coordinate them. We achieve this by structuring them into a ring - a different topology. A token is passed around so that the current bearer gets to sell one of its items. When a seller has nothing to sell, it simply passes the token to the next. This accomplishes the round-robin scheduling strategy.

Recall that we need to know when to stop the cycle. Once in a while, we should be able to test for the condition where all sellers did not attempt to sell. To accomplish this, we use the token as a counter. Initially set to 0, the counter is incremented everytime a seller sells an item. After travelling the whole ring, its value is tested by the facilitator. If the value remains at 0, the session is terminated. The whole listing is found in Listing 2.0. In line 3, we declare a cast with a ring topology.

To pass around the token, we use the **relay** communication protocol with the facilitator as the originator, as shown in lines 24-37. The token is set to 0 before the facilitator relays the token. In this topology, only one role receives the token at a time. The receiver can choose to sell or not (hence, the use of possibly). If not, the token is automatically passed to the next without modifying its value. Otherwise, it sets its flag myturn to indicate that it is going to sell something. A global property $selling is also set. The new seller waits while the selling is taking place. When it is over, the seller resets its flag and increments the token. The token is passed to the next seller in the ring.

The actual selling procedure is performed in a protocol named selling-protocol. Notice that all roles have to execute this protocol. The trigger involves a global property $selling, which is set by a seller in the sell step. See *selling-protocol* from lines 10-22. Only sellers test the value of their local property myturn. Recall that a receiver of the token sets this property. Thus, it gets to satisfy the condition and execute the step, **set** $“current-seller” = *_me*. The property *_me* is predefined and refers to the role executing the step.

When such setting has been accomplished, the protocol is temporarily disabled so that no further trigger can take place. The `truce` we have written in part 1 is performed. The role facilitator is bound to the same facilitator in this imported truce. The role seller is bound to whoever is the current seller while both the buyers and the sellers become the buyers. Once finished with the selling, the protocol is switched back on.

Listing 2.0 Complete `truce` for Part 2

```
truce multiple-fisherman-auction { facilitator, sellers[], buyers[]}
        import={fisherman-auction} {
    cast selling-group{ facilitator,sellers} struct=ring;
    packet token-type { number {counter} };
    episode sell-and-buy {
        initially  action
                facilitator.set {token} token-type{0};
        protocol selling-protocol {
            when { $selling = true } {
                sellers.if {myturn=true} {
                    set $“current-seller”=_me;
                }
                retract {selling-protocol};
                facilitator.set {$selling} false;
                fisherman-auction {facilitator, $“current-seller”,
                    {buyers, sellers} };
                recover {selling-protocol};
                facilitator.proceed{$“current-seller”};
            }
        } % end-of-protocol
    %%
        step sell {
            buyers.wait;
            facilitator.set {token} token-type{0};
            selling-group.relay {token} synch=true {
                sellers.{
                    possibly {
                        set {myturn} true;
                        set {$selling}true;
                        wait;
```

```
                    set {myturn} false;
                    set {token@counter} token@counter +1;
                 }
              }
          } % end-of-relay
          facilitator.proceed{buyers};
          {facilitator, buyers, sellers}.if {token@counter > 0 }
                 { sell;}
       } % end-of-sell
       sell;
   } %end-of-episode
} % end-of-truce
```

The step sell is the entry point of the episode. We have discussed how this step works. The control loops through the same step sell. Observe that the termination condition is checked against the token value.

7 Conclusion

Our work is mainly centered around the notion of a framework for developing collaborative agents in a distributed environment such as the *Internet*. Multi-agent systems are attractive for collaborative problem solving, but support tools and frameworks for development are not available. Our aim is to fill in the void by developing a collaboration framework in which coordination procedures are provided as a service for collaborating applications. The most important properties of our framework include the following:

- Decentralization of the whole coordination process
- Separation of coordination and computation concerns
- Services approach to coordination (advertise-subscribe collaboration model)
- Dynamic coordination using protocol scripts
- Reusable coordination protocols
- Internet-oriented
- Support for simultaneous participation in multiple collaborations by a single agent

We focused on providing coordination services and dynamic coordination via delegation and programmable coordination protocols. We designed a coordination language, named TRUCE, for writing coordination scripts. The TRUCE language is based upon the method of concurrent role-based interpretation. Finally, we expect for a significant increase in productivity of agent developers. This is primarily due to the abstraction of coordination protocols from the main computations of the problem being solved.

References

[1] Aguilar, M., Hirsbrunner, B., and Krone, O., "The CoLa Approach: A Coordination Language for Massively Parallel Systems", `http://wwww.cs.dartmouth.com` Institut d'Informatique de Fribourgm Chemin du Musee, Sept. 28, 1994.

[2] Arbab, F. "Coordination of Massively Concurrent Activities", *CS-R9565 1995* Computer Science/Department of Interactive Systems, Centrum voor Wiskunde en Informatica, Amsterdam, NL.

[3] Barbuceanu, M. and Fox, M., "The Design of a Coordination Language for Multi-Agent Systems", *Technical Report*, Enterprise Integration Laboratory, University of Toronto.

[4] Berry, A. and Kaplan, S.. "Open, Distributed Coordination with Finesse",*Technical Report*, School of Information Technology. The University of Queensland, Australia.

[5] Biddle, Bruce J., *Role Theory: Expectations, Identities, and Behaviors*, Academic Press, Inc. , NY, 1979.

[6] Bond, A. and Gasser, L.(editors) *Readings in Distributed Artificial Intelligence*, Morgan Kauffman Pub. Inc., San Mateo, California, 1988.

[7] Castellani, S., and Ciancarini, P., "Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets", *Technical Report*, Department of Computer Science, University of Bologna, Italy.

[8] Ciancarini, P., Jensen, K., and Yankelevich, D., "On the Operational Semantics of a Coordination Language", *Technical Report*, Department of Mathematics, University of Bologna, Bologna, Italy.

[9] Ciancarini, P., Vitali, F., and Tolksdorf, R., "Weaving the Web in a PageSpace Using Coordination", *Technical Report*, Department of Mathematics, University of Bologna, Bologna, Italy.

[10] Crowston, Kevin (`crowston@umich.edu`), "A Taxonomy of Organizational Dependencies and Coordination Mechanisms", *Technical Report*, The University of Michigan, School of Business Administration.

[11] Factor, M., Fertig, S. and Gelernter, D., "Using Linda to Build Parallel AI Applications", *TR-861*, Yale University Department of Computer Science, June 1991.

[12] Gelernter, D., Carriero, N., "Coordination Languages and their Significance", *Communications of the ACM*, February 1992/Vol. 35, No. 2, pp. 97-107.

[13] Haddadi, Afsaneh, "Communication and Cooperation in Agent System: A Pragmatic Theory", *Lecture Notes in Artificial Intelligence,* Vol. 1056, Springer-Verlag, 1995.

[14] Jamison, W., "ACACIA: An Agency Based Collaboration Framework for Heterogeneous Multiagent Systems", *Multiagent Systems Methodologies and Applications, Lecture Notes in Artificial Intelligents* 1286, Springer-Verlag, 1996, pp. 76-91.

[15] Jamison, W., "Approaching Interoperability for Heterogeneous Multiagent Systems Using High Order Agencies", *Cooperative Information Agents, First International Workshop, CIA'97, Kiel, Germany 1997 Proceedings,* Springer-Verlag, pp. 222-233.

[16] Lea, D. and Marlowe, J. "PSL: Protocols and Pragmatics for Open Systems", available at `http://gee.cs.oswego.edu/dl`,

[17] Malone, T. and Crowston, K., "The Interdisciplinary Study of Coordination", *ACM Computing Survey*, vol. 26. pp. 87-119, March 1994.

[18] Rosenschein, J. and Zotkin, G., *Rules of Encounter, Designing Conventions for Automated Negotiation among Computers*, The MIT Press,, 1994.

[19] Shoham, Y., "Agent-Oriented Programming",' *Artificial Intelligence* 60, pp 51-92, 1993.

[20] Singh, Munindar P., "Multiagent Systemts, A Theoretical Framework for Intentions, Know-How, and Communications", *Lecture Notes in Artificial Intelligence*, Vol. 799, Springer-Verlag, 1994.

[21] Wooldridge, Michael and Jennings, Nicholas R., "Agent Theories, Architectures and Languages: A Survey", in Wooldridge and Jennings (ed), *Intelligent Agent*, Springer-Verlag, 1-22.