Debugging Agent Behavior in an Implemented Agent System

Dung N. Lam and K. Suzanne Barber

The Laboratory for Intelligent Processes and Systems,
Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, TX 78712, USA
{dnlam, barber}@lips.utexas.edu
http://www.lips.utexas.edu

Abstract. As agent systems become more sophisticated, there is a growing need for agent-oriented debugging, maintenance, and testing methods and tools. This paper presents the Tracing Method and accompanying Tracer tool to help debug agents by explaining actual agent behavior in the implemented system. The Tracing Method captures dynamic runtime data by logging actual agent behavior, creates modeled interpretations in terms of agent concepts (e.g. beliefs, goals, and intentions), and analyzes those models to gain insight into both the design and the implemented agent behavior. An implementation of the Tracing Method is the Tracer tool, which is demonstrated in a target-monitoring domain. The Tracer tool can help (1) determine if agent design specifications are correctly implemented and guide debugging efforts and (2) discover and examine motivations for agent behaviors such as beliefs, communications, and intentions.

1 Introduction

There are several agent-oriented software design methodologies (e.g. GAIA [1], MaSE [2], and OMNI [3]) and development environments (e.g., JADE [4], ZEUS [5], and FIPA-OS [6]), but there are few agent-oriented methods and tools that have been created for debugging, maintaining, or testing the resulting implemented system. This paper presents the Tracing Method and accompanying Tracer tool whose purpose is to help better comprehend actual agent behavior in the implemented agent system in terms of familiar agent concepts. The objective is to ensure that an agent is performing actions for the right reasons and, if an unexpected action occurred, to help explain why an agent decided to perform the action. Due to the increasing sophistication of agent software (in particular, the autonomy, proactivity, and social features of agents) and the number of factors to consider when understanding or explaining agent behavior, comparing the implementation's actual behavior with expected behavior can be an intensive task, requiring time and knowledge about the design, implementation, and application domain. In more general terms, "concurrency, problem-domain

R.H. Bordini et al. (Eds.): PROMAS 2004, LNAI 3346, pp. 104–125, 2005.

uncertainty, and non-determinism in execution together conspire to make it difficult to comprehend the activity in a distributed intelligent system" [7]. Despite its difficulty, the task of comprehending agent behavior in the implemented agent system helps to identify undesired agent behaviors (bugs) and to gain insight on how the agents can be improved (for maintenance and testing). In addition to helping developers test and improve implementations produced by themselves or by others (e.g., open-source agent systems), the Tracing Method can help the end-user better understand the agents, and thus increase the end-user's confidence on agent decisions.

A motivation for applying agent-oriented techniques is to make the problem and solution easier to understand (i.e., by localizing beliefs and goals into autonomous agents), but after the agent system has been implemented as source code, distinguishing agent concepts (e.g., beliefs, goals, and intentions) in the implementation can become difficult as the complexity of the implementation increases. The design is specified in terms of agent concepts; however, the implementation is often specified in terms of the programming language structures, such as variables, classes, and flow-control statements. There are agent-oriented tools to generate the implementation from the design (e.g., Doi *et. al.*'s work on generating source code from AUML diagrams [8]), but there are few agent-oriented tools to extract the design concepts (i.e., agent concepts) from the implementation. This research aims to extract agent concepts from the implementation and to regain the advantages of conceptualizing the implementation in terms of agent concepts. To complement the high-level agent concepts, low-level details (e.g., belief values and communication message contents) related to the agent concepts are required for debugging the implemented agent system. Such details are made accessible by the Tracer tool, which aims to alleviate the largely manual task of comprehension during software maintenance.

This paper describes the Tracing Method to observe and interpret actual agent behaviors in terms of agent concepts, the same agent concepts that are used to describe expected agent behaviors in the design. Because agent behaviors in both the design and implementation are understood in terms of agent concepts, establishing the set of agent concepts is central to this research. Section 2 presents the proposed set of agent concepts used for describing agent behaviors. Section 3 outlines the Tracing Method for observing and extracting actual agent behaviors from the implementation's execution so that those actual behaviors can be compared with expected (or designed) agent behaviors. Section 4 demonstrates the Tracer tool in a UAV (Unmanned Aerial Vehicles) target-monitoring domain, where the implemented agents use MDPs (Markov Decision Processes) to decide their actions. The Tracer tool was found to be useful for quickly identifying and understanding the reasons for agent actions in terms of agent concepts. Section 5 describes how this research relates to existing work.

A demonstration of the Tracer tool applied to the UAV domain and to a simple multiagent system (along with example Java source code) is available for download at the website `http://www.lips.utexas.edu/~dnlam/tracer.html`.

2 Agent Concepts

Agent concepts are used in software designs to describe expected agent structure (e.g., an agent encapsulates localized beliefs, goals, and intentions) and behavior (e.g., an agent performs an action when it believes the event occurred). During and after development, agent concepts are used to abstract away from the details of the implementation and to understand and explain actual agent behavior (i.e., to answer the question "Why did agent $a1$ perform action α"). A desirable explanation could be "Action α was performed by agent $a1$ because $a1$ believed belief $b1$, which was due to the occurrence of event e in the environment, which was an expected consequence of agent $a1$ performing intention i, which was created based on belief $b2$ as communicated in a message from agent $a2$ about $a2$'s goal g." Other agent concepts that may be of interest include the roles the agents played during the interactions. An objective of this research is to automate some of the currently-manual tasks that a human must do to comprehend the implemented agent system.

This section describes agent concepts, focusing on their relationship with each other. The proposed set of agent concepts includes *goal*, *belief*, *intention*, *action*, *event*, and *message*. These agent concepts have a general definition or understanding in the agent community, but due to the variety of approaches and applications, there is no definitive representation for the agent concepts. Figure 1 illustrates the relationships among these agent concepts, and Table 1 presents the representation of agent concepts used in this research.

Table 1. Agent concept structure declarations

Agent Concept	Constituent attributes
event	name, preconditions, postconditions
action	agent, name, preconditions, postconditions
message	sender agent, receiver agent, subject, value
belief	agent, subject, value
goal	agent, name
intention	agent, name, goal names, belief subjects, action names

Agents are distributed, goal-oriented entities situated in an environment and encapsulate decision-making capabilities. Agents need their own *goal*(s) in order to be proactive (i.e., take initiative to achieve some goal) and autonomous (i.e., make decisions on their own based on their goals). In addition to localized beliefs about itself, agents also maintain *beliefs* about the environment, including objects situated in the environment. Beliefs are subjective representations of the state of the agent or the system and can affect many other aspects of the agent, including its goals. Using its current beliefs, an agent achieves a goal by generating an *intention* (or plan), which prescribes actions that the agent(s) intend to perform. *Actions* performed by agents and other entities can cause *events* in the environment, which agents may sense and use to update their beliefs. For

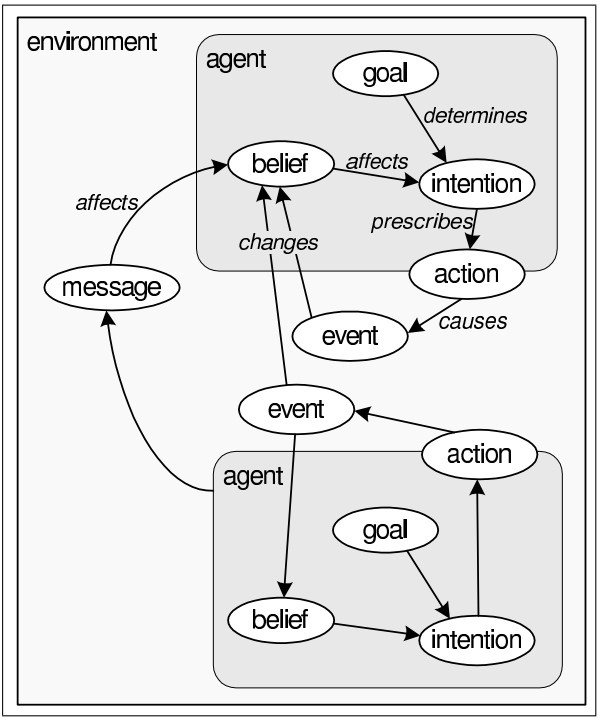

Fig. 1. Agent concepts and their relationships

explaining agent behavior, an agent's goals, beliefs, and intentions, in addition to its actions, must be considered because agents may act as expected but for undesirable reasons.

For multi-agent systems, communication is often an important factor to system performance. An agent may send *messages* to others during belief maintenance (for knowledge-sharing), during planning (for collaboration), or during schedule execution (for coordination). In terms of agent concepts, a communicated message can (directly or indirectly) affect an agent's goal, belief, intention, and/or action. Thus, an explanation of an agent action should include communicated messages that contributed to that action.

This research declares general structures for each agent concept as shown in Table 1. Unlike formal representations (e.g., goal representations for BDI agents [9]), the agent concepts are generalized so that they can be used in any implementation and to minimize the amount of effort required to apply the Tracing Method, (i.e., the effort in adding logging code). With the aim of generalizing the agent concepts, the set of attributes composing each agent concept is minimal. The attributes declared for each agent concept in Table 1 are needed to relate agent concepts with each other as illustrated in Fig. 1. For example, the attributes of a *goal* are the *agent* that wants to achieve the goal and a *name* for the goal. Other details about the goal (e.g., reward value) are not needed to

relate the *goal* to the *intention* created to achieve that goal. Of course, the goal's name must be distinct from names of non-equivalent goals. Other constituent attributes of an *intention* include references to the *beliefs* that were used in the process of creating the intention and *actions* to be performed as prescribed by the intention. Note that the values for the constituent attributes are set when an agent concept is observed, or logged (this is further discussed in Section 3).

By comparing the values of the attributes constituting each agent concept, relations between agent concepts during implementation execution can be automatically formed. For example, to relate an event to an action causing that event, the action's *postconditions* must be equivalent to the event's *preconditions*. Additionally, the *name* attribute of events, actions, goals, or intentions can be compared to the *subject* attribute of messages or beliefs to denote that a message or belief can be about an event, action, goal, or intentions. Section 4 demonstrates some application-specific relations in the UAV domain. For flexibility in the Tracer tool, the structure of each agent concept can be appended for application-specific relations that are not possible with the proposed structure (i.e., there is an implicit *user object* attribute for each agent concept).

Agent concepts and their relationship with each other establish the foundation for work in automated analysis of agent system implementations. For example, in Section 4, the Tracer tool analyzes observations about an agent (in terms of agent concepts) to explain agent actions.

3 Tracing Method and Tracer Tool

Due to the inherent unpredictability of autonomous agents in an uncertain environment and the possibility of emergent behavior, Jennings stresses a need for a better understanding of how agent interaction affects an individual agent's behavior [10]. The idea of the Tracing Method is to capture uncertain, dynamic run-time data (e.g., environmental events and communicated beliefs), to observe each agent's behavioral response, and to help explain this behavior. The Tracing Method can be used repeatedly throughout the software life-cycle from the first skeleton code to the final system. Using the Tracing Method shown in Fig. 2, interpretations (models or diagrams that represent the actual agent behavior in terms of agent concepts) are created using observations resulting from the implementation's execution. These interpretations can be the same models and diagrams that result from reverse engineering (e.g., flow control, component dependence, and class inheritance models or state-chart and process-flow diagrams), or the interpretations can be relational graphs linking observations together (as is the case in this paper).

The Tracing Method and Tracer tool is being developed for agent systems that are implemented in a procedural programming language (e.g., Java, C, and C++), but they can also be used in declarative agent-oriented programming languages (e.g., AF-APL [11] and Suna *et. al.*'s mobile agent language [12]) to visualize and clarify agent behavior in the system. Currently, the Tracer tool

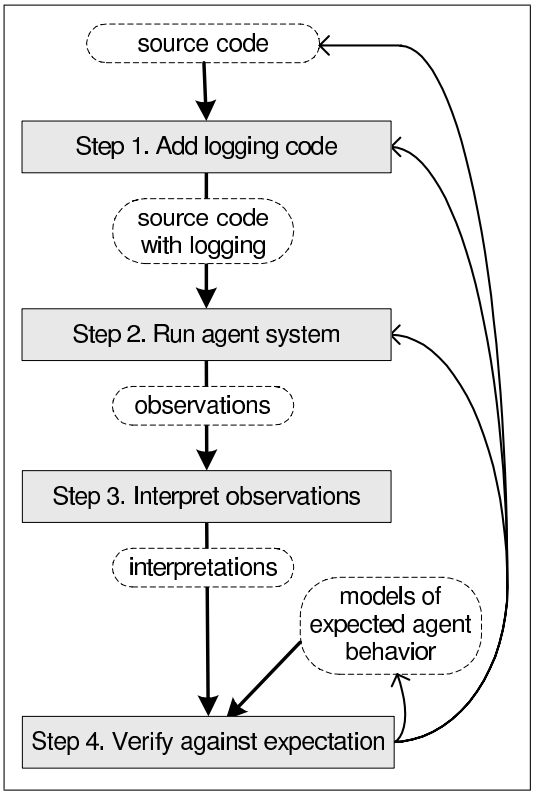

Fig. 2. Tracing Method process diagram

includes Tracing clients that allow Java and Lisp implementations to sends logs to the Tracing server.

The Tracing Method involves logging agent behavior during execution, transforming the log entries and run-time data into interpretations, and comparing those interpretations with the models of expected agent behavior. As a result, the agent concepts (e.g., beliefs about the current state of the environment) are instantiated with actual run-time data. By comparing the interpretations with the models of expected behavior, actual behavior can be verified against expected behavior. Expected behavior may be formally specified as models in design documents or informally understood and modeled by the developer(s). In either case, if there are inconsistencies between the interpretations and the expected behavior, the implementation or the expected behavior may need to be corrected. Currently, comparisons are manually performed because a formal specification for expected agent behavior has yet to be developed. However, explanations of particular observations (i.e., the observations that are inconsistent with expectations) can be generated upon user request. Details about the inconsistencies are presented to the user so that the implementation or expected behavior can be corrected. Since each observation can be traced back to a loca-

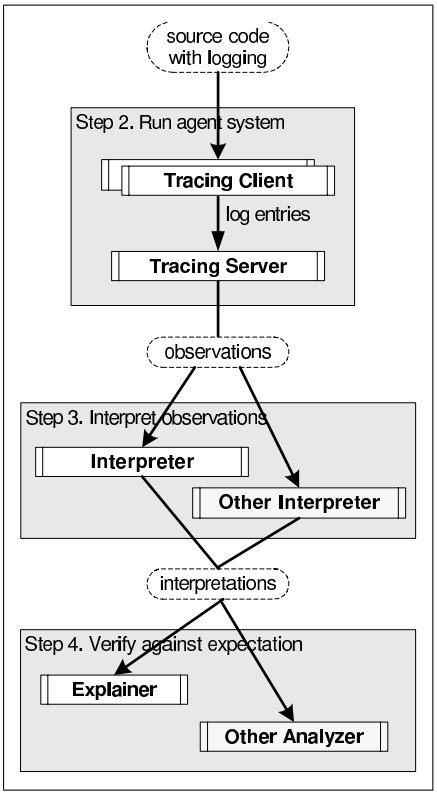

Fig. 3. Components of the Tracer tool

tion in the source code where the log entry was created (using a stack trace), correcting the inconsistency is facilitated.

Each step of the Tracing Method in Fig. 3 is described in the following subsections, accompanied by examples from the Tracer tool. To clarify which tasks in the Tracing Method have been automated and where additional features or tools can be integrated, Fig. 3 illustrates each component of the Tracer tool. The Tracer tool aims to automate the developer's task of analyzing run-time data, creating interpretations of actual agent behavior, and relating those interpretations to models of expected agent behavior. Essentially, the Tracer tool translates the procedural execution of the implementation (resulting in log entries) into declarative statements about what and when the agent believes, intends, and performs (called observations). To accommodate other analyses of the implementation, additional interpreters and analyzers can be added to the Tracer tool. The current products of the Tracer tool include behavioral and structural models (representing interpretations of the logged data based on order, duration, and other run-time attributes) and explanations of particular observations requested by the user. Since the interpretations are similar to design models, the interpretations can be compared to the original design models to ensure im-

portant aspects of the agent design have been traced. The comparison task can be performed by other automated tools that can analyze the observations or interpretations.

3.1 Step 1: Add Logging Code

The first step of the Tracing Method is to insert code that logs run-time data about agent concepts into the source code. Unlike traditional reverse engineering tools (e.g., Gen++ [13] and DESIRE [14]), this method does not analyze code in detail, thus, it is not dependent on any specific language. Instead, run-time data about the implementation's execution is acquired by explicitly logging the data that is desired. The logging code should be added at points in the source code where an agent concept is updated or occurs, such as when an agent (1) changes a *belief* that can affect decision-making, (2) decides about its *intention* (e.g., generates a plan of action), (3) modifies its *goal*, (4) performs an *action*, and (5) sends and receives a communication *message*, as well as (6) when an *event* occurs that can affect an agents behavior. To clarify this step, the demonstration in Section 4 identifies every agent concept that is logged for the UAV domain. Currently, this step is a manual process performed by the developer, tester, or end-user, assuming the implementation is organized and structured enough so that agent concepts can be identified in the source code.

Listing 1: Example logging code in Java for an agent

```
TraceLogger logger =
   Tracer_Client.getLogger("uav.Bot"+agentID);

public void handleNewScan(DetectedTarget target){
   logger.logBelief("Target"+target.getID(), target);
   // remaining implementation . . .
}
```

Listing 1 shows an example of logging code for a belief being updated in the source code. From Table 1, the constituent attributes of a belief are the agent holding the belief, the subject the belief is about, and the value of the subject. In Listing 1, the constituent attributes of the belief are `"uav.Bot"+agentID`, `"Target"+target.getID()`, and `target`, respectively.

In addition to inserting logging code for agent concepts, logging code can be inserted at the beginning and end of code segments to denote the agent's current state or the current task (or activity) the agent is performing. Such logging code denoting the agent's tasks provides more information and context for the logged agent concepts. For example, in the first five rows shown in Fig. 4, the `flyToTarget` action and `uavScan` event occur within the `internalHandleScans` task because they appear between its `START_TASK` and `STOP_TASK`. For more details about logging agent tasks, an earlier paper [15] describes how such con-

loggerName	ID	time	thread	class	method	message	realtime	taskName	type
uav.Bot15....	90	2	13	edu....	interna...		1071606965961	internalHandleScans	START_TASK
uav.Bot15....	92	2	12	edu....	make...	currentTarget 1	1071606965966	flyToTarget	ACTION
uav.Bot15.S...	93	2	13	edu....	handle...		1071606965966	uavScan	EVENT
uav.Bot15....	94	2	12	edu....	decide...	time=2	1071606965969	decideNextMove	STOP_TASK
uav.Bot15....	96	2	13	edu....	interna...		1071606965984	internalHandleScans	STOP_TASK
uav.Bot15....	97	2	13	edu....	interna...		1071606965986	internalUpdateCo...	START_TASK
uav.Bot15....	101	2	13	edu....	recalcu...	at 2	1071606965993	recalculatePrefere...	START_TASK
uav.Bot15....	103	2	13	edu....	recalcu...		1071606965999	newPrefs	BELIEF
uav.Bot15....	111	2	13	edu....	recalcu...		1071606966024	recalculatePrefere...	STOP_TASK
uav.Bot15....	113	2	13	edu....	addCo...		1071606966027	addCommitments	START_TASK
uav.Bot15....	115	2	13	edu....	addCo...	target 13	1071606966030	newCommitment	INTENTION
uav.Bot15....	148	2	13	edu....	addCo...		1071606966128	addCommitments	STOP_TASK
uav.Bot15....	149	2	13	edu....	interna...		1071606966130	internalUpdateCo...	STOP_TASK
uav.Bot15....	153	3	12	edu....	decide...	time=3	1071606966142	decideNextMove	START_TASK
uav.Bot15....	154	3	12	edu....	decide...	time=3	1071606966144	decideNextMove	STOP_TASK
uav.Bot15....	166	3	13	edu....	handle...	Bot16	1071606966194	otherPreferences	BELIEF
uav.Bot15....	174	3	13	edu....	handle...	Bot17	1071606966211	otherPreferences	BELIEF
uav.Bot15....	183	3	13	edu....	handle...	Bot16	1071606966228	otherCommitments	BELIEF
uav.Bot15....	192	3	13	edu....	handle...	Bot17	1071606966250	otherCommitments	BELIEF

Fig. 4. Log entries for agent `Bot15` in the UAV domain

textual logging code can result in state-chart and process-flow diagrams (other representations of software behavior), which are then verified.

Since only agent concepts are logged, this logging approach requires only high-level functional and structural knowledge of the implementation. If there is insufficient or erroneously-placed logging code, these errors will manifest themselves as specific inconsistencies between the interpretations and the models of expected behavior in the fourth step in Fig. 2. The developer can use the identified inconsistency to determine if logging code should be added or modified.

3.2 Step 2: Run Agent System

The second step of the Tracing Method is to execute the agent system so that the Tracer tool can collect run-time data, such as when and where the logging code was executed. A logging mechanism based on the Java Logging Framework [16] has been implemented. When the logging code executes, log entries are created from run-time data (e.g., what the agents believe and intend, what actions are being performed, and what events are occurring in their environment) and are sent by the Tracing Client to the Tracing Server locally or across a network (see Fig. 3).

Log entries for agent concepts and task activities are transformed into generic log entries so that they can all be handled by the same tools. For example, Fig. 4 displays all types of log entries on a single display. Figure 4 shows log entries as rows and the corresponding run-time data (e.g., timestamp and process id) as columns for an agent in the UAV domain. Each column is described as follows:

Loggername : context of the log entry identifying an agent or the simulator (e.g., `uav.Bot15` or `uav.Sim`) or a subcomponent within an agent (e.g., `uav.Bot15.planner`);
ID : unique identifier for the log entry;
Time : simulation time at which log entry was created;

Thread : execution thread id of the process in which the log entry occurred;
Class, Method : class and method in which the logging code executed (a full stack trace with source code line numbers is also available but not shown in the figure);
Message : additional free-form details about the log entry for human readability;
Realtime : real time at which log entry was created;
Taskname : name for observation or task;
Type : type of observation (e.g., action, event, belief) or task.

Most of the data (e.g., time, thread, class, method, and realtime) are acquired at run-time and appended to the log entry.

Due to the large amount of data, the log files need to be pre-processed and organized before they are analyzed. Log file utilities within the Tracing Server were created to sort, splice, and merge the log files so that log entries are organized correctly and interpretations accurately represent the implementation. To organize the log files, the loggerName can refer to an entire agent or a component within the agent. For each unique loggerName, there is a single log file. Thus, a thread that operates across several agent components may write to several log files. Since there may be several execution threads logging to a single file, the log file needs to be spliced into separate log files for each execution thread. For instance, in Fig. 4, the log entry with ID 94 was logged by thread 12, while most of the other entries were logged by thread 13.

3.3 Step 3: Interpret Observations

The third step is to construct interpretations that can be compared with models of expected behavior. There are several ways to automatically interpret the observations listed in Fig. 4, depending on what type of information is desired and what is being analyzed. For example, using the timestamp of observations, a state-transition diagram can be generated by one of the Tracer tool's interpreters (see [15] for an example). Additional interpreters can be added to produce other types of interpretations, including a time-plot of agent activities, data flow graphs, and message sequence charts. Each type of interpretation can be used to verify certain aspects of the agent system implementation as described in the next step. Given the desired interpretation type (in this case, it is a relational graph), the Tracer tool can generate the interpretation by processing the observations during run-time or after the execution has completed. Being able to monitor the agents during run-time offers an additional visualization of the running agent system.

To generate relational graphs used in the UAV demonstration, rules to relate agent concepts to each other are applied to the observations. The purpose for the rules is to form the relations illustrated in Fig. 1. For example, one rule states that if a message m from agent a contains the same information as belief b, then that message m is causally linked to that belief b. Another rule states that if intention p has belief b's subject in its list of belief subjects, then belief b affects

p. The resulting directed graph of these two rules implies that intention p was affected by belief b, which was a result of agent a sending message m.

To reduce the effort of applying the Tracing Method, the idea behind the interpreter is to automatically generate informative representations of agent behaviors from simple, reusable rules. Essentially, the rules compare the constituent attributes of agent concepts (e.g., name, subject, preconditions, and postconditions) in order to associate one agent concept with another. Because the agent concept structures were designed to be general, the rules can be reused in other agents with similar behaviors. Application-specific rules can be created by hand or generated by a pattern discovery mechanism. Future work will consider automatically generating the rules based on patterns in the list of observations.

3.4 Step 4: Verify Interpretations

The fourth step is to verify the interpretations against the models of expected agent behavior. There are several ways to verify the interpretations depending on the interpretation type. For example, state-transition diagrams and message sequence charts can be directly compared to expected behavior expressed as state-chart diagrams and communication protocol diagrams in design documents. Currently, due to the lack of a formal specification of agent behavior, verifying the interpretations is a manual step – a human must determine whether the interpretations are consistent with expectations. However, the Tracer tool's Explainer can assist the user in identifying the causes of unexpected behavior. Given the relational graph from Step 3, an explanation describing the observations relating to an agent action (or any observed agent concept) can be examined to ensure that an agent is performing the action for the right reasons. Section 4 demonstrates how an explanation is created.

To allow for other analyses of different interpretation types, additional analysis tools can be plugged into the Tracer tool (see Fig. 3). Possible analyses include checking safety and liveness properties about the execution trace, verifying that the agents are following communication protocols, and locating computational bottlenecks.

If the interpretations are inconsistent or seem erroneous with respect to expectations, (1) logging code may need to be added or corrected due to missing or misplaced observations, (2) the agent system may need to be executed multiple times to verify interpretation variations due to nondeterminism, (3) an implementation bug may need to be corrected, and/or (4) expected agent behavior may need to be updated. This is why there are arrows pointing from interpretations to previous steps or objects in Fig. 2. Since each high-level observation contains a low-level stack trace denoting where and in what context the logging code was executed in the source code, correcting inconsistencies is facilitated. The result of the Tracing Method is a set of verified interpretations of the implemented agents' behaviors in terms of agent concepts. As a side effect, the source code is sparsely documented with logging code that identifies important points in the code for understanding the implemented agents' behaviors.

4 Tracing the UAV Domain

To demonstrate the Tracing Method, the Tracer tool will be used to trace agents, each controlling a UAV (Unmanned Aerial Vehicle), in an implemented agent system for the UAV target-monitoring domain. The agents' overall objective is to ensure that all mobile targets are being scanned (by flying to the target's believed location and finding the target) with minimal time from when the target was last scanned. Each agent shares with other agents its own preferences about which targets it prefers to monitor. Each agent individually decides which targets it intends to scan (referred to as its committed targets) so that all targets are being scanned as frequently as possible using a Markov Decision Process (MDP) with a value function that considers distances from targets, targets' last scan times, and other agents' target preferences and commitments. Each agent's MDP model is updated as the agent receives new information about targets and other agents, thus affecting the agent's decision about which targets to monitor. Since there are a lot of factors for each agent to consider and the decision-making process occurs frequently, checking agent behavior would be facilitated by using the Tracing Method and Tracer tool.

The first step is to add logging code to the source code for each agent concept. To demonstrate that a low-level understanding of the implementation was not necessary, the person adding the logging code was not intimately familiar with the source code for the simulation or the agents and asked the developer only high-level questions concerning the abstractions from code to agent concepts. Note that the simulation was provided by Metron, Inc., as a contribution to the DARPA TASK project, the agents were programmed by a developer in our lab, and the logging code was added to the simulation and the agents by the author of this paper. The following lists specific instances for each agent concept in the UAV domain:

Message : messages about preferences, commitments, and scanned targets;
Belief : beliefs about an agent's own preferences, commitments, and scanned targets and, via communicated messages, beliefs about other agents' preferences, commitments, and scanned targets;
Desire : (static) minimize time between scans for all targets;
Intention : ordered list of committed targets;
Action : fly to target, spiral (to search for target), and stop;
Event : a target is scanned (if the agent is within range as determined by the simulation).

Once logging code was inserted for each of these agent concepts, the second step is to execute the implementation. The simulation was executed with fifteen targets (`0` to `14`) and three agents (`Bot15`, `Bot16`, and `Bot17`). During execution, the Tracing Clients send log entries (see Fig. 4) to the Tracing Server, which translates the log entries into observations. Table 2 partially lists the observations for agent `Bot15` in human readable format.

For each observation, the table shows the type of observation, the simulation time the log occurred, a unique identifier for the observation, and run-time data

Table 2. Partial list of observations for agent `Bot15`

Type	Time	ID	Run-time data
Belief	0	18	initTargets (3 7 2 0 14 6 1 10 5 13 9 11 4 8 12)
Belief	0	19	initTargetLocations ((151.26 203.46) (536.57 517.74) ... (55.01 77.03))
Action	1	31	stop
Event	1	33	uavScan ( )
Belief	1	42	myPreferences (1 13 6 9 5 10 2 3 12 8 4 11 0 14 7)
Intention	2	67	addCommitment (target 1), intention=(1)
Action	2	92	flyToTarget (1)
Event	2	93	uavScan ( )
Belief	2	103	myPreferences (1 13 6 9 5 10 2 3 12 8 4 11 0 14 7)
Intention	2	115	addCommitment (target 13), intention=(1 13)
Message	3	160	messageReceived from Bot16 sentAtTime 2 (about Bot16 preferences (10 1 6 5 8))
Belief	3	166	otherPreferences Bot16 (10 1 6 5 8)
Message	3	172	messageReceived from Bot17 sentAtTime 2 (about Bot17 preferences (1 6 5 10 9))
Belief	3	174	otherPreferences Bot17 (1 6 5 10 9)
Message	3	180	messageReceived from Bot16 sentAtTime 2 (about Bot16 commitments (10))
Belief	3	183	otherCommitments Bot16 (10)
Message	3	189	messageReceived from Bot17 sentAtTime 2 (about Bot17 commitments (1))
Belief	3	192	otherCommitments Bot17 (1)
Event	4	207	uavScan ( )
Event	4	228	uavScan ( )
Message	4	241	messageReceived from Bot17 sentAtTime 4 (about Bot17 preferences (9 3 6 2 5))
Belief	4	244	otherPreferences Bot17 (9 3 6 2 5)
Message	5	291	messageReceived from Bot16 sentAtTime 4 (about Bot16 preferences (8 10 11 3 2))
Belief	5	293	otherPreferences Bot16 (8 10 11 3 2)
Message	5	307	messageReceived from Bot17 sentAtTime 4 (about Bot17 commitments (1 9))
Belief	5	309	otherCommitments Bot17 (1 9)
Message	5	316	messageReceived from Bot16 sentAtTime 4 (about Bot16 commitments (10 8))
Belief	5	318	otherCommitments Bot16 (10 8)
Event	5	323	uavScan ( )
Event	7	353	uavScan ( )
Event	7	361	uavScan ( )
Event	8	393	uavScan ( )
Event	9	415	uavScan ( )
Event	10	440	uavScan ( )
Event	11	460	uavScan ( )
Event	12	484	uavScan (1)
Belief	13	510	scannedTarget (1)
Action	14	520	flyToTarget (13)
Belief	15	566	myPreferences (13 10 5 12 4 0 1 7 14 3 2)
Intention	15	594	addCommitment (target 10), intention=(13 10)
...	...	...	...

pertaining to the observation. The run-time data offers details such as what was believed, what action occurred, or what commitments were made by agent `Bot15`. Note that the observations are chronologically ordered by the simulation time and that the ID coincides with this temporal ordering. In this agent implementation, only messages received by `Bot15` are listed for conciseness, since messages sent by `Bot15` do not directly affect its own decisions. Also, no goal type observations are listed because all agents have a static goal of minimizing the time between scans for each target.

A glance through the table shows an unexpected (or at least undesired) behavior. The `uavScan` event, which signifies that an agent scans its current location for targets, was found to occur more than once per simulation timestep (e.g., observation 207 and 228 in timestep 4). Since the simulation only provides new scans once per timestep, the duplicate event is unnecessary. After consulting with the developer, this repetition occurs because the simulation's execution thread is running faster than the agent's execution thread. As a result, to make up for missed scans in previous timesteps, the agent performs multiple scans per timestep. For example, in Table 2, the agent does not scan in timestep 6, so the scan event is repeated in timestep 7. This undesired behavior does not adversely affect the overall performance of the agents with respect to its goal. However, this identified inefficiency may affect real-time performance as the number of agents increase or as the agents become slower than the simulation.

The third step is to interpret the observations by generating relational graphs. Rules, including the general rules mentioned in Section 3, were applied to the observations in order to create directed graphs. The general and application-specific rules that were used are listed below. The algorithm applies the appropriate rules to each new observation that appears and searches backward (temporally) to find the latest previous observation that satisfies the rule antecedent.

General rules:

- If belief b occurs after message m and their subjects and values are equivalent, then m affected b.
- If message m occurs after belief b and their subjects and values are equivalent, then b's occurrence caused m to be sent.
- If event e occurs after action a and e's precondition is equivalent to a's postcondition, then a caused e.
- If belief b occurs after event e and b's subject is equivalent to e's name and b's value is equivalent to e's postcondition, then e caused b.
- If an action a's name is contained in a previously observed intention i's action names, then i prescribed a.
- If intention i occurs after belief b and i's belief subjects includes b's subject, then belief b influenced the intention i.
- If message m occurs after an action, intention, or goal o whose name is equivalent to m's subject, then o influenced m.

Application-specific rules:

- If intention $i2$ occurs after intention $i1$, then $i1$ influenced $i2$.
- If a belief $b2$ occurs after the last `myPreferences` belief $b1$ and before a `myPreferences` belief $b3$, then $b2$ affects `myPreferences` belief $b3$.
- If a `flyToTarget` action a occurs after a `scannedTarget` belief b for different targets, then a is cause by b (i.e., the agent believes it has scanned its previous target and is pursuing its next target as prescribed by its intention).

More complicated application-specific rules can be created to relate more than two observations together, but for the general rules, straightforward rules are preferred for better reuse. The rules represent the background knowledge used

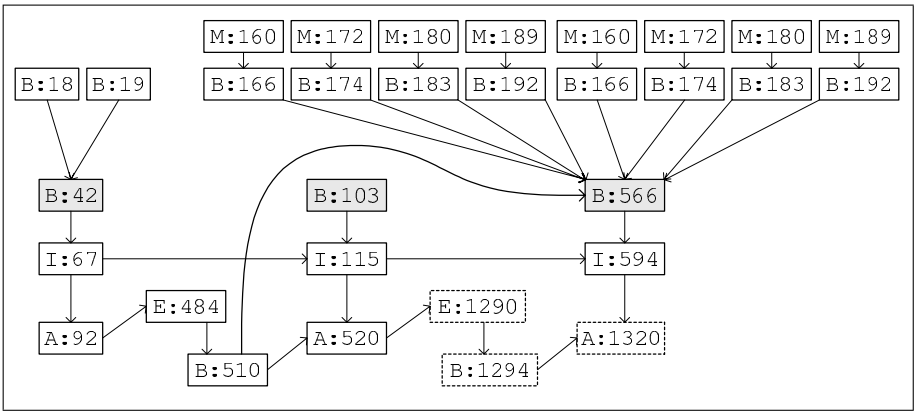

Fig. 5. Relational graph for agent `Bot15`

by the users to generate explanations. Such background knowledge needs to be represented in the Tracer tool in order to automate explanation generation. Currently, these rules are manually defined and given as input to the Tracer tool. For future work, the tool will discover patterns in the observations and suggest to the user rules that relate agent concepts to each other; thus, further automating the comprehension tasks.

Figure 5 illustrates the relational graph generated from the observations in Table 2. Each node in the graph is labeled with the first letter of the observation type (i.e., `B`=belief, `I`=intention, `A`=action, `E`=event, and `M`=message) and the unique id of the observation, so it can be referenced in Table 2. Each edge represents a source node causing (or influencing) the destination node. The *belief* nodes B:18 and B:19 show that agent `Bot15` processes initial data about the targets and their locations. Based on only those beliefs, `Bot15` creates its initial target preferences labeled B:42.

The shaded belief nodes represent `myPreferences` that have been calculated using `Bot15`'s current beliefs about the targets and other agents' target preferences and commitments. For the target preferences in B:42 and B:103, only the initial beliefs were used since the other agents have not yet communicated their preferences or commitments. However, in B:566, `Bot15` takes advantage of several beliefs (about other agents' preferences and commitments) that were created from communicated messages (i.e., M:160, M:172, M:180, etc.) as shown in Fig. 5 and detailed in Table 2.

The fourth step is to analyze actual agent behavior to insure agents are behaving as expected. In doing so, an end-user can gain a better understanding of what the agent is doing and why. A description of what the agent is doing is described below, followed by a description of how explanations are generated by the Tracer tool's Explainer.

Based on preferences in B:42, the agent makes a commitment represented by the *intention* node I:67 (i.e., `Bot15` adds commitment to scan `target 1`). Next, based on *intention* I:67, the agent performs an *action* A:92 (i.e., `Bot15`

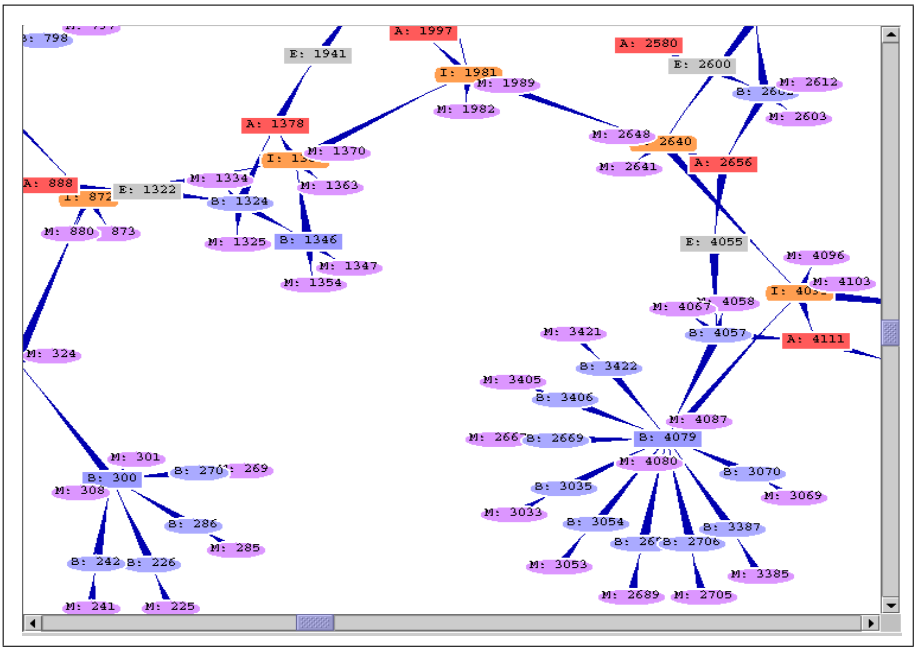

Fig. 6. Relational graph from Tracer tool

flies to `target 1`'s believed location). This series of observations can be easily followed in Table 2. Before *event* E:484 (i.e., `Bot15` scans `target 1`) occurs at timestep 12, the agent recalculates its preferences to create B:103 (i.e., `Bot15`'s new preferences are `( 1 13 6 9 ... )`) at timestep 2, which is not different from B:42 (as seen in Table 2) because there were no new beliefs to consider between B:42 and B:103. The reason the agent recalculates its preferences is to add its next target, which is target 13 as shown by *intention* I:115 in the table. Reasonably, the agent does not perform *action* A:520 (i.e., `Bot15` flies to target 13's believed location) until it believes B:510 (i.e., `Bot15` has scanned `target 1`) as shown in the graph.

A relational graph can present important information that may not be as obvious when presented as a list or table. The Tracer tool's Explainer can assist the user in analyzing the relational graph by generating explanations of specified observations. Given some observation to explain (e.g., an agent action), the explanation is generated by following the incoming edges of the node that represents the given observation. For example, the graph in Fig. 5 clearly shows that *action* A:520 is prescribed by the *intention* I:115, created before communication with other agents. In the table, however, since A:520 chronologically occurs after the communications with other agents, the action A:520 can be misconstrued to be influenced by those communications. Such information can save time and effort in trying to debug the implemented system. Figure 6 and Figure 7 show snapshots of the relational graph and an example explanation from the Tracer tool.

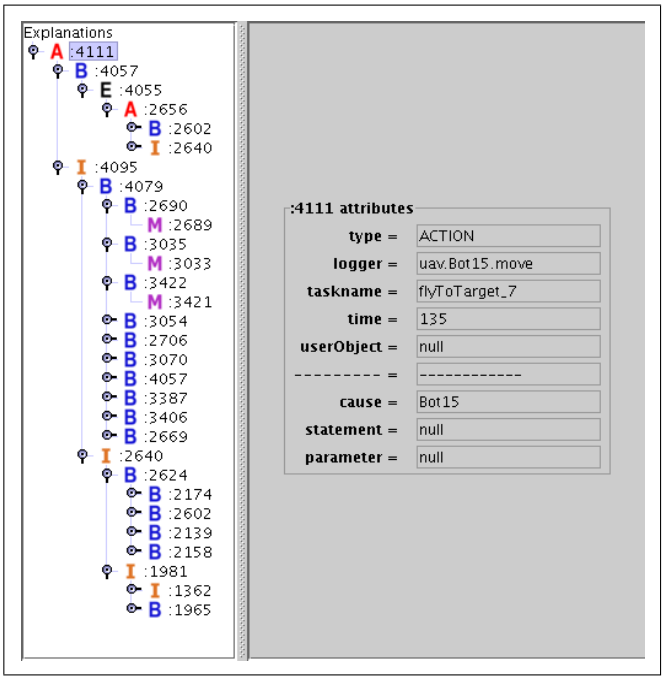

Fig. 7. Explanation of action A:4111 in Fig. 6

As demonstrated, the graph provides a quick way to understand (and ask questions about) the operations of the agent without having to understand the implementation in-depth. The graph can also help answer questions, such as "Why did `Bot15` intend I:594", by narrowing down the set of beliefs that influenced the agent to decide on I:594. Additionally, some patterns of behavior are more easily discovered by the human user in graph form. For example, in Fig. 5, the nodes with dotted outline represent subsequent observations that relate *action* A:520 and *intention* I:594 (namely, the post-conditions of A:520 must be true before the next *action* A:1320 in I:594 can be performed), completing the pattern of behavior. More high-level behavioral patterns can be seen in Fig. 8, where there are clusters of belief and message nodes surrounding intention nodes and the clusters are connected by action nodes. Not surprisingly, such a pattern resembles the classical sense-reason-act cycle used in artificial intelligence.

Given patterns of behavior for an agent, anomalous behavior can be quickly identified as subgraphs that are not similar to the pattern. For example, Fig. 9 shows anomalies (visualized as dangling nodes on the right side of the figure) in behavior for an earlier version of the agent. Such anomalies can identify possible bugs in the system or unexpected changes in agent behavior. Given this information, the user can investigate the cause of the bug at that anomalous observation or add a new rule (to the background knowledge) to account for the new behavior. A future feature in the Tracer tool will be automated graph pattern discovery and anomalous behavior detection.

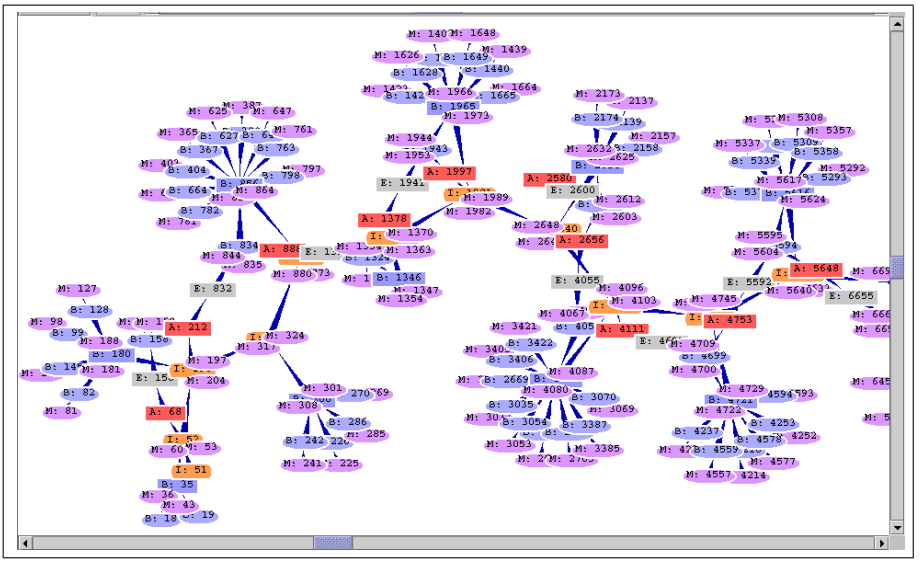

Fig. 8. Patterns in relational graph

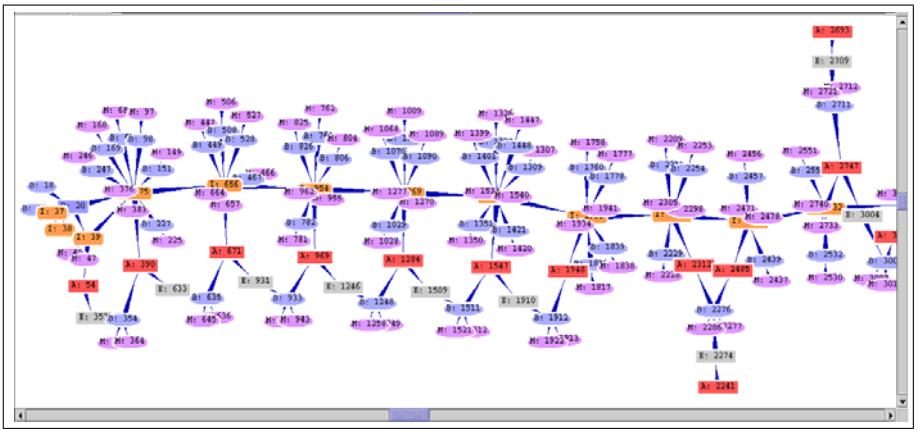

Fig. 9. Anomalies (on right-side) in relational graph for a different agent

5 Related Work

This section discusses the limitations of two popular approaches for verifying software behavior (model-checking and reverse engineering) when applied to agent-based implementations. This research addresses the limitations in using these verification approaches.

Model-checking performs a thorough search through a high-level model of an implementation to find undesired behaviors and to ensure desired behaviors as specified by the user. To verify agent behaviors using model-checking, (1)

observed agent behavior (as understood by the user) and properties to be verified are translated into a format suitable for a model-checking tool, (2) the model is checked by the tool (provided the state space size is manageable), and (3) results from model-checking are interpreted and related back to the actual system (i.e., mapping a property violation back to the implementation).

For software developers who are not experts in model-checking, the translation and interpretation steps may be particularly challenging, and even for a seemingly trivial system, the large state space may be unmanageable. To reduce the learning curve associated with model-checking, software engineering researchers have focused on tools and methods to enable model-checking of high-level models (e.g., Petri-nets [17], UML diagrams [18], and architecture representations [19]). While these approaches have helped reduce the translation and interpretation barriers, they do not leverage software models that incorporate agent-related abstractions (i.e., agent concepts) and do not facilitate translating actual agent behavior from the implementation to models to be checked.

Bordini *et. al.* applied model-checking to reactive-planning agents implemented in the BDI logic programming language AgentSpeak by translating the implementation into a finite-state model that can be verified using the Spin model-checker [20]. Though promising for agent systems implemented in logic or for applications requiring formal verification, the use of model-checking in the numerous procedural (infinite-state) implementations requires effort in abstracting and translating the source code into a checkable model and may not be practical.

Edmunds points out the insufficiency of formal methods and the need for an experimental approach for understanding multi-agent systems [21]. This research offers the Tracing Method as an experimental approach to analyzing agent behavior, unlike model-checking. First, to minimize translation errors due to misunderstanding the implementation, agent behavior is constructed from the log of actual agent beliefs, intentions, and actions. Additionally, the user is only required to know where agent concepts are updated in the source code so that logging code can be added. Second, to avoid the large state space representing the aggregate behavior of an agent-based system, the Tracing Method analyzes agent behavior scoped by the set of scenarios through which the implementation is executed. The scope of scenarios for tracing can be iteratively modified as the typical development effort is iterative. Third, relating the analysis results back to the implementation is facilitated because each observation of agent behavior can be traced to an exact location in the implementation. There are a number of possible points of failure, but future work hopes to provide guidelines or (fully or partially) automate some of the steps in the Tracing Method.

Traditional software reverse engineering tools (e.g., Gen++ [13] and DESIRE [14]) analyze the implementation at the source code level and produce models of the implementation (e.g., flow control, component dependence, and class inheritance models) that are detailed representations of *what* is happening in the implementation. With the increase in complexity of agent systems, it becomes very difficult to get a comprehensive system view of the implementation using

traditional tools due to the number of software components and low-level interactions involved. Without having to analyze the source code in detail, the Tracer tool automates the analysis of *what* is happening and helps to explain *why* such behavior is happening using high-level agent concepts (e.g., beliefs, goals, and communication messages). Since an agent system is conceptualized and designed using agent concepts, comprehension and verification of agent behaviors in the implemented system for debugging, testing, and maintenance should also use agent concepts.

6 Summary

As agent systems become more complex and sophisticated, there is a growing need for agent-oriented methods and tools to debug, maintain, and test agent software. This paper presents the Tracing Method and accompanying Tracer tool to help (1) verify actual agent behavior in the implemented system against expected (or designed) agent behavior and (2) understand the implemented agent system in terms of the same agent concepts used in the software design. Agent concepts are used to describe agent structure (e.g., an agent encapsulates localized *beliefs*, *goals*, and *intentions*) and behavior (e.g., an agent performs an *action* when it believes the *event* occurred). The Tracing Method captures dynamic run-time data during implementation execution, interprets the data as observations of actual agent behavior, and analyzes those interpretations.

The Tracer tool facilitates the ability (1) to determine if agent design specifications are correctly implemented and guide debugging efforts and (2) to examine and discover motivations, such as beliefs, intentions, and communicated messages, for agent behaviors. As demonstrated in the target-monitoring UAV domain, the Tracer tool assists in gaining insight into agent behavior by automating the process of generating interpretations of the implementation execution and presenting observed agent behavior in terms of agent concepts. In addition, the Tracing Method establishes general structures for agent concepts that can be used in most agent system implementations, thus, moving away from ad hoc debugging techniques.

This research proposes a method and tool to create models of agent behavior that not only describe *what* is occurring in the implementation but also *why* a respective agent behavior occurred (e.g., agent X took action a because of belief b). To enable such explanations, the Tracing Method requires only a high-level understanding of where agent concepts are modified in the implementation. In this regard, the behavior of agents in unfamiliar agent systems can be quickly understood. Overall, the Tracing Method and Tracer tool sheds light on how agents actually behave and how agent behavior in the implementation can be improved.

Acknowledgements

This research was funded in part by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF,

under agreement number F30602-00-2-0588. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed on implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References

1. Wooldridge, M., Jennings, N.R., Kinny, D.: The GAIA methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **3** (2000) 285–312
2. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. International Journal of Software Engineering and Knowledge Engineering **11** (2001) 231–258 World Scientific Publishing Company.
3. Dignum, V., Vazquez-Salceda, J., Dignum, F.: OMNI: Introducing social structure, norms and ontologies into agent organizations. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 91–102
4. JADE: Java agent development framework. http://sharon.cselt.it/projects/jade/ (2000)
5. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: ZEUS: A toolkit for building distributed multi-agent systems. In Etzioni, O., Muller, J.P., Bradshaw, J.M., eds.: Third International Conference on Autonomous Agents, Seattle, WA, ACM Press (1999) 360–361
6. Poslad, S., Buckle, P., Hadingham, R.: The FIPA-OS agent platform: Open source for open standards. In: Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchestor, UK (2000) 355–368
7. Gasser, L., Braganza, C., Herman, N.: MACE: A flexible testbed for distributed AI research. In Huhns, M.N., ed.: Distributed Artificial Intelligence. Morgan Kaufmann, San Mateo, CA (1987) 119–152
8. Doi, T., Yoshioka, N., Tahara, Y., Honiden, S.: Bridging the gap between AUML and implementation using FOPL. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 69–78
9. Braubach, L., Pokahr, A., Lamersdorf, W., Moldt, D.: Goal representation for BDI agent systems. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 9–20
10. Jennings, N.R.: Agent-oriented software engineering. In: Lecture Notes in Computer Science: Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99). Volume 1647. (1999) 1–7
11. Ross, R., Collier, R., O'Hare, G.M.: AF-APL – bridging principles & practice in agent oriented languages. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 21–33

12. Suna, A., Fallah-Seghrouchni, A.E.: A mobile agents platform: Architecture, mobility and security elements. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 57–66
13. Devanbu, P.T.: GENOA- a customizable, language- and front-end independent code analyzer. In: Fourteenth International Conference on Software Engineering, Melbourne, Australia (1992) 307–319
14. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: Program understanding and the concept assignment problem. Communications of the ACM **37** (1994) 72–83
15. Barber, K.S., Lam, D.: Enabling abductive reasoning for agent software comprehension. In: 18th International Joint Conference on Artificial Intelligence Workshop on Agents and Automated Reasoning, Acapulco, Mexico (2003) 7–13
16. Sun Microsystems, Inc.: Java Logging API. http://java.sun.com/j2se/1.4/docs/guide/util/logging (2002)
17. Grahlmann, B., Pohl, C.: Profiting from SPIN in PEP. In: SPIN '98 Workshop. (1998)
18. Bose, P.: Automated translation of uml models of architectures for verification and simulation using SPIN. In: IEEE International Conference on Automated Software Engineering. (1999) 102–109
19. Barber, K.S., Graser, T.J., Holt, J.: Providing early feedback in the development cycle through automated application of model checking to software architectures. In: 16th International Conference on Automated Software Engineering, San Diego, CA (2001) 341–345
20. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In Rosenschein, J.S., Sandholm, T., Michael, W., Yokoo, M., eds.: Second International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia, ACM Press: New York (2003) 409–416
21. Edmonds, B., Bryson, J.: The insufficiency of formal design methods. Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (2004) 938–946