Graph-Based Methods for the Analysis of Large-Scale Multiagent Systems

Wilbur Peng Alexander Grushin Vikram Manikonda
William Krueger Patrick Carlos Michel Santos

Intelligent Automation, Inc., 15400 Calhoun Drive, Suite 400, Rockville, MD 20855 1-301-294-5200

{wpeng,agrushin,vikram,wkrueger,sodre,msantos}@i-a-i.com

ABSTRACT
Multiagent systems are often characterized by complex, and sometimes unpredictable interactions amongst their autonomous components. While these systems can provide robust and scalable solutions to a variety of problems, the inherent complexity presents a barrier to their analysis, understanding, debugging and modification. In the work presented here, we seek to overcome this problem by modeling the execution of a multiagent system as a graph, which admits the application of techniques from the well-established field of algorithmic graph theory. In particular, we employ graph search and isomorphism computation to find repeating patterns of communication within a multiagent simulation. We argue, and demonstrate empirically, that the graph, even if it is too large to fit into main memory, carries useful properties, which allow these operations to be performed efficiently. We further show that the resulting patterns (which tend to be manageable in size) present a useful view of a simulation, and facilitate the comparison of different simulations against one another. This is specifically illustrated by applying a tool called *IntelliTrace*, which implements our approach, to multiagent models of the national airspace. At the same time, the tool, and its underlying methodology, is domain-independent, and can be used for the analysis, development and testing of a variety of multiagent systems.

Categories and Subject Descriptors
I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence – *multiagent systems*.

General Terms
Algorithms, Measurement, Performance, Verification.

Keywords
Methodologies/languages, distributed systems, evaluation techniques.

1. INTRODUCTION
Over the past two decades, decentralized systems of autonomous software agents have emerged as an important paradigm for solving a variety of computational problems [28]. In contrast to conventional software systems, which tend to be monolithic, and operate under centralized control, a multiagent system consists of independently acting entities (agents), which interact with each other to achieve some common goal. The latter, distributed approach is arguably more natural for certain application domains (such as those where a number of autonomous entities must be modeled/simulated, as in [16][17]), and carries many potential advantages, including modularity (which facilitates modification) and scalability via parallelism. Unfortunately, these advantages come at a price: even individually simple agents can, through their interaction, create complex, emergent behavior, which is difficult to predict and analyze. The action of some agent, whether it can be attributed to a change in the input or environment, a software modification, a bug, or some source of non-determinism, can affect the states of other agents, and manifest itself in non-intuitive ways. This makes very difficult the analysis of simulation behavior, which may be necessary in order to understand the effects of environmental or internal factors, to modify the system, or to uncover and correct bugs.

In this paper, we describe a novel approach to the analysis and understanding of multiagent systems. The basic idea behind the approach is that the entire lifetime of such a system can be modeled as a graph, where the nodes represent the *events* (roughly speaking, these can be viewed as actions) that are triggered on the various agents, and the edges denote *messages* that are sent during the execution of some event, and that can trigger other events. Although the resulting graph can be very complex (due to high numbers of agents that are active for extended periods of time), a vast arsenal of graph algorithms [26] is available for facilitating analysis. The specific approach considered here is to decompose the graph into a set of repeating *communication patterns* (e.g., agent A sends a message to agent B, which, in turn, sends messages to agents C and D). We argue that a pattern can usefully serve as a descriptive unit for the overall simulation, providing a manageable view of the local dynamics of the system at one or more points in time.

The approach has been implemented within a tool called *IntelliTrace*, which has been tailored to the domain-independent *CybelePro* agent framework. We demonstrate that (a) the tool is able to yield useful insight into complex multiagent interactions, and (b) it can perform the necessary data processing in an efficient manner, both when transforming raw simulation data into a graphical model, and when extracting patterns from the graph. Importantly, by construction, the graph carries certain useful properties, which are utilized in order to tractably find the patterns. As we show, the connected subgraphs within the graph are *trees*, which can be compared for isomorphism in polynomial time; this is necessary in determining which trees are instances of the same communication pattern. Further, the graph possesses

Cite as: Graph-Based Methods for the Analysis of Large-Scale Multiagent Systems, Wilbur Peng, Alexander Grushin, Vikram Manikonda, William Krueger, Patrick Carlos, Michel Santos, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 545–552
Copyright © 2009, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org), All rights reserved.

temporal locality, which allows it to be efficiently traversed, even when it is too large to fit into main memory [9].

The rest of the paper is organized as follows. Section 2 provides a brief summary of past work on multiagent systems analysis, as well as other relevant topics. Subsequently, to provide context for our own work, Section 3 outlines the structure of a multiagent simulation within the CybelePro framework. Section 4 shows how such a simulation can be modeled as a graph, and discusses how the graph's properties greatly facilitate the identification of communication patterns. The implementation of the approach is discussed in Section 5, and its efficiency is evaluated in Section 6. In Section 7, we demonstrate the utility of the approach, by applying it to a scenario from the air traffic management domain. Finally, in Section 8, we discuss our results, and present ideas for future research.

2. RELATED WORK

A number of other approaches to the analysis of multiagent systems have been developed in the past. Some are theoretical in nature, and have been developed both for generalized agent models [15] and for agents in a specific domain [7]. Other approaches, like our own, attempt to extract and elucidate the most relevant information about the execution of a multiagent system. For example, [18][25] present tools for visualizing multiagent simulation data. While the tools give a number of detailed and aggregate views of the simulation, they do not model agent interactions as event/message graphs, in the manner discussed here. The work of [14] presents a graphical model of expected agent behaviors; however, this model is not derived from a particular simulation, but rather, is partially developed by a human. A system called the *Tracer Tool* then compares the execution of the system against the model, and notifies the user of unexpected behavior. (By contrast, our approach requires no knowledge engineering; rather, graphs derived from different simulations are compared against each other, as demonstrated in Section 7). Further, in [2], a tool called the *TTL Checker* is developed to verify multiagent system execution traces against formally-specified properties. (The two aforementioned approaches have been used in combination, in [3]). The work of [21] applies a similar philosophy to the problem of multiagent system debugging; here, the model of expected agent behavior derives from the specifications (e.g., for interaction protocols) that were used in the initial design of the system.

Several relevant multiagent graphical models have been developed and implemented as part of a tool called *ACL Analyser* [4][24][27], which is integrated with the *JADE* agent framework [1]. These models include *communication graphs*, where nodes represent agents, and edges are drawn between those agents that communicate with each other; *causality graphs*, where nodes represent the states of different agents (e.g., an agent in some state caused another agent to change its state), and edges denote causal relationships; and *order graphs*, where nodes represent transmitted messages, and edges indicate temporal relationships. The models are somewhat different from our own (where nodes represent *events*); however, ACL Analyser also provides a means to visualize agent interactions as *sequence diagrams* [11], as is done here.

To our knowledge, the approach of describing a simulation in terms of communication patterns has not received much attention in past multiagent systems research. There has, however, been some work on the extraction of dataflow patterns from low-level machine code, where each pattern shows dependencies between atomic instructions [22]. Pattern-based analysis has also been performed in the context of optimizing communication amongst hardware components [20].

Our work places an emphasis on the fact that the amount of main memory available on most computing devices may be insufficient to store the event-message graph from a large-scale multiagent simulation. In our experience, storing and retrieving graphical data from a conventional, relational database (for example, this is proposed in [24]) proved to be too inefficient for our purposes. Instead, our approach is partially inspired by object-oriented databases [5], as well as work on algorithms for large graphs [6].

3. A DESCRIPTION OF CYBELEPRO

In this paper, we focus on simulations of agents that have been developed within the *CybelePro* agent framework (www.cybelepro.com). This framework provides a generalized methodology for specifying agent behaviors, along with a number of services for communication, distribution, concurrency control, etc. It has been used extensively for a variety of domains, including air traffic management [16][17], scheduling [23], and anomaly detection in networks [8]. Here, we describe the structure of a CybelePro simulation.

Each CybelePro *agent* is an autonomous entity, which resides on some physical machine, and can be subdivided into one or more *activities*, which act as its functional units. Two activities of different agents may only communicate by sending *messages* to each other; such messages can encapsulate arbitrary data. Activities within the same agent have the additional option of sending *internal messages* (which are more efficient, since they are transmitted within the confines of a single Java Virtual Machine), and may even share references to Java objects; for simplicity, we shall ignore such messages in this paper. Each message (whether regular or internal) is sent along some *channel*; an activity must *subscribe* to a given channel, in order to receive its messages.

To perform useful work, activities are able to execute sections of Java code (callbacks) at various points in the simulation; any such execution is called an *event*, and events fall into three categories. A *message event* is triggered when an activity receives a message. Likewise, an *internal event* is triggered by an internal message. Finally, *timer events* are not triggered by messages, but rather, occur at predetermined times in the simulation. To execute timer events, an activity must first subscribe to a *timer* (this is analogous to a channel subscription) in order to specify the future time(s) when the events are to occur.

Although many aspects of our approach are applicable to real-time multiagent systems (with a continuous clock), in this paper, we focus on discrete simulations. In such a simulation, each event occurs at some integral *major tick*, which is further subdivided into *minor ticks*; note that the actual, real-time duration of each tick can vary. The simulation begins at some major tick t_0. For any given major tick t, all timer events are said to occur at minor tick 0. If event e occurs at minor tick d, and sends a message (regular or internal) that triggers event e', then e' is said to occur at minor tick d+1. If, for some minor tick d', no event sends any messages that trigger other events, then the simulation proceeds to major tick t' and minor tick 0, where t' corresponds to the scheduled time of the earliest timer event that has not yet taken place; formally, $t' = \min_k(\{k \in T : k > t\})$, where T is the set of all times at which some timer event is scheduled to occur. If no timer events are scheduled (T is empty), then the simulation terminates.

CybelePro provides a *lockstep mechanism*, which ensures that, for any major/minor tick pairs (t, d) and (t', d'), if t < t', or if t = t' and d < d', then any event at (t, d) will complete before any event at (t', d') commences. Within a minor tick, the events of two different agents may execute in arbitrary order, or even in parallel; however, this should generally not affect the logic of the simulation, since agents cannot modify each others' state without sending messages (even if messages are sent, they are processed in the next minor tick). A mechanism is available to ensure that, within a single agent, the events of its activities execute in some deterministic order.

4. A GRAPHICAL MODEL OF A MULTIAGENT SIMULATION

Given a CybelePro multiagent system, our goal is to create a labeled graphical model G = (V, E, L) which captures its execution. One possibility is to denote the agents/activities within the system as nodes, and represent their communications as edges, as is done in [4]; however, two activities may communicate at disparate points in time, requiring the use of multiple edges between two nodes, which does not elegantly capture the temporal dimension of the simulation. Instead, we use a node $v \in V$ to represent each event that took place within the simulation; a directed edge $(u, v) \in E$ between nodes u, v represents a message (not internal) that is sent from within event u to event v. Each node is labeled with the name of the agent and activity that executed the corresponding event, and each edge is labeled with the name of the channel along which the event is sent. Formally, let us define the resulting graph as follows:

Definition 1: Let the simulation graph G be a triple (V, E, L), where each node $v \in V$ maps to some event in the simulation, and for each edge $(u, v) \in E$, there exists some message that was sent from the event represented by u, to the event represented by v. The labeling function L assigns the name of the executing agent/activity L(v) to v, and the message channel name L(u, v) to (u, v).

In our implementation of the model, nodes contain other information fields, such as the discrete time at which an event occurred, whereas edges store the contents of messages.

A large-scale simulation can involve many thousands of agents, which execute for extended periods of time, yielding a graph G that contains millions of edges and nodes. While the sheer size of a simulation can make it exceedingly difficult to analyze, there exists a large repertoire of graph algorithms [26], which can be applied to transform G, in order to facilitate its understanding. A specific technique, which is the focus of this paper, is to find small, connected subsets of G, which represent local patterns of inter-agent communication. The key philosophy behind our approach is that such patterns present us with a convenient view of the system's dynamics near some point(s) in time. Because this view is localized, a researcher is not overwhelmed with the full complexity of G; on the other hand, it allows him/her to identify causal relationships within the simulation, which would be difficult to accomplish if simulation data was to be presented as individual events or messages, without showing the relationship between them.

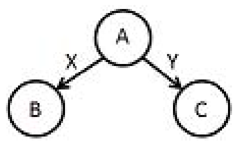


Figure 1. A tree that models agent communication.

To precisely define the notion of a communication pattern, let us first consider the structure of the main graph G. From Section 3, note that each event in a simulation is triggered by at most one message; thus, a node has at most a single incoming edge. Furthermore, the semantics of an edge (u, v) are: "u triggered v". Therefore, observe the following:

Property 1: G is a forest of trees.

Formally, a tree is defined as a connected graph that contains no cycles. Suppose, by way of contradiction, that some connected subgraph G' of G contains a cycle. Since no event is triggered by more than one message, the cycle can be written as the set of directed edges $\{(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)\}$. However, this is semantically interpreted as "v_1 triggered v_2, which triggered v_3, …, which triggered v_k, which triggered v_1". Clearly, this is an impossibility.

While Property 1 thus holds for the most general definition of a tree, for convenience, let us restrict the definition to the connected components of G (rather than their subcomponents):

Definition 2: T is a tree in the graph G if T is a connected subgraph of G, and there exists no other connected subgraph T' of G such that $T' \neq T$ and T is a subgraph of T'.

To facilitate the understanding of a simulation, it may be useful to search through G, and to extract all trees, in order to illustrate local patterns of communication that take place within the simulation; it is well-known that such a graph search can be done in polynomial time [26]. A simple example of a tree is shown in Figure 1, where agent/activity A sends messages to agents/activities B and C, along channels X and Y, respectively. Note, however, that this sort of communication may occur multiple times within a simulation; for example, A may periodically notify B and C of its status. A tree T in G thus represents an instance of some communication pattern, and it is of interest to aggregate similar trees into such patterns. Here, we define similarity in terms of graph *isomorphism*; specifically:

Definition 3: A communication pattern P in G is a collection of trees T in G, such that any two trees T_1, T_2 in P are isomorphic to each other, but are not isomorphic to any tree T_3 that is in G, but not in P.

Formally, we say that T_1 and T_2 are isomorphic if there exists a bijection h from the nodes of T_1 to the nodes of T_2, such that (u, v) is an edge in T_1 if and only if (h(u), h(v)) is an edge in T_2. Since we are dealing with labeled graphs/trees, an additional requirement is that L(v) = L(h(v)) and L(u, v) = L(h(u),h(v)), for any node or edge. In general, determining whether or not two arbitrary graphs are isomorphic to each other is believed to be a difficult computational problem, with no known efficient solution [26]. However, if the graphs involved are trees, then efficient, polynomial-time strategies do exist [26]. Thus, the structure of G greatly facilitates the identification of communication patterns.

From a practical point of view, however, it is important to note that G may be very large, and may not fit into the available amount of main memory. Thus, it becomes necessary to store portions of G externally (e.g. on a hard drive), and to swap them into memory as needed; the lists of extracted trees and aggregated patterns may also require external storage. Because input/output (I/O) operations require substantial time to execute, the swapping rate must be kept low, if the simulation is to be processed in a reasonable amount of time. How can this be accomplished?

In *virtual memory* systems [9], when a program requires access to some item d of data (identified by some address), it is first determined whether or not d is present in main memory. If this is not the case, then a *block* (page) b of data that contains d is fetched from the disk, and swapped into memory. A principle known as *locality of reference* [9] dictates that under the typical behavior of most programs, subsequent data requests will usually be to other items in b (or in other blocks that are presently in memory); thus, most requests for data will not require a swap. This maintains the rate of I/O operations at a reasonable level, and allows the program to execute in a satisfactory amount of time.

In traversing the simulation graph G, we must ensure that locality of reference is maintained. Here, an important observation is that G possesses *temporal locality*, defined as follows:

Property 2: For a node v in a simulation graph G, let t(v) and d(v) be the major tick and minor tick, respectively, at which the event (represented by v) occurred. For any edge (u, v) in G, t(u) = t(v), and d(u) = d(v) - 1.

This property follows as a natural consequence of the CybelePro timekeeping mechanisms (as described in Section 3), and is very beneficial, as it facilitates the efficient processing of simulation graphs. Suppose that the search algorithm encounters some edge (u, v) as it is attempting to extract some tree T. If the nodes and edges of G are stored in a time-ordered fashion, then u and v are expected to be relatively near each other on disk; thus, the traversal of (u, v) will not likely require a swap. Furthermore, note that by Definition 2 and Property 2, all nodes/events within some tree occur within the same major tick. If no major tick contains an excessively large number of events, then the nodes and edges of a given tree may be fully stored in main memory, for efficient access while the tree is being searched. In the following, we describe how we can take advantage of temporal locality to effectively implement a system for pattern identification, and provide metrics of its efficiency.

5. AN EFFICIENT IMPLEMENTATION

To transform raw simulation data into a set of communication patterns, we employ a process that consists of multiple stages, as depicted in Figure 2. The first stage takes place during the execution of the simulation, where a *profiling service* (which is provided as part of CybelePro) captures any events and messages that take place, along with information about created agents, activities, subscriptions, etc., and writes this data to disk in a condensed format; unlike in some approaches (e.g. [24]), agent code need not be modified for this purpose. If the simulation runs on several machines, then multiple streams of data are first transmitted from each machine to a centralized server; in this case, the stored data is not guaranteed to be perfectly ordered. However, we expect the degree of disorder to be small, because data transmission to the server is performed regularly on all machines; the logical fidelity of the data is not affected.

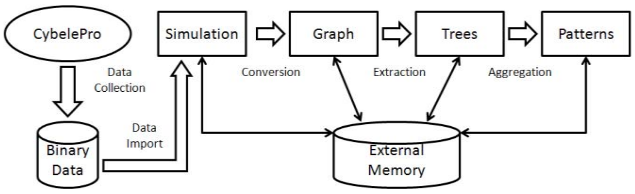
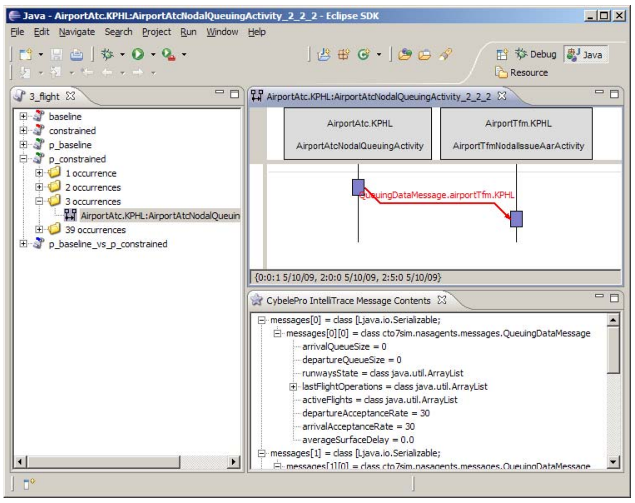


Figure 2. The data processing architecture. Key stages are represented via block arrows.


Java - AirportAtc.KPHL:AirportAtcNodalQueuingActivity_2_2_2 - Eclipse SDK
File
Edit
Navigate
Search
Project
Run
Window
Help
Debug
Java
Resource
3_flight
AirportAtc.KPHL:AirportAtcNodalQueuingActivity_2_2_2
AirportAtc.KPHL
AirportAtcNodalQueuingActivity
AirportTfm.KPHL
AirportTfmNodalIssueAarActivity
QueuingDataMessage.airportTfm.KPHL
CybelePro IntelliTrace Message Contents

```
baseline
constrained
p_baseline
p_constrained
    1 occurrence
    2 occurrences
    3 occurrences
        AirportAtc.KPHL:AirportAtcNodalQueuin
    39 occurrences
p_baseline_vs_p_constrained
```

```
{0:0:1 5/10/09, 2:0:0 5/10/09, 2:5:0 5/10/09}
```

```
messages[0] = class [Ljava.io.Serializable;
    messages[0][0] = class cto7sim.nasagents.messages.QueuingDataMessage
        arrivalQueueSize = 0
        departureQueueSize = 0
        runwaysState = class java.util.ArrayList
        lastFlightOperations = class java.util.ArrayList
        activeFlights = class java.util.ArrayList
        departureAcceptanceRate = 30
        arrivalAcceptanceRate = 30
        averageSurfaceDelay = 0.0
messages[1] = class [Ljava.io.Serializable;
    messages[1][0] = class cto7sim.nasagents.messages.QueuingDataMessage
```

Figure 3. The IntelliTrace tool interface. Sets of data (graphs, patterns, and a comparison of patterns) are shown on the left hand side. The visualized pattern (top right) reveals a simple communication between two agents. Patterns are displayed in the fashion of a sequence diagram [11] (similar diagrams are provided by the ACL Analyser tool [4][24][27]), where each header (top of pattern) represents an activity of some agent; for each activity, a lifeline extends downwards, in time, displaying each event as a rectangular box; arrows between boxes represent messages. The contents of the given message are displayed, in a hierarchical fashion, at the bottom. Times of occurrence are shown near the center of the figure, just below the pattern.

The remaining stages involve the post-simulation processing of the data, and take place within an analysis/visualization tool called *IntelliTrace*, shown in Figure 3, which we have implemented in the Java programming language, as a set of plug-ins for the Eclipse development framework. Rather than relying on the virtual memory capacity provided by the operating system (which is generally outside the control of the application programmer), the tool implements a custom mechanism for swapping data between main memory and the hard disk. This allows for control of the layout of stored data; importantly, the approximately temporal order of the data is preserved. This *external memory* system provides a clean interface for reading and writing data to specific addresses, automatically performing a swap whenever an address is not found in main memory.

The IntelliTrace tool reads the binary data files generated by the profiling service, and produces an object-oriented model that captures, via references between objects, the relationships between agents, activities, events, messages, etc. of a given simulation. Whereas the agent and activity objects can typically fit into main memory, event and message objects may be very numerous, and are stored in external memory. Such *external objects* are typically implemented to contain only one non-static data field, namely, their address in external memory. When a getter/setter method is called, data is read/written from/to an external memory location that is at some appropriate offset from the address. Collections of objects can be stored in an *external linked list*, where some items in the list are in main memory, and others are on the hard disk, at a given point in time.

Once built, the object-oriented model of a simulation is traversed to generate the simulation graph G; a node object is created for each event object, and edges are created for messages (a broadcast message that is sent along the same channel and received by k activities is converted into k edges that emanate from the same node). The nodes and edges are also stored in external memory, in the approximately-temporal order. Subsequently, tree extraction can be performed by iterating over the nodes of G, and for every node r that contains no incoming edges (i.e., a root node), performing a breadth-first graph search towards all nodes reachable from r, yielding a tree T rooted at r. To simplify the output, all isolated nodes (i.e., nodes that have no incoming or outgoing edges) are filtered out. Finally, the extracted trees T are aggregated, by isomorphism, into communication patterns.

As mentioned in Section 4, there exist efficient strategies for computing the isomorphism between two trees. Our approach is as follows: during the tree extraction phase, as a tree T is searched, we compute a *key list* K(T), which is a "flattened" representation of T, containing the labels of the various nodes and edges of T, along with the parent-child relationships between them. The key lists can then be compared for equality, to determine that two trees are isomorphic. Under each unique key list K_i produced, we store all trees T_j where $K_i = K(T_j)$, and aggregate these trees into a single communication pattern, since they are isomorphic to each other.

6. PERFORMANCE METRICS

In this section, we evaluate our algorithms, as implemented within IntelliTrace, in order to verify their ability to efficiently process large sets of data. For the purposes of generating this data, we made use of two different multiagent systems, both of which are used to model traffic within the national airspace, but are otherwise architecturally very distinct. The Airspace Concept Evaluation System (ACES) [17] uses agents to represent air traffic controllers, flights, and various other entities, and incorporates sophisticated models of management and control. The more recently-developed ACES-X [16] models flights as passive Java objects, which are communicated between different agents; at present, its control facilities are very limited. Our main goal here is not to compare the two systems, but rather, to show that our approach can be tractably applied to both.

We invoked both models on a schedule of 4758 flights, and used the profiling service (Section 5) to capture simulation data at the domain-independent, CybelePro level. Subsequently, we imported each set of data into IntelliTrace, which then built a graphical model based on the data, and extracted communication patterns from the graph. The experiments were performed on a Dell Dimension 9200 machine, with a dual core, 2.13 GHz processor, 2.00 GB of RAM and a 7200 rpm hard drive, running Windows XP, Professional Edition. The Java Virtual Machine (JVM), wherein IntelliTrace executes, was allotted 1 GB of RAM. In turn, our external memory system, as described in Section 5, used only 10^8 bytes of RAM to store its data, partitioned into 10,000 blocks of 10,000 bytes each. Blocks of data are swapped from the hard disk, as needed, into the location occupied by the least-recently-used block (which is swapped back to disk). In the current implementation, not all objects are stored within external memory; thus, there are certain limits to scalability. However, as we demonstrate below, considerably large sets can be handled effectively.

Table 1. Performance metrics

Type	Metric	ACES	ACES-X
Storage	Raw data set	1.29 GB	1.05 GB
	Simulation/Graph	2.64 GB	1.39 GB
	Trees/Patterns	2.57 GB	168 MB
Simulation Attributes	Agents	5738	508
	Activities	21,997	14,850
Graph and Pattern Measures	Nodes in graph	2,828,738	1,087,777
	Edges in graph	1,538,231	71,572
	Isolated nodes	712,770	971,215
	Patterns	96,682	14,966
	Avg. nodes in pattern	11.8	3.6
Processing Time	Data import	37 min	12 min
	Graph generation	13 min	2 min
	Tree extraction	28 min	1 min
	Pattern aggregation	21 min	< 1 min
External Memory	Blocks	538,576	158,181
	Reads	844,950	220,481

Table 1 summarizes the results of the experiments. For both simulations, the raw data sets already exceed the 1 GB allotment to the JVM. In the case of ACES, when this set is imported to produce a graph, along with an underlying object-oriented model of the simulation, the storage requirements double. In ACES-X, a (proportionally) much greater subset of the raw data consists of message contents, rather than events or messages themselves, which results in a considerably smaller graph, as indicated in the table. In both cases, the size of the typical communication pattern is orders of magnitude smaller than the size of the graph (size is defined as the number of nodes in the pattern, where each node corresponds to k nodes in the main graph, if the pattern occurs k times).

In spite of the large size of the data sets, the various data processing stages described in Section 5 can be performed in a tractable amount of time, requiring under 100 minutes for the ACES case, and only about 15 minutes for ACES-X. Once the graphical model or set of patterns has been generated, it can subsequently be loaded in less than a minute of time.

The last two rows of Table 1 show the number of data blocks (each one of size 10,000 bytes) within the external memory space, as well as the number of times that some block is read from the hard disk into main memory. Although certain blocks are swapped in multiple times during data processing, this occurs fewer than two times for the average block. This suggests that the system does not experience a great deal of thrashing [9], where blocks are repeatedly swapped between disk and main memory. Most memory accesses are *linear*, meaning that a set of data is processed approximately in the order of storage. This is achieved because the order of storage approximately corresponds to the temporal order of the simulation, thus allowing for locality of reference, and for tractable processing.

7. DEMONSTRATION

Here, we apply our developed approach to a concrete problem, which arose while conducting research on queuing models of air traffic. In order to develop the models, the ACES system was invoked to generate sets of data, and this data contradicted the researchers' expectations. Below, we describe the apparent inconsistency, and show how our approach helped to resolve it.

In a specific scenario, three flights are scheduled to simultaneously travel from the Philadelphia International Airport (PHL) to the Reagan National Airport (DCA). As shown in Figure 4, each flight begins at the departure airport. Upon taking off, and entering the terminal airspace (which surrounds the airport), it falls under the jurisdiction of the departure Terminal Radar Approach Control (TRACON) facility. The subsequent, en-route portion of the flight is controlled by ZNY, which is the New York Air Route Traffic Control Center (ARTCC) (the US National Airspace is subdivided amongst 20 such centers), and later, the ZDC ARTCC. Upon arrival, it passes through the space that is controlled by the arrival TRACON, and lands at the DCA airport. In the ACES system, each airport, TRACON and ARTCC is modeled by a specialized air traffic control (ATC) agent and traffic flow management (TFM) agent, the latter carrying on a more strategic, less reactive role than the former.

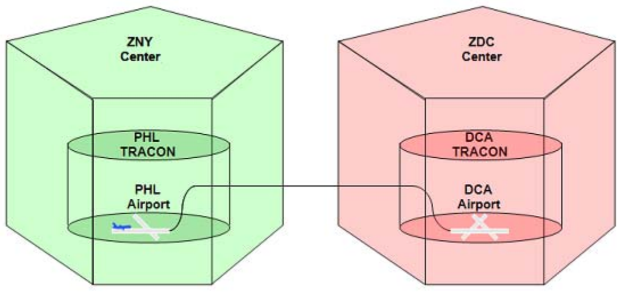
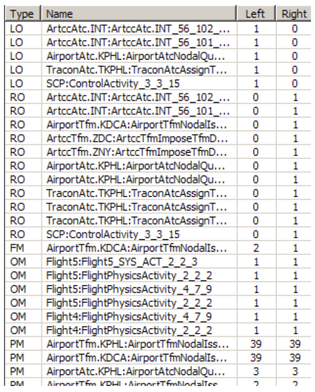


Figure 4. The demonstration scenario.

Two trials of the simulation were attempted: a baseline trial, wherein (unrealistically) any number of flights can take off or land at an airport, and a constrained trial, where takeoffs and landings must be spaced apart by two minute intervals. In the baseline trial, as scheduled, all three flights left the PHL gate simultaneously, and likewise, arrived simultaneously at the DCA gate. In the constrained trial, the first and second flights both left the gate simultaneously, and subsequently, the second flight was delayed (by two minutes) at the runway, prior to takeoff, as expected. On the other hand, the third flight was delayed at the gate by four minutes, and only then taxied to the runway. Thus, the landing of the three flights was (correctly) separated by two-minute intervals; however, why was the third flight not delayed at the runway, like the second flight?

To investigate this issue, we applied the IntelliTrace tool to the sets of data that were collected as the trials executed. Recall that the tool first generates a graphical model for each set of data. It is desirable to compare these graphs for differences, to illustrate how the introduction of the constraint on the takeoff/landing rate affects the overall simulation; however, comparing the graphs directly can be cumbersome. Instead, IntelliTrace extracts communication patterns from the two graphs, and compares sets of patterns against each other. The results of this comparison are illustrated in Figure 5. The vast majority of patterns (over 90%) occurred in both trials, with the same frequency, at the exact same simulation times. This greatly simplifies our analysis, since these patterns are unlikely to yield information about the effects of the introduced constraint. Instead, we specifically focus on the 11 patterns that occurred only in the constrained ("Right") trial.



Type	Name	Left	Right
LO	ArtccAtc.INT:ArtccAtc.INT_56_102_...	1	0
LO	ArtccAtc.INT:ArtccAtc.INT_56_101_...	1	0
LO	AirportAtc.KPHL:AirportAtcNodalQu...	1	0
LO	TraconAtc.TKPHL:TraconAtcAssignT...	1	0
LO	SCP:ControlActivity_3_3_15	1	0
RO	ArtccAtc.INT:ArtccAtc.INT_56_102_...	0	1
RO	ArtccAtc.INT:ArtccAtc.INT_56_101_...	0	1
RO	AirportTfm.KDCA:AirportTfmNodalIs...	0	1
RO	ArtccTfm.ZDC:ArtccTfmImposeTfmD...	0	1
RO	ArtccTfm.ZNY:ArtccTfmImposeTfmD...	0	1
RO	AirportAtc.KPHL:AirportAtcNodalQu...	0	1
RO	AirportAtc.KPHL:AirportAtcNodalQu...	0	1
RO	TraconAtc.TKPHL:TraconAtcAssignT...	0	1
RO	TraconAtc.TKPHL:TraconAtcAssignT...	0	1
RO	TraconAtc.TKPHL:TraconAtcAssignT...	0	1
RO	SCP:ControlActivity_3_3_15	0	1
FM	AirportTfm.KDCA:AirportTfmNodalIs...	2	1
OM	Flight5:Flight5_SYS_ACT_2_2_3	1	1
OM	Flight5:FlightPhysicsActivity_2_2_2	1	1
OM	Flight5:FlightPhysicsActivity_4_7_9	1	1
OM	Flight5:FlightPhysicsActivity_2_2_2	1	1
OM	Flight4:FlightPhysicsActivity_4_7_9	1	1
OM	Flight4:FlightPhysicsActivity_2_2_2	1	1
PM	AirportTfm.KPHL:AirportTfmNodalIss...	39	39
PM	AirportTfm.KDCA:AirportTfmNodalIs...	39	39
PM	AirportAtc.KPHL:AirportAtcNodalQu...	3	3
PM	AirportTfm.KPHL:AirportTfmNodalIss	2	2

Figure 5. A comparison of patterns from the baseline trial (mnemonically denoted by "Left"), and the constrained trial ("Right"), as displayed by IntelliTrace. Each row corresponds to some pattern, which occurred either in the "Left" trial only ("LO"), the "Right" trial only ("RO"), or in both trials. In the third case, there may be a frequency mismatch ("FM"), an occurrence mismatch ("OM"), or a perfect match ("PM"), where the pattern occurs in both simulations with the same frequency, at the exact same times. There were 228 perfect matches (only a few are shown for brevity).

By examination, three of the 11 patterns were found to be particularly informative, and are displayed in Figure 6. Each of these patterns occurs exactly once, and all three take place during the "planning" phase of the simulation, *before* any flights depart. The first pattern is initiated by the TFM of the *arrival* airport (DCA), which sends the TFM of the arrival TRACON the list of projected runway times for the second and third flight, in order to notify it of a potential need to delay these flights. Here, we note that in the ACES system, the TRACON is configured to absorb up to three minutes of delay for a given flight. Thus, the second flight (which needs to be delayed by two minutes, due to the constraint on landings) can be potentially held by the TRACON as it flies through the terminal airspace. On the other hand, the third flight must be delayed by four minutes (to separate it from the first two flights). Thus, the TRACON sends a message (highlighted in the figure) to the TFM of the ZDC center, requesting that the third flight be delayed further upstream.

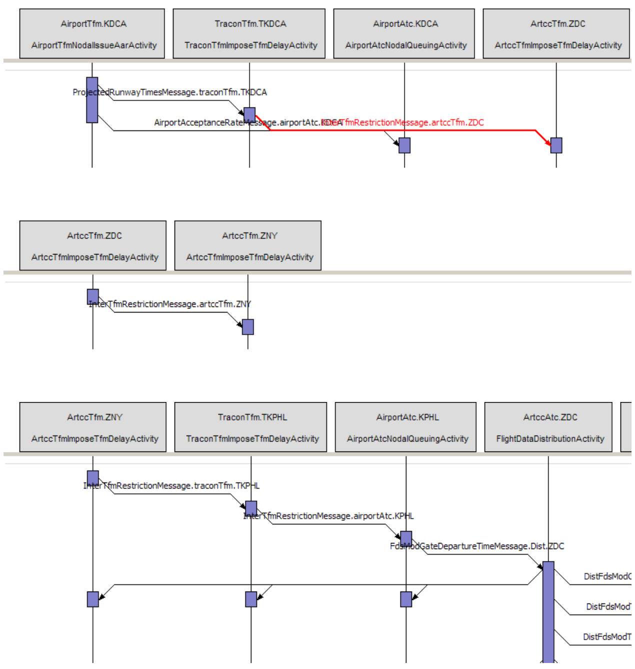

AirportTfm.KDCA
AirportTfmNodalIssueAarActivity
TraconTfm.TKDCA
TraconTfmImposeTfmDelayActivity
AirportAtc.KDCA
AirportAtcNodalQueuingActivity
ArtccTfm.ZDC
ArtccTfmImposeTfmDelayActivity
ProjectedRunwayTimesMessage.traconTfm.TKDCA
AirportAcceptanceRateMessage.airportAtc.KDCA
InterTfmRestrictionMessage.artccTfm.ZDC
ArtccTfm.ZDC
ArtccTfmImposeTfmDelayActivity
ArtccTfm.ZNY
ArtccTfmImposeTfmDelayActivity
InterTfmRestrictionMessage.artccTfm.ZNY
ArtccTfm.ZNY
ArtccTfmImposeTfmDelayActivity
TraconTfm.TKPHL
TraconTfmImposeTfmDelayActivity
AirportAtc.KPHL
AirportAtcNodalQueuingActivity
ArtccAtc.ZDC
FlightDataDistributionActivity
InterTfmRestrictionMessage.traconTfm.TKPHL
InterTfmRestrictionMessage.airportAtc.KPHL
FdsModGateDepartureTimeMessage.Dist.ZDC
DistFdsModG
DistFdsModT
DistFdsModT

Figure 6. Three communication patterns, as rendered by IntelliTrace, which reveal the effects of the constraint on takeoffs and landings.

The centers are configured to perform certain operations at 15 minute intervals. Thus, after 15 minutes of simulation time, another pattern occurs, where the ZDC center TFM sends a message to the ZNY center TFM, asking it to delay the third flight. The ZNY center, after yet another 15 minute interval, triggers a third communication pattern, where the delay request is sent to the TFM of the departure TRACON, which is forwarded to the ATC of the PHL airport, which then knows that the flight must be delayed. (Additionally, the PHL airport ATC sends a message to the ZDC center ATC, notifying it of the change in the flight's schedule, which in turn notifies other interested parties, as partially shown at the bottom of Figure 6). Thus, the request to delay the third flight propagates from the arrival airport to the departure airport, which delays the flight at the gate. On the other hand, for the second flight, the delay request does not come from the arrival airport (since its TRACON can potentially delay the flight on approach, if necessary), but rather, is accomplished by the departure airport (under its own constraint of spacing takeoffs by two minutes).

8. DISCUSSION

As the previous section demonstrates, one can obtain useful insight into the behavior of a multiagent system by modeling its execution as an event/message graph, and moreover, by converting this graph into a set of manageable communication patterns, which (to our knowledge) has not been attempted in past work on multiagent systems. In our experience, a pattern is typically small enough to permit tractable analysis, yet large enough to reveal the causality of interactions between agents within some local window(s) of time. Given sets of such patterns, one is able to illustrate key differences between simulations from which they derive, while factoring out common behavior. As we have illustrated, this approach allows the user to understand the effects of modifications to the input or configuration of a multiagent system (e.g. the introduction of constraints), and to explain unexpected behavior. It can likewise be useful in the development, testing and debugging of multiagent systems. As agent code undergoes modifications, our IntelliTrace tool can efficiently generate patterns from simulations, and compare them with patterns resulting from older versions of the code. While a perfect match between the pattern sets does not guarantee that the modifications did not introduce undesirable effects, it does suggest that the overall flow of the simulation remained unchanged.

While the IntelliTrace tool has been developed specifically for CybelePro simulations, we hypothesize that our approach can be extended to other agent frameworks/toolkits (e.g. [1][13][19]). The sending of a message from an agent to one or more other agents is a concept common to the vast majority of multiagent systems, and can be modeled (as is done here) via one or more edges that emanate from a node. The resulting graph is a forest of trees (Property 1 in Section 4), and thus, isomorphism computations can be performed efficiently, in order to find communication patterns. There are, of course, challenges in the generalization of the approach, since Property 2 (Section 4) may not necessarily hold. In a discrete CybelePro simulation, a new major tick does not commence until all present communication ceases; thus, any communication tree is contained within the boundaries of some major tick. In a real-time multiagent system (whether under CybelePro or under other frameworks), the size of a tree is not bounded, and it may be necessary to split these trees into multiple subtrees. Furthermore, if some variant of lockstep synchronization (Section 3) is not enforced, then the children of some node may correspond to events that occurred at vastly different points in time, and the degree of temporal locality in the graph will be reduced, resulting in greater I/O overhead during processing (if the graph cannot fit into main memory).

We believe that these challenges are a worthy subject of future research, since graph-based methods can potentially be useful in the analysis of multiagent systems developed under different frameworks, and for various domains. In the context of CybelePro, we have recently extended the IntelliTrace tool with the ability to capture and represent the real-time durations of events, and have been using it to identify performance bottlenecks in multiagent simulations. Other promising areas for future research include the application of link mining algorithms [12] to multiagent simulation graphs. Such methods have often been used for the analysis of networks that consist of human individuals, but they can certainly be extended towards software agents; in fact, parallels between the fields of link mining and multiagent systems have been drawn in the past [10]. As multiagent systems become a more prominent feature of practical computation, we expect this research to increase in its impact and importance.

9. ACKNOWLEDGMENTS

This work has been supported by NASA SBIR contract NNA07BA41C. The scenario discussed in Section 7 was investigated under NASA NRA contract NNX07AP16A. We wish to thank Goutam Satapathy and Sapna George for providing respective expertise on CybelePro and ACES.

10. REFERENCES

[1] Bellifemine, F., Poggi, A., and Rimassa, G. 1999. JADE - A FIPA-Compliant Agent Framework. Technical Report. Telecom Italia.

[2] Bosse, T., Jonker, C., van der Meij, L., Sharpanskykh, A., and Treur, J. 2006. Specification and Verification of Dynamics in Cognitive Agent Models. In Proceedings of the Sixth International Conference on Intelligent Agent Technology, 247-254.

[3] Bosse, T., Lam, D., and Barber, K. S. 2006. Automated Analysis and Verification of Agent Behavior. In Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems, 1317-1319.

[4] Botía J., Hernansáez, J., and Gómez-Skarmeta, A. 2006. On the Application of Clustering Techniques to Support Debugging Large-Scale Multi-Agent Systems. In Proceedings of the Fourth International Workshop on Programming Multi-Agent Systems, 217-227.

[5] Cattell, R. 1994. Object Data Management: Object-Oriented and Extended Relational Database Systems. Addison-Wesley.

[6] Chiang, Y.-J., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., and Vitter, F. 1995. External-Memory Graph Algorithms. In Proceedings of the Sixth Annual ACM Symposium on Discrete Algorithms, 139-149.

[7] Cicirello, V. 2001. A Game-Theoretic Analysis of Multi-Agent Systems for Shop Floor Routing. Technical Report. Robotics Institute, Carnegie Mellon University.

[8] Deng, H., Xu, R., Li, J., Zhang, F., Levy, R., and Lee, W. 2006. Agent-Based Cooperative Anomaly Detection for Wireless Ad Hoc Networks. In Proceedings of the Twelfth International Conference on Parallel and Distributed Systems.

[9] Denning, P. 2005. The Locality Principle. Communications of the ACM 48, 19-24.

[10] desJardins, M. and Gaston, M. 2006. Speaking of Relations: Connecting Statistical Relational Learning and Multi-Agent Systems. In Proceedings of the Workshop on Open Problems in Statistical Relational Learning.

[11] Fowler, M. and Scott, K. 1999. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley.

[12] Getoor, L. and Diehl, C. 2005. Link Mining: A Survey. SigKDD Explorations (Special Issue on Link Mining) 7, 3-12.

[13] Graham, J., Decker, K., and Mersic, M. 2003. DECAF - A Flexible Multi Agent System Architecture. Autonomous Agents and Multiagent Systems 7, 7-27.

[14] Lam, D. and Barber, K. S. 2005. Comprehending Agent Software. In Proceedings of the Fourth International Conference on Autonomous Agents and Multiagent Systems, 586-593.

[15] Lerman, K. and Galstyan, A. 2001. A General Methodology for Mathematical Analysis of Multi-Agent Systems. Technical Report. University of Southern California.

[16] Manikonda, V., George, S., and Robinson, C. 2008. Agent-Based Modeling and Simulation of the ACES Terminal Area Plant Model. In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit.

[17] Meyn, L., Windhorst, R., Roth, K., Van Drei, D., Kubat, G., Manikonda V., Roney, S., Hunter, G., Huang, A., and Couluris, G. 2006. Build 4 of the Airspace Concept Evaluation System. In Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit.

[18] Ndumu, T., Nwana, H., Lee, L., and Collis, J. 1999. Visualising and Debugging Distributed Multi-Agent Systems. In Proceedings of the International Conference on Autonomous Agents, 326-333.

[19] Nwana, H., Ndumu, T., Lee, L., and Collis, J. 1999. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. Applied Artificial Intelligence Journal 13, 129-186.

[20] Ogras, U. and Marculescu, R. 2005. Energy- and Performance-Driven NoC Communication Architecture Synthesis Using a Decomposition Approach. In Proceedings of the Conference on Design, Automation and Test, 352-357.

[21] Poutakidis, D., Padgham L., and Winikoff, M. 2002. Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols. In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, 15-19.

[22] Sassone, P. and Wills, D. 2004. On the Extraction and Analysis of Prevalent Dataflow Patterns. In Proceedings of the Seventh International Workshop on Workload Characterization, 11-18.

[23] Satapathy, G., Lang, J., and Levy, R. 2000. Application of Agent Building Tools in Factory Scheduling and Control Systems. In Proceedings of the International Symposium on Intelligent Systems and Advanced Manufacturing.

[24] Serrano E. and Botía J. 2008. Infrastructure for Forensic Analysis of Multi-Agent Systems. In Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems.

[25] Thomas, S., Mueller, J., Harvey, C., and Surka, D. 2001. Monitoring and Analysis of Multiple Agent Systems. In Proceedings of the Second GSFC Workshop on Radical Agent Concepts.

[26] Valiente, G. 2002. Algorithms on Trees and Graphs. Springer.

[27] Vigueras G. and Botía J. 2007. Tracking Causality by Visualization of Multi-Agent Interactions Using Causality Graphs. In Proceedings of the Fifth International Workshop on Programming Multi-Agent Systems, 190-204.

[28] Wooldridge, M. 1999. Intelligent Agents. In Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence, G. Weiss, Ed. MIT Press, 27-77.