Concurrent Role-Based Interpretation of Protocol Scripts: A Paradigm for Agent Coordination

Wilfred C. Jamison
Syracuse University
wcjamiso@cat.syr.edu

Douglas Lea
SUNY Oswego
dl@cs.oswego.edu

Abstract

This paper presents an approach to coordination based upon concurrent interpretation of scripts among agents adopting various roles in cooperative problem solving tasks. The paper also introduces an associated coordination language **TRuCE** that can be used for writing protocol scripts.

1 Introduction

Establishing dynamic protocols among domain-independent cooperative problem solving tasks by groups of fine-grained agents is a central coordination issue. Yet most current approaches to coordination encounter problems when applied in the kinds of open, heterogeneous environments (including, especially, the Internet) for which agent coordination is most needed.

ACACIA (Agency Collaboration Architecture for Cooperating Intelligent Agents) [9] is a coordination framework designed for use in open systems. Beyond the basic information sharing and synchronization support common to nearly all approaches to coordination, ACACIA includes a *collaboration group* framework providing:

- Support for engagement in protocols (for example, *auctions*) that become known to agents only dynamically, in the course of their lifecycles.
- Separation of the identity of an agent from its *role(s)* in a protocol, thus allowing agents to adopt multiple roles in the same protocol, and to simultaneously engage in multiple protocols.
- Accommodation for agents to dynamically join and leave collaboration groups and protocols. (This embodies fault-tolerance as well.)
- Encapsulation of coordination support as services.
- Availability of high-level supervisory capabilities without dependence upon fragile, centralized control mechanisms.

The ACACIA model and framework is simplest to describe in terms of the key elements of **TRuCE** (Task and Roles in a Unified Collaboration Environment), the ACACIA scripting language.

TRuCE operates in an environment in which coordination is itself a service that can be offered by one or more specialized agents or *coordinator agents*. Collaborators may therefore delegate their coordination concerns to these coordinators and thereby allow them to focus on their assigned task or even get involved with other collaboration groups. This is illustrated in Figure 1

This notion of collective agent interaction and collaboration forms a virtual agent community. This community is populated with coordinator agents who, together with the collaborators, are deployed around the network. Multiple collaboration groups may also be formed simultaneously. Management of services across groups may in turn be performed via *agencies* – high-level facilities that supervise coordinator activities and serve as initial points of contact. An instance of an agency exists in every host in a virtual community. An agency thus serves as the main locus of **TRuCE** run-time support.

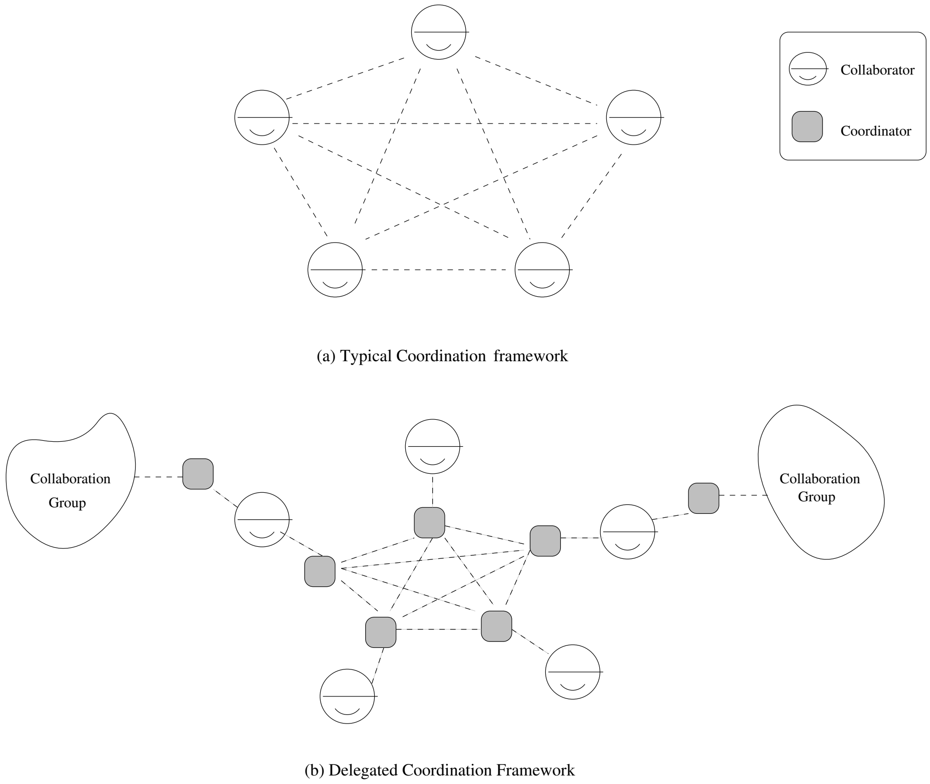

Figure 1: Delegating Coordination to Coordinating Agents

2 Coordination by Concurrent Role-Based Interpretation

A simple but effective approach for coordination is to build a coordination plan [10] or *protocol* prior to the group's actual encounter. A plan is viewed as a composition of simple coordination procedures and/or rules that are expressed using some notations or a language. **TRuCE**, adapts a coordination process that is guided by a protocol *script*. Thus, the interpretation of a protocol script defines the coordination process. **TRuCE** script interpretation may be defined in goal-directed terms. A script succeeds (is *satisfied* if all assertions about conditions and states in the script are true and all actions asserted have occurred.

2.1 Role-Based Interpretation

A single **TRuCE** script simultaneously defines all actions of all collaborators in a given protocol. Each interpretor deals only with those actions defined for the agent role that it is providing. Role-based interpretation is a form of selective interpretation in which an interpreter has a point of view of its script, based on its role. It executes only those instructions that are covered by such point of view.

For example, one frequently encountered high-level classification of roles is between a *participant* and an *observer*. A participant is directly involved or affected by an event. An observer is at most indirectly affected by the event. On one side we can group together all actions where a given role is a participant while maintaining their proper sequence. This grouping constitutes the role's point-of-view. Roles and agents are are related through *role-binding* in which an agent assumes a role and *promises* to fulfill it. A role is an abstraction of three facets: *capabilities, obligations, commitments*.

An agent that assumes a given role is referred to as *binding* the role with that agent. This act necessarily means that the agent qualified as having the capabilities, whether directly or indirectly (which is immaterial to us), required in the truce, and that it is willing to take the obligations of fulfilling the protocols therein. Roles and agents are loosely coupled that the former need not be bound to the same or similar agent every time. As long as it meets the requirements mentioned previously, any agent may fill up a given role. Furthermore, the commitment to accomplish its obligations completely comes with the binding as a contract between the initiating agent and the collaborator. A role, however, cannot be bound to more than one agent at a time per script. But it is possible for an agent to bind multiple roles simultaneously.

2.2 Concurrent Interpretation and Distributed Control

A collaboration group always involves two or more agents. With **TRuCE**, each obtains a copy of the script through the network. From there, individual agents are left to interpret the truce independently at their own pace. The underlying synchronization required for concurrent interpretation is managed by agencies, without the need for intervention by collaborators themselves.

The combination of role-based and concurrent interpretation allows a coordination process to work without centralized control. This avoids performance and fault-tolerance problems associated with centralized mediation. This scheme also minimizes network communication. Since each host has a copy of the complete script, the only communication necessary is that required by the protocol. Furthermore, the provision of "open expectations" (i.e., each agent already has knowledge of the other's expected behavior at every step), naturally builds the overall coordination machinery by way of meeting these expectations. This social behavior also brings about the needed responses to problems stemming from differences in CPU speed, network performance, I/O latencies, OS policies, etc, again to fulfill these expectations. Each agent knows when to wait and interact with the others and what to expect from them.

3 Protocol Specifications with TRuCE

We present here only some highlights of **TRuCE**, minimizing discussion of syntax and of details that are similar to those found in other coordination languages. Interested readers may find the complete online reference at [8].

Some general features of **TRuCE** include:

1. It is dynamic and extendible. The set of primitive steps is small, but can be extended by defining additional *specifiers*.
2. It supports both direct (callback) communications and shared spaces.
3. Existing scripts can be imported into new ones.
4. It is platform and agent-model independent.
5. It uses KQML (Knowledge Query Manipulation Language) as the medium of communication between coordinators and collaborators.
6. It combines rule-based, event-based and procedural approaches.
7. Its delegation model enables collaborators to participate in multiple collaboration groups simultaneously, while also ensuring compliance of established protocols.

3.1 The Virtual Machine

The underlying virtual machine of the language is shown in Figure 2. Each coordinator sits on its own virtual machine who owns two data spaces, its knowledge base where it stores definitions, rules, conditions, *etc*, and also one for its *properties*. Properties are declared attributes that can be created dynamically , with values of text, numerical, or boolean type. A third data space is (conceptually, if not physically) shared by all coordinators, and maintains global properties. Local and global properties are treated in the same way and differentiated only by a dollar sign ("$") prepended to global properties. Properties can be created with the *Set* step.

The whole virtual machine is then a network of *coordinator's virtual machines*. As can be seen, the coordinator has direct communication lines with its collaborator. The special role name, *Client*, is reserved to refer to its collaborator. Internally, the two use KQML [6] to communicate. Every pair of coordinators also has a dedicated communication line which may be disabled and enabled in the course of the coordination process.

3.2 The Protocol Steps

A **TRuCE** plan is written as a protocol script, or `truce`. Each `truce` is a collection of primitive protocol `steps`. A step typically has the following information:

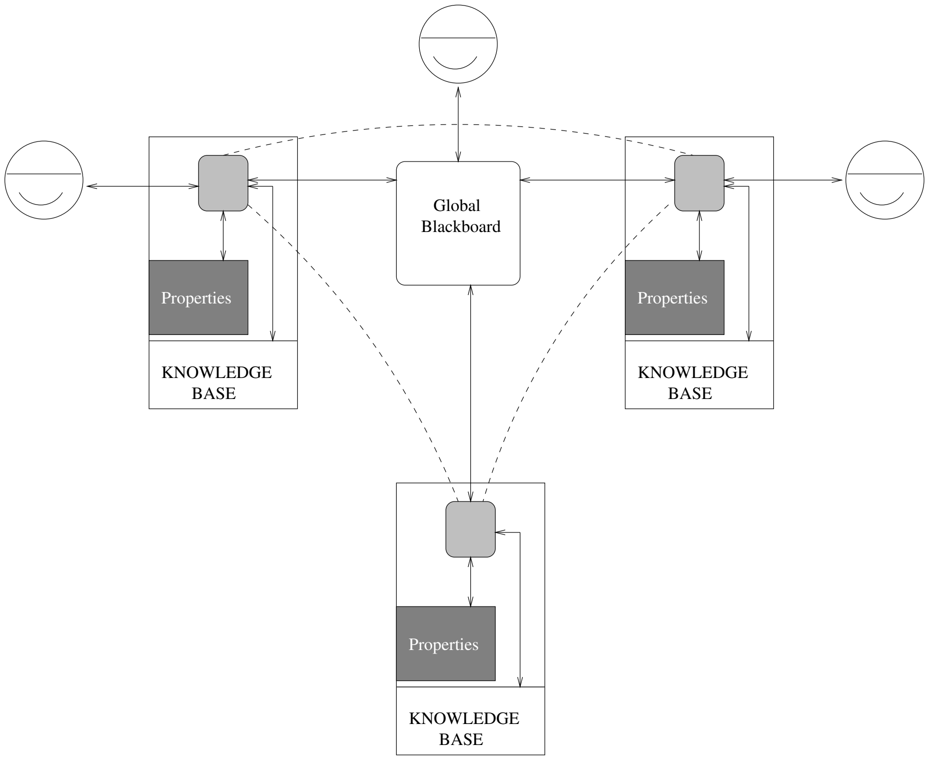

Figure 2: TRuCE virtual machine

- Active role - is the role that is assigned to initiate the action. There is at least one active role in every action.
- Passive role - is another role involved to which the action is directed, i.e., the receiver of the action.
- Object - the object of the action is that which the action manipulates.
- Directives - are optional information or actions to give more specifications about the action.

An example of a step is given.

```
producer.tell { consumer }  data=msg  ack=true ;
```

This step illustrates the *tell* protocol in which information is passed from the active role to the passive role. The role *producer* initiates the action to the role *consumer* with the data stored in a buffer named *msg*, the object of the action. The directive *ack* instructs *consumer* to send back an acknowledgment once data is received.

The general format of a step is:

```
[executor(s)].protocol-name (id | { a1, a2, ..., an} |
                            id { a1, a2, ..., an})
           [specifier=<value>]* { b1;  b2; ..., bm};
```

The executors of a step may be specified by prepending the protocol-name with a role or a cast and a dot. A list of roles may be given as well, e.g. *{ recruiter, solicitor, examiner}.protocol-name* . If not specified, all roles in the truce become the executors. The *protocol-name* may be one of **TRuCE**'s primitive protocol steps. They are listed down and illustrated in Table 1 and Table 2. Declarative steps act only to formulate knowledge or facts that will be used later in the coordination plan.

Each individual protocol step has a set of specifiers. These are directives to the interpreter to give more information about the execution of the step. For example, all steps have the *exception* specifier whose value is also a step or set of steps . This specifier tells the interpreter to execute these these steps whenever the original step causes an exception. To learn more about other specifiers, the reader is referred to the language's reference manual.

Protocol Steps	**Description**
truce	declares and defines a new truce e.g. truce my-truce $\{role_1, role_2, roles[]\}$ import={contract-net} { body-of-truce };
cast	groups a set of roles together so that they can be referenced as one entity e.g. cast educators {teacher,tutor,professor,author};
step	defines a new step by giving a name to a sequence of steps e.g. step verify $\{step_1; step_2; step_3; ...; step_n\}$;
packet	defines a data protocol for passing information format e.g. packet datapacket{text{name,dependencies }, number { iterationCount } };
protocol	defines a set of conditional protocols
episode	defines a new episode in the current truce

Table 1: Declarative Protocol Steps

Ciancarini [4] introduced three concepts that coordination models define. These are *coordinating entities, coordination medium* and *coordination laws*. These provide a convenient basis for discussing other properties of **TRuCE**.

Protocol Steps	Description
set	binds a property or data item to a specific value e.g. gatekeeper.set{cycle-cnt, initiator}{3, agent-name};
tell	sends information/data to a target role e.g. james-bond.tell {dr-no} {"I am coming."};
wait	suspends a role to wait for a signal e.g. consumer.wait {producer};
proceed	sends a signal to waiting role(s) e.g. producer.proceed { consumer };
retract	undefines an existing name where a name can be a name of a cast, step, property or protocol e.g. recruiter.retract { educators };
trigger	signals a truce or an episode to start e.g. trigger { recruitment};
break	terminates the execution of the truce, step or episode whichever the role is currently in e.g. recruiter.break;

Table 2: Imperative Protocol Steps

3.3 Coordination Entities

There is only one coordinating entity in **TRuCE**, the *Role*. Each step in a truce involves at least one role. (For this reason, *role* and *executor* are used synonymously in **TRuCE**.) Roles, such as *broker* and *inspector*, are uniquely defined. When a set of roles behave exactly in the same way, for example multiple bidders in a bidding protocol, they can be collectively grouped as a *squad* and assume the same base name. Hence, a group of *bidder* roles is declared as *bidder[n]* where n is an optional specification of the exact count. Otherwise, the squad's size is unknown and assumes no boundaries. The notation, *bidder[i]* where *i* is either a positive number or a variable that assumes such number, is a unique role referring to the *ith* member of the squad.

Different roles may also be grouped together and referenced as one entity. For example, { *recruiter, solicitor, examiner* } is a team or what is called a *cast*. Casts may also be referred to with a name. They can be given a name using the *cast* protocol.

Truces are written first with a header as shown below.

```
truce fisherman-auction { fisherman, middleman, bidders[] }
import={contract-net} { <body-of-the-truce> };
```

A *fisherman-auction* coordination plan is declared with three main roles involved, a *fisherman*, a *middleman* and an unlimited squad of *bidders.* The declaration also says that a truce named *contract-net* is known to exist and that it is going to be reused in the current protocol, as indicated by the *import* specifier. Finally, the body is any sequence of steps.

3.4 Coordinating Medium

There are three explicit communication media: *communication lines and mailboxes*, *packet messages*, and *shared dataspaces*. Every pair of coordinators has direct communication lines that can be used with the *tell* step. The role name of the target receiver serves as the identifier for the line. The *tell* step has a few specifiers that include *timeouts*, *blocking or non-blocking*, *reply or acknowledgment needed*.Meanwhile, each coordinator has a mailbox where messages can be buffered.

Packet messages are the medium of exchanging structured information among collaborators. Packet formats can be declared using the *packet* step. They are used along with the *tell* step when specifying the format of the data being told.

```
packet bid { text { item-code }, number {price} };
bidders.tell {middle-man} message=data packet=bid;
```

The example above shows that each role in the *bidders* squad are expected to send information, which is stored in its property *data*, to the *middle-man* role. On the other hand, the latter expects that each information received is formatted in a manner similar to the packet *bid*. Moreover, since the action is to be initiated by a squad, the receiver role is expected to wait for all messages to come from all members of the squad.

The shared data space acts as a blackboard [7]. Here, global properties are stored and accessed. Global properties are expressed as associative attributes. For example,

```
recruiter.set {$''limit''} {currentLimit};
supervisor.set {mylimit} {``limit''};
supervisor.set {currentLimit} {$mylimit};
```

Here, role *recruiter* creates a new global property with the name *limit* by assigning to it a value that comes from a local property, *currentLimit*, The next two steps allow *supervisor* to access the same global property. A NIL value is returned when such global property does not exist.

3.5 Coordinating laws

The main law or mechanism in **TRuCE** is that of *concurrent role-based interpretation* as described in the previous sections. Rendezvous points are established by protocol steps such as *tell, wait* and *proceed*. A *wait* step may specify who to wait for, e.g. *recruiter.wait* {supervisor, examiner}. The role *recruiter* waits until it receives a *proceed* signal from both *supervisor* and *examiner*. This is also true of the *proceed* protocol that specifies the target of the signal. If not specified, *wait* will wait for any one role while *proceed* will send a signal to all waiting roles.

Conditional protocols are declared and taken into effect by the *protocol* step. A protocol condition is a pair of condition and action law such that whenever the condition is satisfied, the corresponding action will have to be satisfied too by executing it. This is a rule and event based mechanism in which conditions subscribe to events that affect them so that their occurrences will trigger a check on the conditions. The format of a conditional protocol is:

```
protocol <protocol-name> {
   (rule;)+
}

where rule ::= when {<conditional-expr>} necessarily {
                        [ cond { <conditional-expr> }, ]
                          action { <list-of-steps> } }
                    }
           ::=  never {conditional-expr}
           ::=  always {conditional-expr}
```

Upon execution of the protocol step, these rules are taken into effect. Any violations will result to an asynchronous exception. In addition, the use of this step together with the *retract* step creates a dynamic coordination mechanism in which protocols may be enabled and disabled depending on the circumstances.

Episodes. A truce may be composed of one or more episodes. Episodes are grouping constructs that help organize large scripts. An Episode is a collection of protocol steps, possibly with pre and postconditions. Episodes execute sequentially – one cannot take place unless its immediate predecessor have taken place and finished successfully. Before an episode can be started, all roles must rendezvous (thus, wait) at that point. Similarly, they will have to leave the episode together. Thus, this is another synchronization construct at the activity level.

The format of an *episode* step is:

```
episode <episode-name> {
            [initially {  cond { <conditional-expr> } |
                          action { <list-of-steps> } };]
            [eventually { cond { <conditional-expr> } |
                          action { <list-of-steps> } };]

           <list-of-steps>
}
```

The optional *initially* and *eventually* clauses are specifications for preconditions and postconditions respectively. The former must be satisfied upon entry to the episode while the latter immediately before leaving the episode. When actions are specified, they are to be be executed. Without satisfying them, a role cannot enter or leave the episode. Episodes are executed using the *trigger* step.

Topologies. Truces may define topologies to simplify descriptions of communication patterns within steps and episodes. For example:

```
cast mycast { executor, recruiter, examiner } struct = pipe;
```

The step introduces a new cast and uses the specifier *struct* to define its topology, in this case is a *pipe*. This results to a pipe communication pattern among the three roles, i.e., *executor* and *recruiter* can communicate, so are *recruiter* and *examinaer* but not *executor* and *examiner*. Other supported topologies are *ring*, *tree*, *star* and *simple mesh*. Each communication topology comes with a set of communication primitives such as *spread* and *collect*. A simple user-defined topology may also be specified.

4 Conclusion

We have presented our approach to solving the *coordination problem* in a collaborative work by a group of agents. The ACACIA framework approaches coordination of collaborating agents via coordination plans, or *truce* written in **TRuCE**. Concurrent interpretation of the same truce by the coordinating agents is the principal mechanism for the whole coordination process. More importantly, each coordinator has a unique *point of view* of the truce which is based on its role in the truce. Thus, a role-based interpretation of the script is necessary to achieve the desired effect of the coordination plan. An implementation of the language is currently under development.

References

[1] Baral, C., Gelfond, M. and Provetti, A., "Representing Actions: Laws, Observations and Hypotheses", The Journal of Logic Programming, Elsevier Science Inc., 1997, pp. 201-243.

[2] Cammarata, Stephanie, McArthur, David and Steeb, Randall, "Strategies of Cooperation in Distributed Problem Solving", *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser (editor), Morgan Kauffman Pub. Inc., San Mateo, California, 1988.

[3] Cardozo, E., Sichman, J.S., Demazeau, Y., "Using Active Object Model to Implement Multi-Agent Systems", *Proc. of the 5th IEEE Int. Conf. on Tools with Artificial Intelligence*, 1993, Boston, USA.

[4] Ciancarini, Paolo and Gaspari, Mauro, "Rule-based Coordination of parallel logic programs", Department of Mathematics, University of Bologna, Bologna, Italy.

[5] Durfee, Edmund H., Lesser, Victor R. and Corkill, Daniel D., "Cooperation Through Communication in Distributed Problem Solving Network", *Distributed Artificial Intelligence*, Michael N. Huhns (editor), Pitman Publishing, London. 1987.

[6] Finin, Tim, Fritzon, Rich and McKay, Don, "A Language and Protocol to Support Intelligent Agent Interoperability", *Proceedings of the CE and CALS Washington '92 Conference*, June 1992.

[7] Hayes-Roth, Barbara, "A Blackboard Architecture for Control", *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser (editor), Morgan Kauffman Pub. Inc., San Mateo, California, 1988.

[8] Jamison, Wilfred and Lea, Douglas, "TRuCE: A Role-Based Coordination Language for Collaborating Agents", http://www.cat.syr.edu/ wcjamiso/truce, 1998.

[9] Jamison, Wilfred, "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.

[10] Katz, Matthew J. and Rosenschein, Jeffrey S., "Plans for Multiple Agents",*Distributed Artificial Intelligence Vol. II*, Les Gasser and Michael N. Huhns (editor), Pitman Publishing, London. 1989.

[11] Rosenschein, Jeffrey and Zotkin, Gilad, **Rules of Encounter, Designing Conventions for Automated Negotiation among Computers**, The MIT Press, Cambridge, Mass., 1994.

[12] Smith, Reid G. and Davis, Randall, "Frameworks for Cooperation in Distributed Problem Solving", *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser (editor), Morgan Kauffman Pub. Inc., San Mateo, California, 1988.

[13] Wooldridge, Michael and Jennings, Nicholas R., "Agent Theories, Architectures and Languages: A Survey", in Wooldridge and Jennings Eds., **Intelligent Agent**, Berlin:Springer-Verlag, 1-22.