# Modeling and Verification of Distributed Autonomous Agents using Logic Programming*

L. Robert Pokorny and C. R. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, 11794-4400, U.S.A.
E-mail: `pokorny@xsb.com, cram@cs.sunysb.edu`

**Abstract.** Systems of autonomous agents providing automated services over the Web are fast becoming a reality. Often these agent systems are constructed using procedural architectures that provide a framework for connecting agent components that perform specific tasks. The agent designer codes the tasks necessary to perform a service and uses the framework to connect the tasks into an integrated agent structure. This bottom up approach does not provide an easy mechanism for confirming global properties of constructed agent systems. In this paper we propose a declarative methodology based on logic programming for modeling such procedurally constructed agents and specifying their global properties as temporal logic formulas. This methodology allows us to bring to bear a body of work for using logic programming based model checking to verify certain global properties of procedurally constructed Multi-Agent Systems.

## 1 Introduction

The Internet is fast becoming a venue for automated services. The advent of the Semantic Web and Web Services fosters an environment where complex services can be provided that are composed of a number of tasks. The tasks that compose the service are often accomplished by a group of autonomous agent programs. These agents communicate asynchronously over a LAN or the Internet to provide the desired service. Ideally, specifying agents as programs in a declarative logic programming language both facilitates the implementation of agent systems for desired service and also provides a formal model for proving that the implemented agent system performs the service with expected results.

While a number of high-level formalisms for specifying multi-agent systems have been proposed (see, e.g. [19, 3, 17]), many agent systems are being currently implemented in a procedural language such as Java. Development and deployment of agent systems using traditional languages such as Java has been simplified by the presence of frameworks that provide a rich array of services ranging

* This research was supported in part by NSF grants CCR-9876242, IIS-0072927, CCR-0205376, and CCR-0311512.

from communication and database interfaces to persistence and fault-tolerance (e.g., the Cognitive Agent Architecture, *Cougaar* [2]). It should be noted that the standardization efforts in the web services community (e.g. BPEL4WS [1]) have been oriented towards languages for specifying agent interfaces (e.g. the services offered and the types of data exchanged) to facilitate service discovery and composition, while leaving the implementation of the agents themselves unspecified. Although these developments alleviate some of the drudgery involved in constructing agents and provide facilities to compose agent systems, they do not provide mechanisms to give formal assurances about the behavior of agent systems. The interesting problem here is to develop methods and techniques to ensure that agent systems built in this manner exhibit certain desired properties. We outline here a declarative approach to addressing this problem.

Using a procedural agent architecture such as Cougaar, described in Section 2, agent systems are most easily developed in a bottom-up fashion. Individual agent programs are first built to perform specific tasks and then the allowable communications between agents are defined. The key to formally verifying the behavior of agent systems implemented in this manner is to first develop a formal model of the agent architecture itself. The main contribution of this paper is the development of a formal model of the main parts of the Cougaar architecture, including its persistence and fault-tolerance features. We then develop a framework, based on this model, to formally describe an agent system by specifying the behavior of the individual agent programs. The internal behavior of an agent is modeled as an extended finite-state automaton (EFSA), i.e., an automaton where states may be associated with variables and transitions may be guarded by constraints on values of the variables). In particular, the EFSA models a state transition system where there are a finite number of control states but potentially an infinite number of data states that can be partitioned into a finite number of data types. This is outlined in Section 3.

The intra-agent processes of an agent are presented as Horn clauses representing state transitions between control states in the EFSA. The EFSA for an agent describes the intra-agent actions. The behavior of the agent system can then be obtained as a concurrent composition of individual agent EFSAs and the architecture model that accounts for the possible synchronizations due to inter-agent communications.

The service being provided by an agent system is most easily described as a temporal process in which certain changes to occur to a set of objects in a certain order. This is a workflow-centric view of the service where its global properties are enumerated. The workflow describes the desired or, at least, anticipated outcomes of the service without making any explicit statements about the implementation details of the system of agents providing the service. While a graph-based workflow formalism can be used to easily specify certain required (or prohibited) behaviors of an agent system at a high-level, a more expressive temporal logic formalism can be used to describe complex properties such as availability, resilience to failure, etc.

We choose to represent workflow properties as temporal logic formulas for two reasons. First, temporal logic formulas make statements about infinite executions of EFSAs and, in particular, Linear Temporal Logic (LTL) [13] can represent fairness properties. Second, this formulation allows us to directly use the logic-based model checking techniques that have been developed in the past few years, (in which properties expressed in temporal logics can be directly verified for state transition models), to determine whether an agent implementation possesses certain high-level behavioral properties. Therefore in this paper, we use generalized linear temporal logic (GLTL), described in Section 4, which allows for statements about properties of states and labels on state transitions. GLTL is extended with data variables as the formalism for specifying behavioral properties. In Section 5 we present workflow properties represented in GLTL. We have developed model checkers for verifying GLTL properties for transition systems expressed as logic programs [15]. We can user this model checker to verify GLTL properties that depend on the control structure or data types in the model as long as the the GLTL formula being checked does not make starements that depend on the values of specific data objects. We also compare this to other work where agent systems expressed in Belief-Desire-Intension (BDI) agent languages are model checked for properties described in BDI temporal logics.

## 2 Cougaar, an implementation architecture for distributed autonomous agents

Cougaar is a Java based procedural implementation architecture for building systems of autonomous agents. It was originally funded by DARPA and is now maintained by an open-source community. It uses a design framework that handles both intra-agent data manipulation and inter-agent communications in a manner that provides transparency to the agent system designer. The architecture uses a distributed blackboard for inter-agent as well as intra-agent communication. This design framework provides persistence and recovery for individual agents and also system resilience against the loss of agents.

Data is stored and persisted at the agent level. Each agents keeps only the data necessary to perform its own functions. Data needed by more than one agent is shared by copying data objects from one agent to another. This distributed data model has the advantage that data is only stored where needed and dose not have to be made continuously available to all agents in the system. The disadvantage is that agents needing to share data are responsible for maintaining synchronization of that data. It is the responsibility of the agent designer to insure this synchronization.

At the agent level, all data is stored in a communal blackboard. The blackboard contains objects that are instantiations of Java classes representing items of interest to the agent. Objects are added to the blackboard either through communication with another agent or by an agent subprocess called a plugin. Plugins can also change or delete objects on the blackboard. Plugins are designed to be stateless processes that handle the computation required of the agent.

Plugins subscribe to objects on the blackboard and execute a defined procedure in response to changes in those objects. The executed procedure can query the blackboard about objects; add, change, or delete objects and publish these changes to the blackboard; change the plugin's subscription; or interact with the environment outside the agent system. Data on the blackboard is changed by the plugins, but the data changes are persisted by the agent control structure.

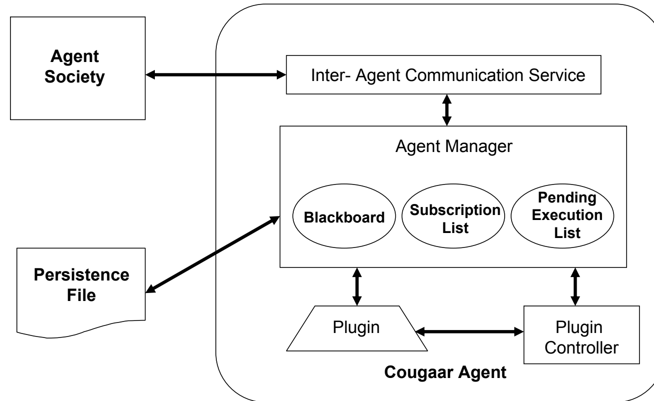


**Fig. 1.** Cougaar Architecture

The agent control structure is illustrated in Figure 1. When an agent starts up, it first instantiates a agent manager which contains a blackboard, subscription list, and plugin pending execution list and an inter-agent communication service. It then instantiates its component plugins. When a plugin is instantiated, it runs a subscribe method which notifies the agent manager about the objects in which it is interested. Once all plugins have been instantiated and have run their subscribe methods, the agent checks to see if any objects have been added to the blackboard which match a plugin's subscription. If so, that plugin is queued to run an execute method which can publish changes that add, modify, or delete blackboard objects. Whenever a change is published to the blackboard, plugin subscriptions are checked and the plugins affected by the change are added to the pending execution list and scheduled to run by the plugin controller.

Blackboard objects can also be communicated to other agents. The inter-agent messenger service sends copies of these objects as messages to other agents and also publishes added objects to the blackboard when they are received as messages from other agents. The state of the blackboard, subscription list, and

plugin pending execution list is persisted by saving to a file before every sent message and after every received message.

If an agent crashes and is then restored, the restoration proceeds in a similar fashion to agent initialization. The main difference is that agent state is restored from the persisted state file written during the last inter-agent communication before the crash. This method of restoring an agent coupled with the fact that copies of data objects are passed between agent blackboards means that when an agent is restored, it will have internal consistency but its blackboard might be out of synchronization with other agents in the system. In the Cougaar implementation it is up to the agent designer to provide inter agent synchronization if needed. Also Cougaar assumes that any state information that individual plugins need is embodied in data objects that the plugins publish to the blackboard.

We will use an order processing system as a running example of a Cougaar-based multi-agent system. In this example, a simple Cougaar agent would contain order objects on its blackboard. New orders would be received from other agents and cause order objects to be added to the blackboard. The order objects would contain a status flag that is set to received when the order is added. This order agent might have a capacity setting so that when the number of orders on the blackboard reaches a certain level no more orders will be accepted. Processing of orders in the agent would be handled by plugins. In the simplest case, a plugin would subscribe to order objects on the blackboard and be notified when orders are added. When notified, the plugin would execute and check an external database for credit and inventory information and change the status of the order to shipped, rejected, or back-ordered. The agent would then communicate these revised statuses to other agents in the system by sending a copy of the order object to the appropriate agent. An order with a shipped status might go to a billing agent, a rejected order to a customer notification agent, and a back-ordered order to a production scheduling agent. Once copies of the order objects are sent to these other agents the objects would then be removed from the processing agent's blackboard. As order objects are removed the capacity to receive and process new orders is correspondingly increased.

In summary, the plugins in each agent can be considered as *actions* taken by an agent with each plugin representing a specific action. The agent system is developed by specifying, albeit in a procedural form, the behavior of each plugin. Note that the development of an agent focuses on the detailed behaviors of the plugins. Combining the models of behaviors of each plugin with a detailed formal model of the behavior of the Cougaar architecture itself, we can derive the agent-wide and system-wide behaviors. Note, however, that the Cougaar architecture itself does not directly support the specification of global agent-wide and system-wide behaviors. Hence it is possible that the actual agent or system behavior deviates from its expected behavior. In the next section we introduce a declarative model of the Cougaar Agent Architecture.

## 3 A Declarative Model of the Cougaar Architecture

We now develop a high-level model of the Cougaar architecture. The model for an agent consists of a set of concurrent automata, one automaton for each component: the blackboard and agent manager, the communication interface, and the components representing plug-ins. The automata have a finite number of control locations with local variables, and transitions in the automaton may be guarded by conditions on the valuation of these variables. Each automaton, formalized as an *extended finite-state automaton* (EFSA) can be simply described by a logic program that represents its transition relation [18].

We represent the transition relation of an automaton in our model using the ternary relation `trans`. A tuple in this relation of the form `trans(S, A, T)` represents a transition from state `S` to state `T` labeled with action `A`. The states may be in general be *terms* representing both the control information (e.g. the program counter value at an agent state) and data values at a state. The action labels represent *events*: communication with other automata, or simply computation steps internal to the automaton. The labels for internal computations may specify additional parameters that qualify the computation. Labels for communication operations are written as terms either of the form $f(t_1, \ldots t_n)$ where $f$ is a function symbol, or of the form $\overline{f(t_1, \ldots, t_n)}$. The two are usually taken to represent an *input* action (where $f$ stands for the channel or port over which the communication takes place), and an *output* action, respectively. In our case we do not distinguish between input and output actions; rather than considering communication as a transmission of data from one automaton to another, we generalize the approach of CCS [14] and view communication as an agreement of data values in two automata. Two concurrent automata synchronize by simultaneously taking transitions with complemenary labels: e.g. $f(t_1)$ and $\overline{f(t_2)}$. At synchronization, the terms $t_1$ and $t_2$ are unified. In general, synchronization takes place only when the labels of the two transitions unify.

The transition relation model captures the details of the operational behavior of a Cougaar agent. However, such an explicit representation may become tedious to develop (and consequently, error-prone) when used to model large systems. Hence we represent the transition relation by a set of Horn clauses defining the relation, rather than as an explicit set of tuples.

We divide agent models into two parts: a generic part consisting of services provided by the Cougaar architecture, such as the blackboard service, communication service, etc; and a part specific to a particular agent instance, which is described by the behaviors of the plug-ins in the agent. The Cougaar architecture provides a rich variety of common services to simplify agent development and deployment. In terms of the behavioral models, this means that an agent model can be obtained by composing models of generic services (developed once and subsequently reused for all agents) with models describing the behaviors of the specific plugins. We first describe the models for Cougaar's generic services.

### 3.1 A Model of Cougaar's Generic Sevices

The blackboard service is central to a Cougaar agent. The blackboard serves as a storehouse for passive information— the objects manipluated by the different plugins within the agent— and at the same time actively participates in agent behaviours such as serving object change notifications to plug-ins, handling persistance, scheduling certain communication operations, etc.

The storage used by the blackboard service comprises of the following components:

1. the set of objects in the agent's blackboard (`data`)
2. the set of plugins pending execution in response to changes to data objects (`pending`)
3. the set of object subscriptions in which each plugin is interested (`subscription`)

We represent these three areas collectively by `store(D,P,S)` where `D`, `P` and `S` represent the above three storage areas respectively. In addition, to enable recovery from faults, an agent checkpoints its execution by saving the blackboard state at each intra-agent communication point. We model this persistence by representing a blackboard's state by `state(Current, Saved)` where `Current` is the representation of the current storage (a term of the form `store(...)`) and `Saved` is the representation of the storage at the last checkpoint.

The `data` part of a blackboard's storage is simply a set of objects. We use a notation borrowed from F-logic [12] to denote objects and use F-logic's mechanisms for representing an object store using attribute-value, subclass and instance relations. For instance, an object `Obj` belonging to class `Cls` and whose `status` field holds the value `new`, represented in F-logic by `Obj:Cls[status->new]`, will be stored in the blackboard's storage as tuples `instance(Obj, Cls)` and `attr(Obj, status, new)`. Evaluation of attribute values follow F-logic's inheritance mechanisms.

The `pending` list is a set of pairs of the form (*plugin*, *object*) where a change to the *object* matches the *plugin* subscription. The set of subscriptions associates a plugin with *subscription patterns* which are of the form (*class*, *change*), where *class* is the class of objects and *change* is the change flag for this subscription.

The blackboard is the arbiter of data and communication between the plugins and other Cougaar services in an agent. Plugins communicate synchronously with the blackboard using the following four primitives:

1. **query**: check the presence or absence of an object in the `data` area, and to retrieve information from objects in the `data` area
2. **modify**: add/delete objects to/from the `data` area
3. **subscribe**: add/remove self from subscription lists
4. **publish**: notify the rest of the agent system about changes made to the blackboard objects by this plugin

Apart from the data access operations from the agent's plugins, the blackboard also services communication requests from other agents. Although the Cougaar implementation separates the data service provided by the blackboard

from the communication services, it vastly simplifies the model to combine the two. A Cougaar agent may receive a `put` request to place an object in its blackboard from another agent; and may send objects, when requested to do so by its plugins, to other agents. Each of these requests (from plugins or other agents) represent events; the behavior of the generic services of Cougaar in response to these events (or when generating these events) is captured by the Horn clause rules in Figure 2 defining the `trans` relation.

Plugins are executed under the control of a plugin scheduler. Initially, the plugin scheduler invokes the `subscribe` method of each plugin which enables them to register with the blackboard service for object modification notifications. After the initialization phase is complete, the scheduler enters a loop, nondeterministically selecting a plugin to execute from the pending set in the blackboard, and invoking the corresponding plugin. The plugins, may in general, be run on a separate thread from the scheduler. We model the simpler and more common case where the plugins are sequentialized in the same thread as the scheduler. The transition relation of the scheduler's automaton can then be written as illustrated in Figure 3.

In the above, we assume that the `subscribe(Pin,C)` and and `execute((Pin, Obj),C)` correpond to the entry points of the subscribe and execute methods of a plugin `Pin`. The second argument `C` is the *continuation*: the state to which the methods return.

States of a system composed of two concurrent automata are represented by terms of the form `par(P1, P2)` where `P1` and `P2` represent the *local* states of the component automata. Operationally, an interleaving of the executions of two concurrent automata is an execution of the composition. In addition, the two automata may synchronize by unifying their action labels. The behavior of the concurrent composition of two automata is captured by the transition rules in Figure 4. It should be noted that synchronization by unification generalizes CCS's agreement-based synchronization for non-value-passing systems and synchronization by substitution for value-passing systems.

Note that with the above notation, it is straightforward to extend the model to deal with agents with multi-threaded plugins: instead of the *sequential* composition encoded by `execute((Pin,Obj),C)`, the scheduler loop will spawn `Pin` in an available concurrent thread and return immediately to picking up another plugin to notify.

When an agent crashes, the current state of the blackboard and other generic services is lost, and so are the local states of the plugins and the scheduler. When the agent recovers, it refreshes its state from the one saved at the last checkpoint, and resumes the scheduler loop. Thus, the crash and the eventual recovery of an agent can be captured by the transition rules given in Figure 5.

The `crash` and `recover` labels can be used in the model checker to specify properties to specify fair behaviors, considering only paths where `crash` occurs only finitely often, or those where `recover` occurs infinitely often.

```
% QUERY
trans(S, present(Q), S) :-
   S = state(store(Data,_,_),_), Q ∈ Data.
trans(S, absent(Q), S) :-
   S = state(store(Data,_,_),_), Q ∉ Data.

% MODIFY
trans(S, add(Q), T) :-
   S = state(store(Data,P,Subs), Saved),
   Data' = Data ∪ {Q},
   T = state(store(Data',P,Subs), Saved).
trans(S, delete(Q), T) :-
   S = state(store(Data,P,Subs), Saved),
   Data' = Data − {Q},
   T = state(store(Data',P,Subs), Saved).

% SUBSCRIBE
trans(S, subscribe(Pin, Class, Change), T) :-
   S = state(store(D,P,Subs), Saved),
   Subs' = Subs ∪ {sub(Pin, Class, Change)},
   T = state(store(D,P,Subs), Saved).
trans(S, unsubscribe(Pin, Class, Change), T) :-
   S = state(store(D,P,Subs), Saved),
   Subs' = Subs − {sub(Pin, Class, Change)},
   T = state(store(D,P,Subs), Saved).

% PUBLISH
trans(S, publish(Obj, Change), T) :-
   S = state(store(D,Pending,Subs), Saved),
   Notify = {Pin | subs(Pin, Class, Change) ∈ Subs, Obj:Class},
   Pending' = Pending ∪ Notify,
   T = state(store(D,Pending',Subs), Saved).

% PENDING_EXECUTION
trans(S, select(Pin, Obj), T) :-
   S = state(store(D,Pending,Subs), Saved),
   Pending' = Pending − {Pin},
   T = state(store(D,Pending',Subs), Saved).

% PUT
trans(S, put(Obj), T) :-
   S = state(store(Data,Pending,Subs), _),
   Data' = Data ∪ {Obj}
   Notify = {Pin | subs(Pin, Class, add) ∈ Subs, Obj:Class},
   Pending' = Pending ∪ Notify,
   SavedStore = store(Data', Pending',Subs),
   T = state(SavedStore, SavedStore).

% SEND
trans(S, put(Obj), T) :-
   S = state(Current, _),
   Current = store(Data,P,Subs),
   Data' = Data − {send(Obj)}
   NewStore = store(Data',P,Subs)
   T = state(NewStore, Current).
```

**Fig. 2.** Transition Relation for Generic Cougaar Services

```
% INITIALIZE
trans(scheduler, initialize, init(Pins, scheduler_loop)). :-
   initial_plugins(Pins).
trans(init([], S), A, T) :- trans(S, A, T).
trans(init([Pin|Pins], S), A, T) :-
   trans(subscribe(Pin, init(Pins, S)), A, T).

% EXECUTE
trans(scheduler_loop, select(Pin, Obj), execute((Pin, Obj), scheduler_loop)).
```

**Fig. 3.** Transition Relation for the Plugin Scheduler

```
% INTERLEAVE
trans(par(P1, P2), A, par(Q1, P2)) :-
   trans(P1, A, Q1).
trans(par(P1, P2), A, par(P1, Q2)) :-
   trans(P2, A, Q2).

% SYNCHRONIZE
trans(par(P1, P2), tau, par(Q1, Q2)) :-
   trans(P1, A, Q1),
   trans(P2, B, Q2),
   complement(A, B).

complement(L(X), L̄(X)).
complement(L̄(X), L(X)).
```

**Fig. 4.** Transition Relation for Parallel Composition

### 3.2 Modeling Specific Cougaar Agents

Having developed a detailed model for the generic Cougaar services, we can instantiate an agent by simply specifying (a) the set of plugins in the agent, and (b) the behaviors of their subscribe and execute methods. We illustrate such an instantiation by considering a simple order processing agent with a plugin `process_order` which takes an object of class `order` whose `status` field is `new`, and changes the order status field to one of `shipped`, `back_ordered` or `rejected`. For the purposes of this illustration, we will replace the logic for determining the status field with a nondeterministic choice. Orders processed by the agent

```
% CRASH
trans(agent(par(state(_,Saved), _)), crash, agent_crashed(Saved)).

% RECOVER
trans(agent_crashed(Saved), recover,
      agent(par(state(Saved,Saved), scheduler_loop))).
```

**Fig. 5.** Transition Relation for Crash and Recovery

then need to be transmitted to the other agents. The transition system for the execute method of this plugin can be written as:

```
trans(execute((process_order,order(Order)), C),
      delete(Order[status->new]), order_1(Order, C)).
trans(order_1(Order, C)), add(Order[status->NS]), order_2(Order, C)) :-
   choose_status(NS).
trans(order_2(Order, C)), send(Order) order_3(Order, C)).
trans(order_3(Order, C)), publish(Order, modify) C).

choose_status(shipped).
choose_status(back_ordered).
choose_status(rejected).
```

Since plugins typically have a simple structure (e.g. no thread creation, and usually no loops), we can simplify the specification of plugin behaviors by using a DCG-like notation that makes the states implicit. For instance, the above order plugin may be written as:

```
order(Order) -->
    [ delete(Order[status->new]) ],
    {choose_status(NS)},
    [ add(Order[status->NS]) ],
    [ send(Order) ],
    [ publish(Order, modify) ].
```

Each terminal symbol in the above DCG specifies only the action label of a transition, leaving the source and destination states implicit. It is easy to convert the above specification to the explicit transition rules given earlier. We can thus derive models of agent systems by modeling each plugin separately and combining these models with the models of generic services.

## 4 Linear Temporal Logic

We now review Linear Temporal Logic (LTL) and its extensions that are used for specifying temporal properties of finite-state systems. In particular we describe Generalized LTL (GLTL) which can make statements about properties of system states as well as action labels on transitions between states. GLTL has the following syntax ($P$ is the finite set of propositions and $A$ is the finite set of action labels):

$$\Psi \rightarrow \mathtt{A}\Phi \;\mid\; \mathtt{E}\Phi$$

$$\Phi \rightarrow p \mid \neg p \mid \alpha \mid \neg\alpha \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \;\mathtt{U}\; \Phi \mid \Phi \;\mathtt{R}\; \Phi \mid \mathtt{X}\Phi \qquad p \in P,\, \alpha \subseteq A$$

Formulas derived from $\Phi$ are called *path* formulas and formulas derived from $\Psi$ are state formulas Traditionally, GLTL is defined to include only $\mathtt{A}\Phi$; we consider the trivial addition of $\mathtt{E}\Phi$ since the model checking procedure we discuss is based on such formulae.

The semantics of GLTL is given in terms of infinite paths (called *runs*) of a Labeled Transition System (LTS). Runs are infinite sequences of states of the

LTS. The formal definition of GLTL semantics is standard (see, e.g. [6, 5]) and is omitted. Briefly, the semantics expresses how a run can satisfy a path formula. A formula $\Phi$ is true if $\Phi$ is true in the first state of a run. If $\Phi$ is $p$ then $p$ is a proposition that must hold in this state for $\Phi$ to be true. If $\Phi$ is $\alpha$ then the transition from the first state to the second state in the run must be labelled with an element in $\alpha$ for $\Phi$ to be true. For $\neg p$ and $\neg\alpha$, $p$ must be false and the transition label must not be an element of $\alpha$ respectively to make $\Phi$ true. $\mathtt{X}\Phi$ is true if $\Phi$ is true in the next state of a run, $\Phi_1 \wedge \Phi_2$ is true if both $\Phi_1$ and $\Phi_2$ are true for a given run. $\Phi_1$ $\mathtt{U}$ $\Phi_2$ is true of a run if $\Phi_1$ holds in every state until a state where $\Phi_2$ holds. $\Phi_1$ $\mathtt{R}$ $\Phi_2$ is true of a run if $\Phi_2$ holds in every state or until a state where $\Phi_1$ holds. $\mathtt{A}\Phi$ is true for state $s$ if $\Phi$ is true for all runs originating in $s$ and $\mathtt{E}\Phi$ is true if $\Phi$ is true for some run originating in $s$.

$\wedge$ and $\vee$ are duals. Similar to $\wedge$ and $\vee$, $\mathtt{U}$ and $\mathtt{R}$ are duals (i.e., $\neg(\phi_1 \ \mathtt{U} \ \phi_2) = \neg\phi_1 \ \mathtt{R} \ \neg\phi_2$), $\mathtt{E}$ and $\mathtt{A}$ are duals (i.e., $\neg\mathtt{A}\psi = \mathtt{E}\neg\psi$), and $\mathtt{X}$ is its own dual (i.e., $\neg\mathtt{X}\phi = \mathtt{X}\neg\phi$). It is easy to see that the standard semantics respects this duality.

To write more legible GLTL formulae, we define the following shorthand constructs for common GLTL formulas:

$$G\phi \equiv false \ \mathtt{R} \ \phi$$
$$F\phi \equiv true \ \mathtt{U} \ \phi$$
$$\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$$

$G$ is the global temporal quantifier. It is used to describe a property that is always true along a given path. F is the eventual temporal operator and describes a property that eventually becomes true along a path. The third shorthand is the standard logical implication.

Finally, GLTL can be enhanced by allowing terms containing logical variables to replace propositions. In the next section we describe the encoding of workflow properties about the expected global behaviors of agent systems in GLTL.

## 5 Workflows as Property Specifications

Agents and systems of communicating agents are built to provide specific services. Often these services are explicitly described by a workflow. Even when such an explicit definition is lacking, there is an implicit workflow which describes the anticipated outcome from invoking a service. The standard view of a workflow with respect to agents is that the workflow is a specification for the agent. In contrast, we consider the workflow as a specification of a property that the agent must exhibit.

Workflows have been directly represented in Transaction Logic [9]. One approach to showing that an agent system possesses a behavior expressed as a workflow would be to use Theorem Proving Techniques to show that the Transaction Logic representation of the workflow and the agent were equivalent. We believe a better approach is to express the workflow property in GLTL and use

Logic Programming based model checking to show that the GLTL formula holds for the EFSA model of the agent system.

GLTL is uniquely suited for representing workflow properties and more expressive than Transaction Logic for temporal properties. Workflows, in essence are temporal graphs that express sequences of events. Consider a simple workflow in which an order is first received and then shipped. The workflow implies an order to these two events, but no absolute time period between them. This is precisely the type of property that is easy to describe in GLTL.

If we let $\mathit{dependency}(\phi, \psi)$ stand for the GLTL state formula

$$G(\phi \Rightarrow X(F\psi))$$

We can write the following GLTL formula to describe the ordering property expressed in the above workflow as:

$$\mathtt{A}(\mathit{dependency}(\{\mathit{received}\}, \{\mathit{shipped}\}))$$

This states that along all paths if a *received* action occurs it is eventually followed by a *shipped* action.

Since the Cougaar agent model described above can crash, this property would not hold for it. The agent could crash between the *received* and *shipped* actions and never recover. This leads to describing fairness properties for which GLTL is also well suited. We would like to have the above property hold as long as the agent recovers from every crash. This can be written as:

$$\mathtt{A}(GF(\mathit{recover}) \Rightarrow \mathit{dependency}(\{\mathit{received}\}, \{\mathit{shipped}\})$$

Notice that neither the workflow or the above formulas say anything about what order is received or shipped. Implicit in the workflow is the idea that the workflow describes the events for a specific order. This can be handled by parameterizing the *received* and *shipped* actions, leading to:

$$\mathtt{A}(GF(\mathit{recover}) \Rightarrow \mathit{dependency}(\{\mathit{received}(\mathit{order}1)\}, \{\mathit{shipped}(\mathit{order}1)\})$$

Finally, the agent system is designed to run multiple instances of the specifying workflow so that we could be interested in properties that express ordering between workflow instances. For instance, we may want orders to be shipped in the order they were received. Enhancing GLTL with logical variables allows us to express these type of properties. We define $\mathit{ordered_events}(\phi, \psi)$ to stand for the GLTL formula:

$$F\phi \wedge F\psi \wedge \neg\psi \ \mathtt{U} \ \phi$$

which express that $\phi$ occurs before $\psi$. We can now express the property that orders are shipped in the order they are received as:

$$\mathtt{A}(\mathit{ordered_events}(\{\mathit{received}(\mathit{order}(X))\}, \{\mathit{received}(\mathit{order}(Y))\}) \Rightarrow \mathit{ordered_events}(\{\mathit{shipped}(\mathit{order}(X))\}, \{\mathit{shipped}(\mathit{order}(Y))\}))$$

This shows that GLTL is a logic that is well suited for specifying global properties of agent systems either as specifications of workflow properties or directly as fairness properties. GLTL also allows us to take advantage of logic programming for verification of these properties.

## 6 Ongoing Work and Concluding Remarks

Having been able to declaratively model a real world agent architecture as an EFSA and also express specifications for that system as temporal logic properties, we are now in a position to apply model checking techniques to verifying properties of agent systems.

We have been developing and using model checkers for finite and several classes of infinite systems based on logic programming [16, 10, 4]. We have also developed a model checker that can verify GLTL properties of labeled transition systems [15]. This model checker, implemented as a logic program, first constructs a Büchi automaton from a given GLTL formula, constructs the product of the given system model and the automaton, and performs good-cycle detection, i.e. cycles that meet the acceptance conditions of the automaton, to complete the model checking. Subsequently, we have also developed a constraint-based model checker where system models as well as properties are expressed using EFSAs [18]. This model checker can be directly used to verify properties of a Cougaar-based agent system. This model checker can verify certain class of infinite-state systems called data independent systems: those whose control behavior is independent of the domain of the data values. This is especially useful for the verification of agent systems since many aspects of their behaviors are data independent. For instance, the behavior of the ordering agent is independent of the domain of identifiers associated with different order objects. Thus we can use a constraint-based model checker to verify properties like the order of receiving and shipping of a specific order object with or without a fairness constraint on the agent crashing. It also allows us to check properties about the ordering of events. There is a complexity price to pay for this added capability. Standard model checking of finite-state systems runs in time linear in the size of the model. The constraint-based model checker in in the worst case exponential. Our future work will explore the limits and efficiency of using Logic Programming-based Model Checking to verify global behaviors of procedurally constructed MAS.

The main limitation of our approach is the representation of the blackboard. The blackboard is a part of an agent's state and we have to bound the number of objects that may be present simultaneously in the blackboard in order to ensure termination of verification runs.

The main contribution of this paper is the development of a logic-based high-level model of agent systems built using a procedural framework such as Cougaar. Since there are many such frameworks being proposed and implemented to address providing Web Services, this concept could have significant application. A secondary contribution of this modeling technique is that it allows us to verify

properties of MAS that are data independent infinite-state systems with finite control structures.

We want to point out how our work compares to other efforts in the field. There have been a number of presentations of applying model checking to verifying properties of MAS including [8, 20], These presentations model MAS in languages that have a direct translation to a finite-state labelled transition system and express properties to be verified in Belief-Desire-Intention (BDI) logics which can be transformed into propositional LTL properties. Our goal was to be able to verify properties of MAS developed in a procedural framework like Cougaar where global behavior is emergent and non-obvious. Also, by modelling such systems as EFSAs we do not need to limit our model to finite-state systems, but can consider properties of infinite-state data independent systems. This allows us to verify properties concerned with the general ordering of events. Our model also allows us to investigate fault tolerance of MAS expressed as GLTL fairness properties. There is an interesting parallel between Cougaar agents and BDI agents. Data on the Cougaar blackboard is similar to BDI beliefs, plugin subscriptions have a similar flavor to BDI desires, and plugins pending execution are similar BDI intentions. We feel that this similarity should be investigated, especially since properties expressed in BDI logics can easily be incorporated into an expansion of GLTL and be directly verified using our model checker. We see this as a fertile area for future work.

There has also been a considerable amount of work addressing workflows as specifications. Workflows have been represented in Transaction Logic [7] and their properties as theorems that satisfy these models [9]. In addition, [11] presents workflows modeled as UML Activity Diagrams and using LTL model checking to verify properties of these models. These approaches look at workflows as the model about which properties are stated. In our work we view the workflow as specifying global properties for a model of an independently constructed agent system. There are also a number of efforts to declaratively specify connectivity of autonomous agents using XML such as BPEL4WS cited earlier. These are primarily focused on finding and connecting agents that can compose a service, but they do not provide any method of verifying the behavior of the composition. What we propose allows the agent designer to use a procedural framework like Cougaar to build an agent system and gain some assurance about the conditions under which that system will exhibit expected behaviors.

## References


1. Business process execution language for web sevices (BPEL4WS). `http://www.ibm.com/developerworks/library/ws-bpel/`.
2. Cougaar: Cognitive agent architecture. `http://www.cougaar.org`.
3. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Logic programming for evolving agents. In *Intl. Workshop on Cooperative Information Agents (CIA'03)*, number 2782 in LNAI, pages 281–297. Springer Verlag, 2003.
4. S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS*, volume 2280 of *LNCS*, pages 236–250, 2002.

5. G. Bhat, R. Cleaveland, and A. Groce. Efficient model checking via beuchi tabeau automata. In *Computer Aided Verification (CAV)*, 2001.
6. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL$^*$. In *IEEE Symposium on Logic in Computer Science*. IEEE Press, 1995.
7. A. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, 1994.
8. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Second Internatonal Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2003.
9. H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 25–33. ACM, 1998.
10. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *IEEE Real Time Systems Symposium (RTSS)*, Orlando, Florida, 2000.
11. R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.
12. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1991.
14. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
15. L. R. Pokorny and C. R. Ramakrishnan. Model checking linear temporal logic using tabled logic programming. In *Workshop on Tabling in Parsing and Deduction (TAPD)*, 2000.
16. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV*, 1997.
17. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, number 1038 in LNAI, pages 42–55. Springer Verlag, 1996.
18. B. Sarna-Starosta and C. R. Ramakrishnan. Constraint based model checking of data independent systems. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, LNCS, 2003.
19. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, and F. Ozcan. *Heterogenous Agent Systems*. MIT Press, 2000.
20. M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In *First Internatonal Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 952–959. ACM Press, 2002.