Modeling Agent Conversations with Colored Petri Nets

R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, Yun Peng
Laboratory for Advanced Information Technology
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland
cost@acm.org, yechen,finin,jklabrou,ypeng@cs.umbc.edu

Abstract

Conversations are a useful means of structuring communicative interactions among agents. The value of a conversation-based approach is largely determined by the conversational model it uses. Finite State Machines, used heavily to date for this purpose, are not sufficient for complex agent interactions requiring a notion of concurrency. We propose the use of Colored Petri Nets as a model underlying a language for conversation specification. This carries the relative simplicity and graphical representation of the former approach, along with greater expressive power and support for concurrency. The construction of such a language, Protolingua, is currently being investigated within the framework of the Jackal agent development environment. In this paper, we explore the use of Colored Petri Nets in modeling agent communicative interaction.

1 Introduction

Conversations are a useful means of structuring communicative interactions among agents, by organizing messages into relevant contexts and providing a common guide to all parties. The value of a conversation-based approach is largely determined by the conversational model it uses. The presence of an underlying formal model supports the use of structured design techniques and formal analysis, facilitating development, composition and reuse. Most conversation modeling projects to date have used or extended finite state machines (FSM) in various ways, and for good reason. FSMs are simple, depict the flow of action/communication in an intuitive way, and are sufficient for many sequential interactions. However, they are not adequately expressive to model more complex interactions, especially those with some degree of concurrency. Colored Petri Nets (CPN) [12, 13, 14] are a well known and established model of concurrency, and can support the expression of a greater range of interaction. In addition, CPNs, like FSMs, have an intuitive graphical representation, are relatively simple to implement, and are accompanied by a variety of techniques and tools for formal analysis and design.

To appear in *Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents '99*, Seattle, Washington, May 1999.

We have explored the use of model-based conversation specification in the context of multi agent systems (MAS) supporting manufacturing integration [20]. Agents in our systems are constructed using the Jackal agent development platform [6], and communicate using the KQML agent communication language [8]. Jackal, primarily a tool for communication, supports conversation-based message management through the use of abstract conversation specifications, which are interpreted relative to some appropriate model. Conversation specifications, or protocols, can describe anything from simple message/acknowledgment interactions to complex negotiations.

In the next section, we present a motivation for using conversations to model and organize agent interaction. Next, we present CPNs, the model we propose to use, in more detail. Following this, we discuss the implementation of these ideas in a real MAS framework. Finally, we present two examples of CPN use: the first, specification of a simple KQML register conversation, and the next, a simple negotiation interaction.

2 Conversation-Based Interaction Protocols

The study of agent communication languages (ACLs) is one of the pillars of current agent research. KQML and the FIPA ACL are the leading candidates as standards for specifying the encoding and transfer of messages among agents. While KQML is good for message-passing among agents, directly exploiting it in building a system of cooperating agents leaves much to be desired. After all, when an agent sends a message, it has expectations about how the recipient will respond to the message. Those expectations are not encoded in the message itself; a higher-level structure must be used to encode them. The need for such conversation policies is increasingly recognized by the KQML community, and has been formally recognized in the latest FIPA draft standard [9, 7].

It is common in KQML-based systems to provide a message handler that examines the message performative to determine what action to take in response to the message. Such a method for handling incoming messages is adequate for very simple agents, but breaks down as the range of interactions in which an agent might participate increases. Missing from the traditional message-level processing is a notion of message context.

A notion growing in popularity is that the unit of communication between agents should be the conversation. A conversation is a pattern of message exchange that two (or more) agents agree to follow in communicating with one

another. In effect, a conversation is a communications protocol, albeit one that may be initiated through negotiation, and may be short-lived relative to the way we are accustomed to thinking about protocols. A conversation lends context to the sending and receipt of messages, facilitating interpretation that is more meaningful. The adoption of conversation-based communication carries with it numerous advantages to the developer. There is a better fit with intuitive models of how agents will interact than is found in message-based communication. There is also a closer match to the way that network research approaches protocols, which allows both theoretical and practical results from that field to be applied to agent systems. Also, conversation structure can be separated from the actions to be taken by an agent engaged in the conversation, facilitating the reuse of conversations in multiple contexts.

To date, relatively little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Strides must be taken in the toward facilitating the construction and reuse of conversations. An ontology of conversations and conversation libraries would advance this goal, as would solutions to the following questions:

1. Conversation specification: How can conversations best be described so that they are accessible both to people and to machines?
2. Conversation sharing: How can an agent use a conversation specification standard to describe the conversations in which it is willing to engage, and to learn what conversations are supported by other agents?
3. Conversation aggregation: How can sets of conversations be used as agent 'APIs' to describe classes of capabilities that define a particular service?

2.1 Conversation Specification

A specification of a conversation that could be shared among agents must contain several kinds of information about the conversation and about the agents that will use it. First, the sequence of messages must be specified. Traditionally, deterministic finite-state automata (DFAs) have been used for this purpose; DFAs can express a variety of behaviors while remaining conceptually simple. For more sophisticated interactions, however, it is desirable to use a formalism with more support for concurrency and verification. This is the motivation behind our investigation of CPNs as an alternative mechanism. Next, the set of roles that agents engaging in a conversation may play must be enumerated. Many conversations will be dialogues, and will specify just two roles; however conversations with more than two roles are equally important, representing the coordination of communication among several agents in pursuit of a single common goal. For some conversations, the set of participants may change during the course of the interaction.

DFAs and roles dictate the syntax of a conversation, but say nothing about the conversation's semantics. The ability of an agent to read a description of a conversation, then engage in such a conversation, demands that the description specify the conversation's semantics. To be useful though, such a specification must not rely on a full-blown, highly expressive knowledge representation language. We believe that a simple ontology of common goals and actions, together with a way to relate entries in the ontology to the roles, states, and transitions of the conversation specification, will be adequate for most purposes. This approach sacrifices expressiveness for simplicity and ease of implementation. It is nonetheless perfectly compatible with attempts to relate conversation policy to the semantics of underlying performatives, as proposed for example by [3].

These capabilities will allow the easy specification of individual conversations. To develop systems of conversations though, developers must have the ability to extend existing conversations through specialization and composition. Specialization is the ability to create new versions of a conversation that are more detailed than the original version; it is akin to the idea of subclassing in an object-oriented language. Composition is the ability to combine two conversations into a new, compound conversation. Development of these two capabilities will entail the creation of syntax for expressing a new conversation in terms of existing conversations, and for linking the appropriate pieces of the component conversations. It will also demand solution of a variety of technical problems, such as naming conflicts, and the merger of semantic descriptions of the conversations.

2.2 Conversation Sharing

A standardized conversation language, as proposed above, dictates how conversations will be represented; however, it does not say how such representations are shared among agents. While the details of how conversation sharing is accomplished are more mundane than those of conversation representation, they are nevertheless crucial to the viability of dynamic conversation-based systems. Three questions present themselves:

- How can an agent map from the name of a conversation to the specification of that conversation?
- How can one agent communicate to another the identity of the conversation it is using?
- How can an agent determine what conversations are handled by a service provider that does not yet know of the agent's interest?

2.3 Conversations Sets as APIs

The set of conversations in which an agent will participate defines an interface to that agent. Thus, standardized sets of conversations can serve as abstract agent interfaces (AAIs), in much the same way that standardized sets of function calls or method invocations serve as APIs in the traditional approach to system-building. That is, an interface to a particular class of service can be specified by identifying a collection of one or more conversations in which the provider of such a service agrees to participate. Any agent that wishes to provide this class of service need only implement the appropriate set of conversations. To be practical, a naming scheme will need to be developed for referring to such sets of conversations, and one or more agents will be needed to track the development and dissolution of particular AAIs. In addition to a mechanism for establishing and maintaining AAIs, standard roles and ontologies, applicable to a variety of applications, will need to be created.

There has been little work on communication languages from a practitioner's point of view. If we set aside work on network transport protocols or protocols in distributed computing (e.g., CORBA) as being too low-level for the purposes of intelligent agents, the remainder of the relevant research may be divided into two categories. The first deals with theoretical constructs and formalisms that address the

issue of agency in general and communication in particular, as a dimension of agent behavior (e.g., AOP [22]). The second addresses agent languages and associated communication languages that have evolved somewhat to applications (e.g., TELESCRIPT [23]). In both cases, the bulk of the work on communication languages has been part of a broader project that commits to specific architectures.

Agent communication languages like KQML provide a much richer set of interaction primitives (e.g., KQML's performatives), support a richer set of communication protocols (e.g., point-to-point, brokering, recommending, broadcasting, multicasting, etc.), work with richer content languages (e.g., KIF), and are more readily extensible than any of the systems described above. However, as discussed above, KQML lacks organization at the conversation level that lends context to the messages it expresses and transmits. Limited work has been done on implementing and expressing conversations for software agents. As early as 1986, Winograd and Flores [24] used state transition diagrams to describe conversations. The COOL system [2] has perhaps the most detailed current finite state automata model to describe agent conversations. Each arc in a COOL state transition diagram represents a message transmission, a message receipt, or both. One consequence of this policy is that two different agents must use different automata to engage in the same conversation. COOL also uses an :intent slot to allow the recipient to decide which conversation structure to use in understanding the message. This is a simple way to express the semantics of the conversation, though it is not sufficient for sophisticated reasoning about and sharing of conversations.

Other conversation models that have been developed include those of Parunak [19], Chauhan [4], who uses COOL as the basis for her multi-agent development system, Kuwabara et al. [15], who add inheritance to conversations, Nodine and Unruh [18], who use conversation specifications to enforce correct conversational behavior by agents, Bradshaw [3], who introduces the notion of a conversation suite as a collection of commonly-used conversations known by many agents, and Labrou [16], who uses definite clause grammars to specify conversations. While each of these makes contributions to our general understanding of conversations, none show how descriptions of conversations might be shared by agents and used directly by them in implementing conversations.

2.4 Defining Common Agent Services via Conversations

A significant impediment to the development of agent systems is the lack of basic standard agent services that can be easily built on top of the conversation architecture. Examples of such services are: name and address resolution; authentication and security services; brokerage services; registration and group formation; message tracking and logging; communication and interaction; visualization; proxy services; auction services; workflow management; coordination services; and performance monitoring. Services such as these have typically been implemented as needed in individual agent development environments. Two such examples are an agent name server and an intelligent broker.

2.4.1 Agent Name Server

At first blush, the problem of mapping from an agent name to information about that agent (such as its address) seems trivial. However, solving this problem in a way that can easily scale as the number of users and amount of data to be processed grows is difficult. We believe that development of a successful symbolic agent addressing mechanism demands at least two advances:

1. A simple naming convention to place each role an agent might play in an organization at a unique point in a namespace for that organization. Currently there is no widely-accepted mechanism for universal unique agent naming (in the way that there now is, e.g., for Internet hosts or web documents).

2. An efficient, scalable name service protocol for mapping from symbolic role names to information about the agents that fill those roles.

To a large extent, the desired techniques can be modeled after existing name service techniques such as DNS (which is widely implemented) and CORBA (whose namespace mechanisms are only narrowly implemented). Such techniques are well-studied, highly reliable, and scalable. Agent name service will differ from DNS primarily in that agents will tend to appear, disappear, and move around more frequently than do Internet hosts. This will necessitate the development of naming conventions that are less rigid than those used in DNS, and algorithms for mapping from names to agent information that do not rely on the static local databases found in DNS.

2.4.2 Intelligent Broker

A system that is to respond to the demands of multiple users, with needs that vary over time, under an ever-increasing query load must be able to do on-the-fly matching of queries to documents and services. In an agent-based architecture, this means that one agent must be able to dynamically discover other agents based on the content of their knowledge. It should exploit the research on conversations and the symbolic agent-addressing scheme described above, while at the same time fitting neatly into existing brokered systems. Such systems will continue to see a single broker where there had been a single broker all along; now, however, that broker will have the option of coordinating many other disparate brokers of varying capabilities.

3 Colored Petri Nets

Petri Nets (PN), or Place Transition Nets, are a well known formalism for modeling concurrency. A PN is a directed, connected, bipartite graph in which each node is either a *place* or a *transition*. *Tokens* occupy places. When there is at least one token in every place connected to a transition, we say that transition is *enabled*. Any enabled transition may *fire*, removing one token from every input place, and depositing one token in each output place. Petri nets have been used extensively in the analysis of networks and concurrent systems. For a more complete introduction, see [1].

Colored Petri Nets (CPN) differ from PNs in one significant respect; tokens are not simply blank markers, but have data associated with them. A token's *color* is a schema, or type specification. Places are then sets of tuples, called *multi-sets*. *Arcs* specify the schema they carry, and can also specify basic boolean conditions. Specifically, arcs exiting and entering a place may have an associated function which determines what multi-set elements are to be removed or deposited. Simple boolean expressions, called *guards*, are associated with the transitions, and enforce some constraints on tuple elements. This notation is demonstrated in examples

below. CPNs are formally equivalent to traditional PNs; however, the richer notation makes it possible to model interactions in CPNs where it would be impractical to do so with PNs.

CPNs have great value for conversational modeling, in that:

- They are a relatively simple formal model.
- They have a graphical representation.
- They support concurrency, which is necessary for many non-trivial interactions.
- They are well researched and understood, and have been applied to many real-world applications.
- Many tools and techniques exist for the design and analysis of CPN-based systems.

High-level petri nets are a very natural formalism for modeling concurrent systems, and the notion of applying them to MASs is not new [10]. In some related work, Holvoet and Kielmann [11] have applied the CPN formalism to specifying agent systems based on their Objective-Linda coordination model. Moldt and Wienberg [17] (using their variant, Object-Oriented Nets) and Purvis and Cranefield [21] have also demonstrated the use of high-level petri nets in modeling systems of agents.

4 Putting Colored Petri Nets to Work

Currently, we are investigating the value of CPNs in a general framework for agent interaction specification. Within this scheme, agents use a common language, Protolingua, for manipulating CPN-based conversations. Protolingua itself is very sparse, and relies on the use of a basic interface definition language (IDL) for the association of well known functions and data types with a CPN framework. Agents use Protolingua interpreters to execute various protocols. Protolingua itself is simple in order to facilitate the porting of interpreters to many different platforms.

One advantage to this approach is that a variety of interpreter implementations may be used, and the agent may trade resources for conversational 'power'. A very simple CPN interpreter may be able to efficiently execute very small or simple protocols; an agent may chose to use this in most interactions, while employing more expensive and powerful interpreters for more complex negotiations. In addition to using direct CPN simulators, CPN specifications have a very natural embedding in a general rule-based framework.

To clarify the relationship between agents, interpreters, and protocols, let us assume that a Java-based agent would like to converse with another agent, and that it has determined, through assumption, negotiation, or other means, that it needs to use protocol xyz. It can obtain the declarative specification for xyz, if it does not already have it, from the other agent or from some third party; let's say a protocol server identified through a broker. Xyz contains the wire-frame specification of the protocol (arcs, places, transition), plus schema and functions given in the IDL. The agent can then obtain the executable attachments (as it did the specification) and type specifications appropriate for its interpreter (in the case of Jackal, Java classes and associated methods), and then use the protocol to engage the other agent.

This CORBA-like approach allows the use of very lightweight, universal interpreters without restricting the expressiveness of the protocols used. Note that the purpose of the IDL in Protolingua however is the identification and retrieval of executable modules, not the interaction of distributed components. If types and actions are appropriately specified, they should be suitable for analysis, or translation into some analyzable form. For example, we are using DesignCPN, a tool from Aarhus University, Denmark, for high level design and analysis of protocols. This system uses an extension of ML, CPN-ML, as its modeling language. We plan to translate developed protocols into Protolingua and Java extensions, and restrict modification in such a way that CPN-ML equivalents of the extensions can be used to facilitate analysis of the protocols. As such, CPN-ML has played a major role in influencing the development of Protolingua. For the remainder of this paper, we will focus on the abstract application of CPNs to conversations, rather than their specification in Protolingua.

5 Example: Conversation Protocol

From its inception, Jackal has used JDFA, a loose Extended Finite State Machine (EFSM), to model conversations [6, 20]. The base model is a Deterministic Finite State Automaton (DFA), but the tokens of the system are messages and message templates, rather than simply characters from an alphabet. Messages match template messages (with arbitrary match complexity, including recursive matching on message content) to determine arc selection. A local read/write store is available to the machine.

CPNs make it possible to formalize much of the extra-model extensions of DFAs. To make this concrete, we take the example of a standard JDFA representation of a KQML Register conversation (see Figure 1) and reformulate it as a CPN. Note that this simplified Register deviates from the [16] specification, in that it includes a positive acknowledgment, but does not provide for a subsequent 'unregister' event. The graphic depiction of this JDFA specification can be seen in Figure 2.

```
// Conversation Template
// Convention: Initial and accepting states all caps,
//             other states initial caps,
//             arc-labels lower case.
(conversation
  (name kqml-ask-one)
  (author "R. Scott Cost")
  (date "3/5/98")
  (start-state START)
  (accepting-states STOP)
  (transitions
    (arc (label reg)   (from START) (to R) (match "(register)"))
    (arc (label reply) (from R) (to STOP)  (match "(reply)"))
    (arc (label error) (from R) (to STOP)  (match "(error)"))
    (arc (label sorry) (from R) (to STOP)  (match "(sorry)"))))
```

Figure 1: Conversation template for simplified KQML Register

There are a number of ways to formulate any conversation, depending on the requirements of use. This conversation has only one final, or accepting, state, but in some situations, it may be desirable to have multiple accepting states, and have the final state of the conversation denote the *result* of the interaction.

In demonstrating the application of CPNs here, we will first develop an informal model based on the simplified Register conversation presented, and then describe a complete

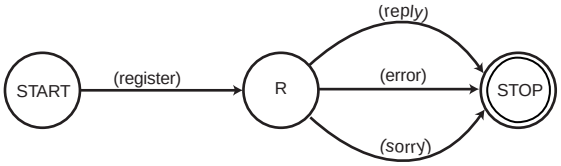

Figure 2: Diagrammatic DFA representation of the simplified KQML Register conversation

and working CPN-ML model of the full Register conversation.

Some aspects of the model which are implicit under the DFA model must be made explicit under CPNs. The DFA allows a system to be in one state at a time, and shows the progression from one state to the next. Hence, the point to which an input is applied is clear, and that aspect is omitted from the diagrammatic representation. Since a CPN can always accept input at any location, we must make that explicit in the model.

We will use an abbreviated message which contains the following components, listed with their associated variable names: performative(p), sender(s), receiver(r), reply-with(id), in-reply-to(re), and content(c).

We denote the two receiving states as places of the names **Register** and **Done** (Figure 3). These place serve as a receipt locations for messages, after processing by the transitions **T1** and **T2**, respectively. As no message is ever received into the initial state, we do not include a corresponding place. Instead, we use a a source place, called **In**. This is implicit in the DFA representation. It must serve as input to every transition, and could represent the input pool for the entire collection of conversations, or just this one. Note that the source has links to every place, but there is no path corresponding to the flow of state transitions, as in the DFA-based model.

The match conditions on the various arcs of the DFA are implemented by transitions preceding each existing place. Note that this one-to-one correspondence is not necessary. Transitions may conditionally place tokens in different places, and several transitions may concurrently deposit tokens in the same place.

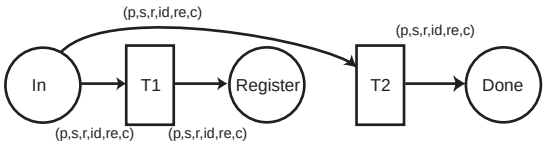

Figure 3: Preliminary CPN model of a simplified KQML register conversation.

Various constants constrain the actions of the net, such as performative (Figure 4). These can be represented as color sets in CPN, rather than hard-coded constraints. Other constraints are implemented as guards; boolean conditions associated with the transitions. Intermediate places **S**, **R** and **I** assure that sender, receiver and ID fields in the response are in the correct correspondence to the initial messages. **I** not only ensures that the message sequence is observed, as prescribed by the message IDs, but that only one response is accepted, since the ID marker is removed following the receipt of one correct reply. Not all conversations follow a simple, linear thread, however. We might, for example, want to send a message and allow an arbitrary number of asynchronous replies to the same ID before responding (as is the case in a typical Subscribe conversation), or allow a response to any one of a set of message IDs. In these cases, we allow IDs to collect in a place, and remove them only when replies to them will no longer be accepted. Places interposed between transitions to implement global constraints, such as alternating sender and receiver, may retain their markings; that is implied by the double arrow, a shorthand notation for two identical arcs in opposite directions.

We add a place after the final message transaction to denote some arbitrary action not implemented by the conversation protocol (that is, not by an arc-association action). This may be some event internal to the interpreter, or a signal to the executing agent itself. A procedural attachment at this location would not violate the conversational semantics as long as it did not in turn influence the course of the conversation.

This CPN is generally equivalent to the JDFA depicted in Figure 2. In addition to modeling what is present in the JDFA, it also models mechanisms implicit in the machinery, such as message ordering. Also, the JDFA incorporates much which is beyond the underlying formal DFA model, and thus cannot be subjected to verification. The CPN captures all of the same mechanisms within the formal model.

5.1 Register Implemented in CPN-ML

We further illustrate this example by examining a full, executable CPN implementation of the complete Register conversation. Register as given in [16] consists of an initial 'register' with no positive acknowledgment, but a possible 'error' or 'sorry' reply. This registration may then be followed by an unacknowledged 'unregister', also subject to a possible 'error' or 'sorry' response. This Register conversation (Figure 6) has been extracted from a working CPN model of a multi-agent scenario, implemented in CPN-ML, using the DesignCPN modeling tool. The model, a six agents scenario involving manufacturing integration, uses a separate, identical instance of the register conversation, and other KQML conversations, for each agent. They serve as subcomponents to the agent models, which communicate via a modeled network. The declarations (given in Figure 5) have been restricted to only those elements required for the register conversation itself. The diagram is taken directly from DesignCPN. The full model uses concepts for building hierarchical CPNs, such as place replication and the use of sub-nets, which are beyond the scope of this paper. The interested reader is encouraged to refer to [12, 13, 14].

The declarations specify a message format **MES**, a six-tuple of performative, sender and receiver names, message IDs, and content. For simplicity, performative and agent names in the scenario are enumerated, and IDs are integers. For the content, we have constructed a special **Predicate** type, which will allow us to represent content in KIF-like expressions. The **Reg** type is used for registry entries, and encodes the name and address of the registrant, the name of the registrar, and the ID of the registration message. Finally, the **Signature** type is used to bind the names of the sender and receiver with the ID for a particular message.

The model is somewhat more complex than our informal sketch (Figure 4) for several reasons, which will become clear as we look more closely at its operation. For one thing, it is intended to model multiple concurrent conversations, and so must be able to differentiate among them. Also, it implements the complete registration operation, rather than

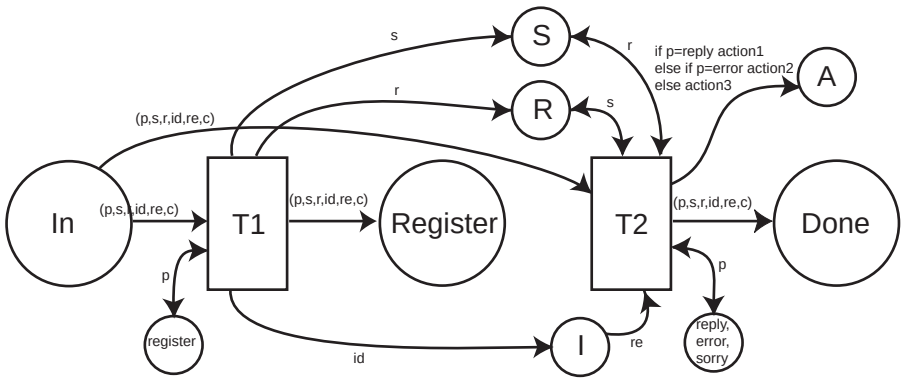

Figure 4: Informal CPN model of a simplified KQML register conversation.

simply modeling the message flow. All messages are initially presented in the **In** place, and once processed by each transition are moved to the **Out** place. Messages from the **Out** place are moved by the agent to the model network, through which they find their way to the **In** place of the same conversation in the target agent. The first transition (**T4**) accepts the message for the conversation, based on the performative 'register', and makes it available to the **T1** transition. **T1**, accepts the message if correct, and places a copy in the **Out** place. It also places an entry in the registry (**Reg**), and a message signature in **Sig1**. This signature will be used to make sure that replies to that message have the appropriate values in the sender and receiver fields. Message ID is included in the signature in order to allow the net to model multiple Register conversations concurrently. Note that because KQML does not provide for an acknowledgment to a 'register' message, the registration is made immediately, and is then retracted later if an 'error' or 'sorry' message is received.

Transition **T2a** will fire if an 'error' or 'sorry' is received in response to the registration. It unceremoniously removes the registration from **Reg**. The message signature constrains the names in the reply message. It is also possible for the initiating agent to send a subsequent 'unregister'; in that case **T2b** will fire (again, contingent on the constraints of the message signature being met), also removing the registration. However, since it is possible for an 'unregister' to be rejected (by an 'error' or 'sorry'), **T2b** archives the registration entry in **Arc**, and constructs a new signature for the possible reply. Such a reply would cause transition **T3** to restore the registration to **Reg**.

6 Example: Negotiation Model

In this section we present a simple negotiation protocol proposed in [5]. The CPN diagram in Figure 7 describes the pair-wise negotiation process in a simple MAS, which consists of two functional agents bargaining for goods. The messages used are based on the FIPA ACL negotiation performative set.

The diagram depicts three places places: **Inactive**, **Waiting**, and **Thinking**, which reflect the states of the agents during a negotiation process[1]; we will use the terms state and place interchangeably. Both agents in this simple MAS have similar architecture, differing primarily in the number of places/states. This difference arises from the roles they play in the negotiation process. The agent that begins the negotiation, called the *buyer* agent, which is shown on the left side of the diagram, has the responsibility of handling message failures. For this, it has an extra 'wait' state (**Waiting**), and timing machinery not present in the other agent, *seller*. For simplicity, some constraints have been omitted from this diagram; for example, constraints on message types, as depicted in the previous examples.

In this system, both agents are initially waiting in the **Inactive** places. The buyer initiates the negotiation process by sending a call for proposals ('CFP') to some seller, and its state changes from **Inactive** to **Waiting**. The buyer is waiting for a response ('proposal', 'accept-proposal', 'reject-proposal' or 'terminate'). On receipt, its state changes from **Inactive** to **Thinking**, at which point it must determine how it should reply. Once it replies, completing the cycle, it returns to the **Inactive** state. We have inserted a rudimentary timeout mechanism which uses a delay function

```
color Performative = with register | unregister
  | error | sorry;
color Name = with ANS | Broker | AnyName;
color ID = int;
color Address = with ans | broker | anyAddress;
color PVal = union add:Address + nam:Name;
color PVals = list PVal;
color PName = with address | agentName;
color Predicate = product PName * PVals;
color Content = union pred:Predicate + C;
color MES = product Performative * Name * Name
  * ID * ID * Content;
color Reg = product Name * Name * Address * ID;
color Signature = product Name * Name * ID;

var c : Content;
var message : MES;
var s, r, anyName, name : Name;
var i, j : ID;
var p : Performative;
var a : Address;
```

Figure 5: Declarations for the Register Conversation.

[1] It is not always the case with such a model that specific nodes correspond to states of the system or particular agents. More often the state of the system is described by the combined state of all places.

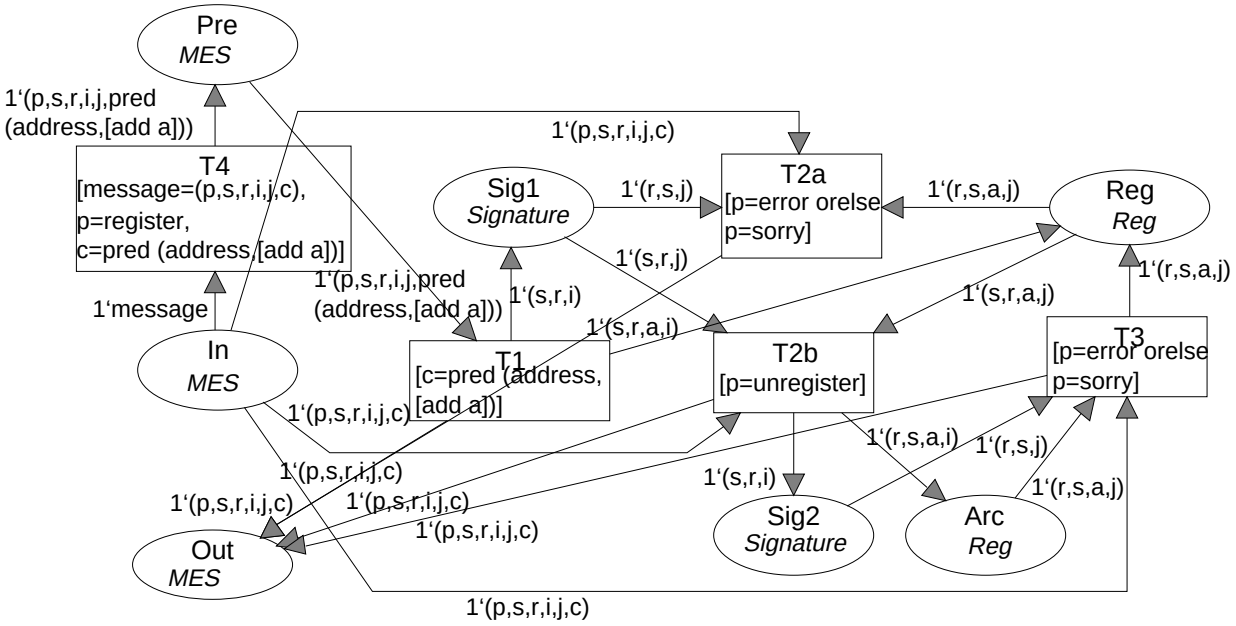

Figure 6: KQML Register.

to name messages which have likely failed in the **Timeout** place. This enables the exception action (**Throw Exception**) to stop the buyer from waiting, and forward information about this exception to the agent in the **Thinking** state. Timing can be handled in a number of ways in implementation, including delays (as above), the introduction of timer-based interrupt messages, or the use of timestamps. CPN-ML supports the modeling of time-dependent interactions through the later approach.

Note that this protocol models concurrent pairwise interactions between a buyer and any number of sellers.

7 Summary

The use of conversation policies greatly facilitates the development of systems of interacting agents. While FSMs have proven their value over time in this endeavor, we feel that inherent limitations necessitate the use of a model supporting concurrency for the more complex interactions now arising. CPNs provide many of the benefits of FSMs, while allowing greater expression and concurrency. Using the Jackal agent development platform, we hope to demonstrate the value of CPNs as the underlying model for a protocol specification language, Protolingua.

References

[1] Tilak Agerwala. Putting petri nets to work. *Computer*, pages 85–94, December 1979.

[2] Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 17–25, San Francisco, CA, 1995. MIT Press.

[3] Jeffrey M. Bradshaw, Stuart Dutfield, Pete Benoit, and John D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1998.

[4] Deepika Chauhan. JAFMAS: A java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati, 1997.

[5] Ye Chen, Yun Peng, Tim Finin, Yannis Labrou, and Scott Cost. A negotiation-based multi-agent system for supply chain management. In *Working Notes of the Agents '99 Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain.*, Seattle, WA, April 1999.

[6] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield, and Akram Boughannam. Jackal: A java-based tool for agent development. In Jeremy Baxter and Chairs Brian Logan, editors, *Working Notes of the Workshop on Tools for Developing Agents, AAAI '98*, number WS-98-10 in AAAI Technical Reports, pages 73–82, Minneapolis, Minnesotta, July 1998. AAAI, AAAI Press.

[7] Ian Dickenson. Agent standards. Technical report, Foundation for Intelligent Physical Agents, october 1997.

[8] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[9] FIPA. FIPA 97 specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents, october 1997.

[10] T. Holvoet. Agents and petri nets. *The Petri Net Newsletter*, (49):3–8, 1995.

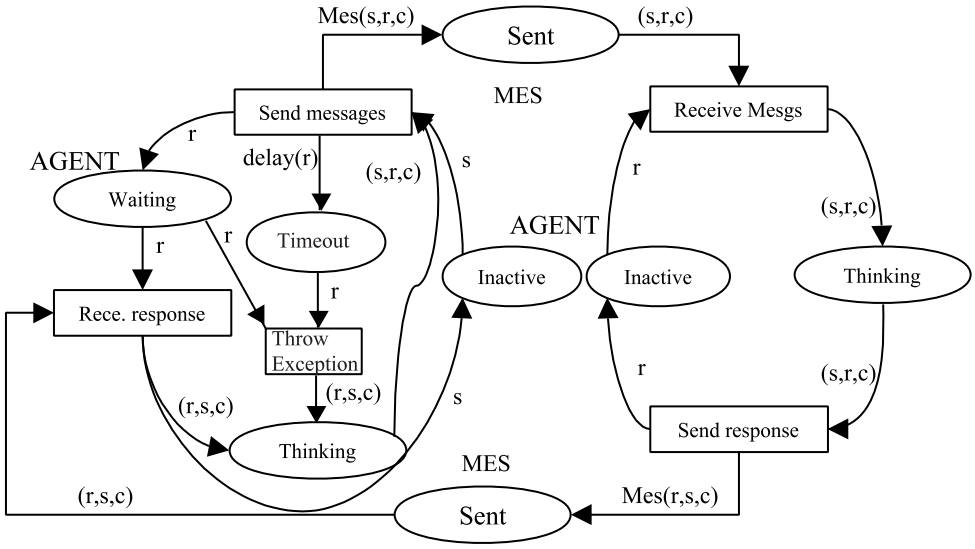

Figure 7: Pair-wise negotiation process for a MAS constituted of two functional agents.

[11] Tom Holvoet and Thilo Kielmann. Behaviour specification of parallel active objects. *Journal on Parallel Computing*, 24(7):1107–1135, 1998.

[12] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 1, Basic Concepts of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[13] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 2, Analysis Methods of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1994.

[14] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 3, Practical Use of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.

[15] K. Kuwabara. AgenTalk: Coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS '95)*. AAAI/MIT Press, 1995.

[16] Yannis Labrou. *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland Baltimore County, 1996.

[17] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured petri nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN '97)*, number 1248 in Lecture Notes in Computer Science, pages 82–101, Toulouse, France, June 1997.

[18] M. H. Nodine and A. Unruh. Facilitating open communication in agent systems: the InfoSleuth infrastructure. In Michael Wooldridge, Munindar Singh, and Anand Rao, editors, *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 281–295. Springer-Verlag, Berlin, 1997.

[19] H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced dooley graphs for agent design and analysis. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS '96)*, 1996.

[20] Y. Peng, T. Finin, Y. Labrou, R. S. Cost, B. Chu, J. Long, W. J. Tolone, and A. Boughannam. An agent-based approach for manufacturing integration - the CIIMPLEX experience. *International Journal of Applied Artificial Intelligence*, 13(1–2):39–64, 1999.

[21] M. Purvis and S. Cranefield. Agent modelling with petri nets. In *Proceedings of the CESA '96 (Computational Engineering in Systems Applications) Symposium on Discrete Events and Manufacturing Systems*, pages 602–607, Lille, France, July 1996. IMACS, IEEE-SMC.

[22] Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[23] James White. Mobile agents. In Jeffery M. Bradshaw, editor, *Software Agents*. MIT Press, 1995.

[24] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, 1986.