Comprehending Agent Software

D. N. Lam and K. S. Barber
The University of Texas at Austin
Laboratory for Intelligent Processes and Systems
1 University Station C0803, Austin, TX 78712-0240
{dnlam, barber}@lips.utexas.edu

ABSTRACT

Software comprehension (understanding software structure and behavior) is essential for developing, maintaining, and improving software. This is particularly true of agent-based systems, in which the actions of autonomous agents are affected by numerous factors, such as events in a dynamic environment, local uncertain beliefs, and intentions of other agents. Existing comprehension tools are not suited to such large, concurrent software and do not leverage concepts of the agent-oriented paradigm to aid the user in understanding the software's behavior. To address the software comprehension of agent-based systems, this research proposes a method and accompanying tool that automates some of the manual tasks performed by the human user during software comprehension, such as explanation generation and knowledge verification.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: *Debugging aids, diagnostics, monitors, testing tools, tracing.*

General Terms

Documentation, Design, Experimentation, Human Factors, Verification.

Keywords

Software comprehension, agent-oriented software engineering, reverse engineering, debugging, maintenance, tracer.

1. INTRODUCTION

Software comprehension is crucial for the development, maintenance (e.g., debugging, testing, and improving), and redesign of software. Software comprehension (a.k.a. reverse engineering) is performed by the human user to understand the structure and behavior of the software. The current suite of tools to help a user comprehend software includes (1) those that gather, summarize, and enable the user to browse through the source code (static analysis) and (2) those that generate behavioral diagrams from captured execution traces (dynamic analysis). A commonality among these comprehension tools is that they capture artifacts about the software (e.g., static dependencies among objects and events that occur during simulation) and leave it up to the user to interpret and reason about the software's behavior.

This research aims to reduce the amount of manual effort performed by the human user during software comprehension by developing a method (1) to reason about observed software behavior at the design abstraction level and (2) to explain why given observations occurred. The proposed method is particularly important for agent-based software, where agents (autonomous software entities) take actions based on localized (possibly erroneous) beliefs about their environment. Comprehension of such sophisticated software systems is complicated by the number of agents and factors (e.g., beliefs, environmental events, other agents' intentions and beliefs) that affect each agent's actions. This research aims to help designers who want to improve agent or system behavior; developers who need to debug and verify agent behavior; and end-users who want to understand (at the design level) why agents performed certain actions.

The approach of this research is to imitate what a human user does in software comprehension. Specifically, this paper presents the Tracing Method to semi-automate the process of building and refining the user's understanding of the system (called knowledge base **K**) and using **K** to explain and verify agent behavior in the implemented system. As a result of the Tracing Method, an explicit representation **K** is used to express what the user knows about the software system. For agent-based systems, **K** is expressed in terms of *agent concepts* (i.e., beliefs, goals, intentions, actions, environmental events, and inter-agent messages), which are typical concepts used in agent-oriented software designs. The Tracer Tool has been implemented to assist the user through the proposed comprehension method. An experiment of the tool being applied to two agent systems is described.

Taking the viewpoint that a person's comprehension of a subject can be indirectly measured by how much the person can explain about the subject, the Tracer Tool is able to generate explanations (i.e., reasons for why an observation occurred), given **K** and observations captured from the system's execution. These explanations can be used by designers, developers, and end-users to elucidate, debug, and build trust in the agent system's behavior.

Section 2 reviews existing work related to software comprehension. Section 3 describes the Tracing Method and accompanying Tracer Tool, which is demonstrated in Section 4. Section 5 summarizes this research and its contribution to agent-oriented software engineering.

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.

2. RELATED WORK

Software maintenance will always be needed as long as the design or implementation needs to be modified. Software bugs inevitably arise during the development of the implementation due to programming error, design error, or communication error. To fix bugs or to ensure that the implementation is free (for the most part) of bugs, the designer or developer must have an adequate understanding of the software. Such an understanding can be acquired by reading design documents, examining source code, and analyzing execution traces of the system.

Section 2.1 describes several software comprehension tools (often categorized as reverse engineering tools) to assist the user in analyzing and perusing the source code. Section 2.2 discusses the use of model-checking, which focuses on testing and verification of software behavior, as another approach for software comprehension since the model being checked can be thought of as a formal design specification. That design specification abstracts away many of the details of the implementation, and the entire system can be viewed in a concise representation, unlike the overwhelming abundance of data gathered by traditional comprehension tools. This research takes the model-checking approach for representing software behavior and the reverse engineering approach for verifying the model against the actual implementation.

Though useful, these techniques have their limitations (as described below) and can be improved for the purpose of comprehending agent-based software. For agent software in particular, the difficulties in comprehension are exacerbated by characteristics of individual agents and of the system itself. Agents are distributed, localized software entities that are typically characterized as being rational, proactive, adaptable, social, and able to deal with uncertainty. A software system with many agents, each with its own goals, resources, and constraints, making decisions, interacting, and acting on its own, makes software comprehension a challenging activity. An end-user may be hesitant to trust such autonomous agents without knowing how the agents make their decisions. For comprehension, the user must consider the system's distribution, concurrency, domain uncertainty, and non-determinism [6]. This paper proposes a method and tool that builds on the ideas from existing approaches and extends the state-of-the-art to better assist the human user (of various skill levels) in comprehending agent-based software.

2.1 Comprehension (or RE) Tools

Comprehension tools are synonymous with reverse engineering (RE) tools in that both aim to offer the user a better understanding of the structure and relationship among software components of the system. In addition to identifying components and their dependencies, RE involves creating abstractions of the system design [3]. For example, Rigi and PBS extract structural data from the source code, analyze its structure, and visualize the software architecture for the user to browse [1; 5]. RE tools tend to produce a large amount of data for the user to interpret. To deal with this, SoftSpec allows users to query a relational database of automatically gathered information about the software architecture [2]. Alternatively, using a graph-oriented approach, the GUPRO (Generic Understanding of Programs) toolset transforms source code into graphs according to a defined concept model [11]. In a case study by Lange, Winter, and Koblenz comparing graph-oriented and database approaches, GUPRO's graph-oriented approach offered a more efficient way to analyze and search through the large amounts of data extracted from the source code of a large stock-trading application [13]. This research employs graphs to visualize the system's behavior.

Additionally, since these tools parse the source code directly, they are limited to programming languages that are parsable by the tool (i.e., tools must be extended for additional languages). For real-world applications, RE tools need to support software that use more than one programming language [19]. Since the Tracer Tool does not parse code, the tool can operate with any software system implemented in practically any mix of languages.

For users familiar with design patterns, Schauer and Keller have a tool to extract, abstract, and visualize design patterns; however, the tool can only identify pre-defined patterns [18]. This research also uses abstractions to help extract familiar design concepts from the implementation, thus, aiding the comprehension of the overall system behavior, independent of language-dependent implementation constructs.

In addition to static analysis of the source code, some RE tools analyze the dynamic aspects of the system in order to understand software behavior (i.e., actions and their motivations and consequences) in varying execution scenarios. For example, in SCED, detailed event trace information (e.g., method invocation) is used to create a set of sequence and state diagrams [10]. Other tools, such as Hindsight and Lemma, can create flow charts and control-flow diagrams [7; 15]. Moreover, results of dynamic analysis can also be represented as use cases and recurring behavioral patterns, which brings the implementation closer to the design [9; 14]. Since dynamic agent behavior is central to comprehending agent systems, the Tracer Tool focuses on elucidating reasons for agent behavior.

Due to the large amount of run-time data generated (even for small systems), dynamic analysis tools must also manage and abstract the extracted data [20]. To limit the amount of data that the user must search through, the user often must select specific components to be analyzed by the RE tool. Thus, it is usually assumed that the user has some understanding of the static structure of the software [20]. This is also assumed for the Tracing Method described in this paper. This research deals with the large amount of data by automating data interpretation for the user (described in Section 3). As an alternative to dealing with the large amount of data, Poutakidis et al. focuses only on the verification of agent interaction protocols [16]. The Tracing Method allows the user to analyze any agent concepts in the agent design, including interaction protocols.

2.2 Model-checking

Reverse engineering is usually performed as a supporting activity for other software engineering tasks after a working implementation has been created. Such tasks include re-engineering to add or extend features; maintenance to correct or improve the software, reusing of components in other software; and assessing if the software satisfies certain requirements. Alternatively, system behaviors (as expressed by a formal model of the design) can be checked before any development takes place. Model-checking is used to formally verify behavioral properties (e.g., deadlocks and assertion failures) of software systems, but the ideas behind model-checking can be used for software comprehension. Model-checking using tools such as

Spin [8] and Bogor [17] can be very useful in guaranteeing dynamic properties of the implementation, provided the model being checked accurately represents the implementation. Unfortunately, model-checking can only be performed on models whose state space is small enough to be exhaustively checked in a reasonable amount of time. For most agent-based systems (which are written in procedural languages such as Java or C), this means that a simpler model of the implementations must be created manually, which suggests that the model is representative of what the user believes about the implementation and may not be representative of the actual implementation.

In addition, as the implementation changes, the model must be updated as appropriate. In practice, keeping an accurate model that reflects the implementation is very difficult, particularly if the model extraction is done manually. This problem is commonly known as the translation gap. There are a few tools, such as Java PathFinder (JPF) [21], that extract models from the source code, but these tools are not yet mature, are language-dependent, and do not scale well (JPF is suited to programs with approximately 10,000 lines of code).

Due to these and other issues with applying model-checking in practice, Edmonds emphasizes the need for empirical analysis of agent behavior, such as scenario and field testing [4]. This research borrows the approach of modeling what the user understands from model-checking. To extend this approach, the Tracing Method also verifies the model (called **K**) against the actual implementation using a RE approach and maintains the model so that it accurately represents the implementation.

3. TRACING METHOD & TRACER TOOL

This research combines and extends ideas of empirical analysis from reverse engineering and abstraction from model-checking. The result is a high-level, more scalable, practical, semi-automated solution for agent software comprehension. Comprehension is performed at the system-level using high-level *agent concepts* (explained in Section 3.1) that are familiar to designers, developers, and end-users. Hence, all activities in the Tracing Method (described in Section 3.3) operate in the realm of agent concepts, rather than detailed execution traces and programming data structures of traditional reverse engineering. By abstracting implementation details as agent concepts, scalability is dependent on the number of agent concepts, rather than on code size or state-space complexity. The solution is practical in that it can be applied to any programming language (that can log data) and that the learning curve is low, unlike many RE and model-checking tools. The Tracer Tool assists the user by automating as much of the Tracing Method as possible, such as organizing the logged data, creating interpretations of logged data, verifying the knowledge base **K**, and generating explanations (described in Section 3.4).

3.1 Agent concepts

Agent concepts denote constructs (e.g., goals and beliefs) used in agent-based systems and are abstracted away from low-level implementation constructs. Since agent concepts are used in software designs 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), agent concepts should be leveraged for comprehending

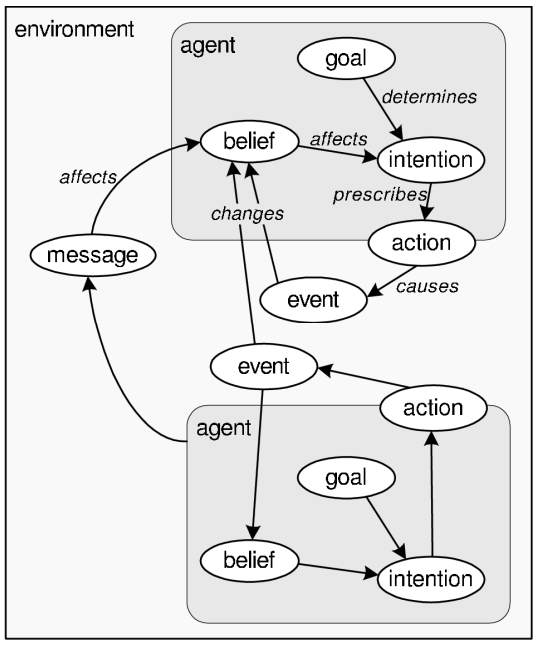

Figure 1. Agent concepts and some typical relationships.

the software. If the same concepts and models are used in forward and reverse engineering, tools would be able to better support re-engineering, round-trip engineering, maintenance, and reuse [20]. In this research, agent concepts are used to leverage the user's intuitive knowledge of general agent-based systems to comprehend the implementation.

The current 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. Instead, agent concepts can be better defined by their relationship with each other (Figure 1 illustrates some typical relationships).

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 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 for 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 being performed.

3.2 Knowledge Base

A knowledge base (**K**) is used to explicitly represent what the user comprehends about the software; specifically, **K** describes the expected agent behavior in the system. **K** is represented as a concept graph where nodes denote agent concepts and directed edges denote relations between agent concepts. Each node has a set of attributes, some of which are static while others are assigned values during runtime (agent concepts are instantiated as *observations* during runtime). The relations represent causal or temporal links between agent concepts. For example, beliefs can "cause" (loosely defined) or influence a certain intention to be created, or an event occurs after an action is performed. *Incoming relations* for an agent concept refer to relations that point to that concept. Continuing the example, the aforementioned event would have an incoming relation originating from the action. With an explicit representation of the user's knowledge of the system (**K**), that knowledge can be verified against the actual behavior of the implementation.

The representation for **K** is designed to be general enough to allow for various research ideas and agent models. Each agent concept has a user-defined name and a set of attributes (e.g., agent names, preconditions, postconditions, and associated parameters). When the instrumented implementation is run, the agent concepts are recorded as *observations* that are instantiated and populated with run-time data for each attribute. Users can define additional attributes to capture application-specific details about the agent concept. These attributes are critical for automation, such as suggesting possible relations between agent concepts, by the Tracer Tool. Details about the process of building **K** with the aid of the Tracer Tool are described in Section 3.3.

3.3 Tracing Method

A human's learning process centers around building and refining their knowledge, in this case, represented as **K**. Learning is a cycle consisting of (1) *hypothesizing* about concepts (i.e., adding agent concepts and relations to **K**) based on collected data about system behavior, (2) *testing* the hypothesis against the actual system (i.e., verifying **K** against the actual implementation), and (3) *interpreting* the observations collected from testing. The Tracing Method imitates this cycle and the Tracer Tool automates some of the manual tasks involved in each of these steps.

Initially, the knowledge base **K** is empty, so the user begins by defining agent concepts of interest to be recorded during execution. The agent concepts of interest may be derived from design documents, informal communications, experience with the agent-based systems, etc. For example, the specification of a communication protocol identifies messages sent and received by the agent. Note that not all agent concepts must be identified – **K** is incrementally refined and gradually increases in size as the user adds relevant agent concepts and learned relations.

When the implementation runs, agent concepts are instantiated with run-time data (e.g., call stack, simulation timestamp, and values for attributes) and are logged as *observations* (described in Step 2). From these observations, an interpretation is created using **K** as described in Step 3. Initially, when **K** has no relations defined, the interpretation is simply a set of unconnected observations.

Step 1 - Hypothesizing

Given an interpretation, the Tracer Tool can suggest possible relations for observations with no incoming relation. The algorithm compares attribute values between observations (hence the importance of agent concept attributes) and if attribute values "match", then a generalized relation is suggested (e.g., intention *i* causes action *a* because *a* is listed within *i*). Experiments showing the performance of the relation-suggesting algorithm are described in Section 4. With the Tracer Tool, the user can efficiently investigate the suggested relation, verify that the relation is meaningful, and approve the relation to be added to **K**. The user can also add relations that could not be automatically found.

In addition to adding relations, the user can add and modify agent concepts in **K** to gather more relevant run-time data that can help understand the agent's or system's behavior. In so doing, logging code will need to be added or modified in the source code. Currently, there is no way to automatically suggest that agent concepts be added. With each iteration through this cycle, the increase in comprehension is apparent as agent concepts and relations are added to **K**.

Step 2 - Testing

Testing is needed to insure that the agent concepts and relations defined in **K** accurately reflect the actual implementation. Testing involves instrumenting the source code with simple logging code to denote where agent concepts occur or change. For each agent concept in **K**, the Tracer Tool generates single-line logging code to be inserted into the source code by the user. It is assumed that the user has general *structural* knowledge about the source code to be able to identify where agent concepts occur. Given this assumption, hypothesis testing can be done by the designer, developer, or end-user. The set of agent concepts that should be logged is usually selected from (but not limited to) the agent design.

When the implementation runs, the logging code is executed and observations are recorded. The Tracer Tool uses a client-server model to collect log data from the running agent system, which may be distributed across several machines. Since observations may not arrive at the server in order, the Tracer Tool organizes and sorts the observations to be interpreted in Step 1.

Though the observations are at a high abstraction level, enough run-time details are recorded to locate the exact place in the source code that created the observation. Note that since only agent concepts are logged, the amount of data produced is reduced and scalability is improved. The result of this step is a set of observations (i.e., logged run-time data) about the agent concepts specified in **K**.

To test the robustness of **K**, the agent system must be run through a set of scenarios that adequately covers the execution space of the agent systems. Obviously, an exhaustive scenario set is impossible and impractical. This is a limitation of reverse engineering that is difficult to overcome. Another limitation with this approach is that for agents that dramatically change their

behavior (e.g., agents based on genetic algorithms or agents that dynamically learn new actions), there is usually no expected behavior to be modeled by **K**.

Step 3 – Interpreting

Traditionally, observing and relating observations together is performed manually by the user. Using **K**, the Tracer Tool can automatically collect and interpret the observations for the user by linking appropriate observations together, building a relational graph. Based on relations defined in **K**, the Tracer Tool compares observations and their attributes to determine if the observations are related. If they are related, a directed edge is created between the observations. For example, if the "postcondition" attribute of an agent's action has the value "near target" and the "precondition" attribute of an environmental event also has the value "near target", then a directed edge is created from the action to the event.

This is repeated for each observation, and the resulting interpretation is visualized as a relational graph (seen in Figure 2) whose structure is similar to **K**. In some sense, **K** is being used as a template to create the interpretation. The difference is that the interpretation represents actual behavior while **K** represents expected behavior. If the interpretation is inconsistent with **K**, **K** may need to be modified, just as the user's knowledge must be modified if the implementation did not behaved as expected. Sources of inconsistencies are enumerated in Section 3.4.

The process of interpreting can be performed during run-time or after the agent system has finished execution. Interpreting is similar to design recovery, a subfield within reverse engineering. Since many types of interpretations can be created from the implementation, it is possible to add additional interpreters to the Tracer Tool to aid in comprehension. For example, another interpreter can create state-transition diagrams for processes within an agent based on the same set of observations.

Repeat: This cycle repeats until **K** is complete with respect to a set of implementation executions. **K** is complete if for all interpretations for a given set of implementation's executions, every observation has an incoming relation (except for initial observations and observations of exogenous events that occur in the environment). In other words, given any observation *o*, **K** can be used to trace back from that observation to some other observation that caused *o*. The Tracer Tool will suggest possible relations for observations that do not yet have an incoming relation.

Though **K** may be complete, the knowledge in **K** may not be sufficient to get an adequate grasp of the system. For example, a trivial **K** can simply contain three agent concepts (e.g., belief, action, and event), one concept leading to the next. The issue is how to determine if all relevant agent concepts have been included in **K**. This depends on the level of detail desired, but as a guideline, all agent concepts that are present in an agent-oriented software design should be included in **K**. The defined set of agent concepts shown in Figure 1 helps to keep **K** at an abstraction level consistent with the design.

Note that all agent concepts and relations are added to **K** by the user. Thus, **K** contains only information that the user understands. By explicitly representing what the user understands in **K**, some additional tasks can be automated. For example, explanations of observed behavior can be automatically generated and anomalous (or unexpected) behavior can be automatically identified. The next section describes automated explanation generation.

3.4 Automated Explanation Generation

In this process of building, verifying, and refining **K**, anomalous behavior may be discovered. Anomalous behavior manifests itself as inconsistencies between the interpretation and the implementation. Inconsistencies appear as observations (or nodes) without incoming relations or relations between observations that do not semantically make sense. The source of

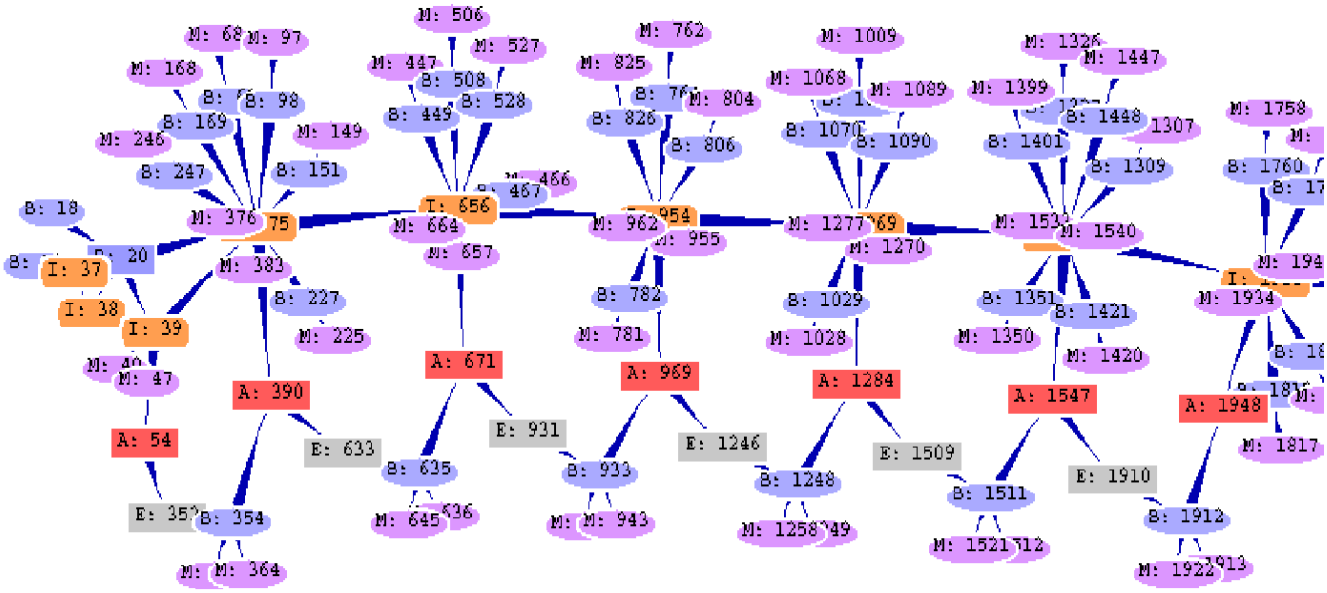

Figure 2. Example relational graph interpretation of a single agent.

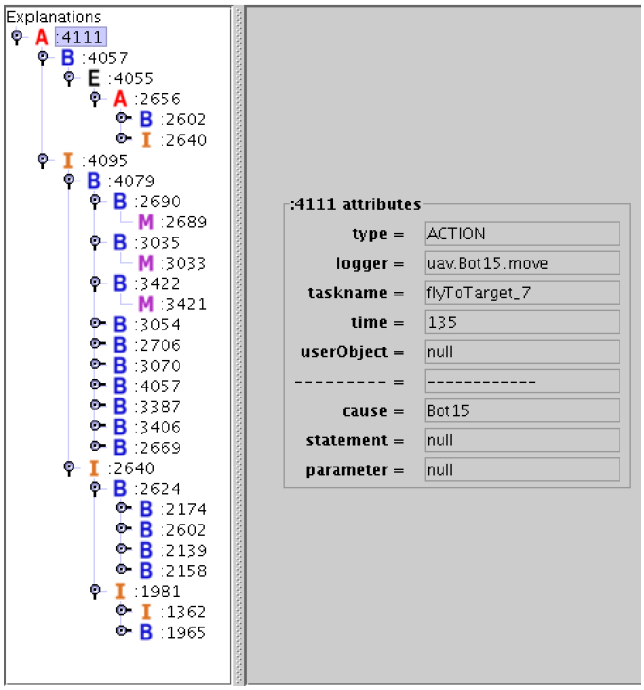

Figure 4. Explanation of agent action A:4111.

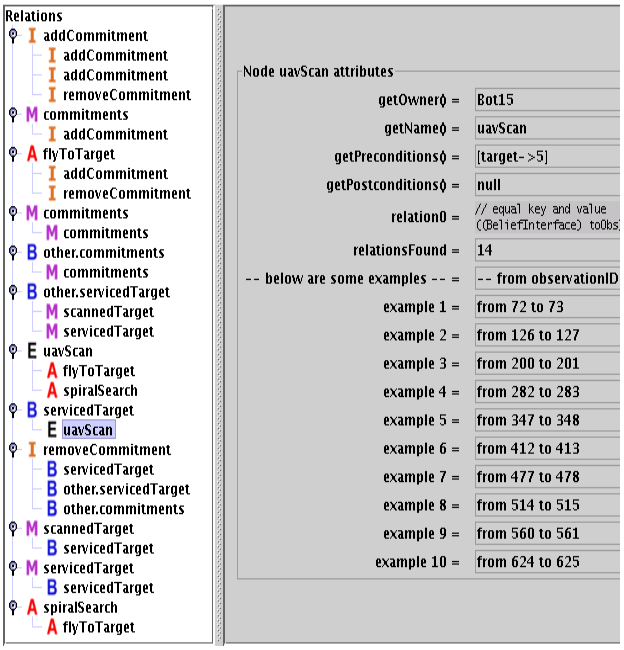

Figure 3: Suggested relations from the Tracer Tool for the UAV application.

these inconsistencies could be any of the following: (1) a bug exists in the implementation, (2) a defined relation in **K** is incorrect, (3) **K** is incomplete and requires additional agent concepts or relations, or (4) the location where the logging code was inserted by the user is incorrect. The Tracing Method was designed to insure that the user's comprehension, represented in **K**, is complete with respect to a set of execution scenarios and correctly reflects the implementation.

To track down bugs in the design or implementation, the Tracer Tool can generate explanations for a specified observation (e.g., an anomalous action performed by an agent) in terms of agent concepts. An explanation is created by traversing (backwards) through the relational graph interpretation. Starting from the observation *o* being explained, incoming relations (edges) are followed to other observations that caused or preceded *o*. Given a complete **K**, an observation can be followed all the way back to the initial observations. If **K** is incomplete, explanations may not be complete (i.e., may not include all relevant observations).

As seen in Figure 4, an explanation is shown as a tree structure consisting of observations, thus keeping the explanation in the realm of agent concepts, familiar to the designer, developer, and end-user. For example, action A:4111 was a result of belief B:4057 and intention I:4095. Looking at the details of those nodes, the B:4057 was a precondition that enabled the action and I:4095 was the intention that included the action. Going deeper into the tree, B:4057 was a result of event E:4055, which in turn was caused by A:2656. Similarly, I:4095 was formulated from belief B:4079 and previous intention I:2640. The depth of the explanation tree continues until an observation with no incoming relation exists, which is one of the initial observations or an exogenous event that independently occurs in the environment. If the explanation does not end with one of these observations, then **K** is incomplete and requires that relations be added. Section 4 demonstrates the algorithm used to suggest relations and summarizes some experimental results.

In addition to showing *what* is happening in the system in terms of agent concepts, the Tracer Tool can facilitate software comprehension by generating explanations that describe *why* agents behave as they do. Lam and Barber [12] presents a detailed example of how the Tracer Tool was used to generate explanations in the UAV (unmanned aerial vehicle) application domain and to comprehend the agents implemented for that domain. In that paper, relations in **K** were manually specified as rules, each defining the relation between observations. In this paper, relations are suggested automatically.

4. SUGGESTING RELATIONS

Automatically suggesting relations is only valuable if most of the suggested relations are semantically correct. To show how well the relation-suggesting algorithm currently performs for different applications, this section describes the experimental results of applying the Tracer Tool on two multi-agent systems: "Simple" (about 500 lines of Java code) and "UAV" (about 20,000 lines of Java code).

The relation-suggesting algorithm is initiated for an observation *o* when the Tracer Tool cannot create an incoming relation for observation *o* based on the current **K**. This occurs when **K** has no incoming relation for the agent concept corresponding to observation *o* that originates from some other observation. Starting from *o*, the algorithm searches backwards (temporally) through the observation list, using heuristics to determine if a previous observation is related in some way to *o*. For example, if *o* is an action, then the algorithm searches for the last observed intention *i* that has some similar attribute as those of the action *o*. If such an intention is found, a relation from intention *i* to action *o*

is suggested. The relation-suggesting algorithm is outlined in pseudo-code as follows:

```
suggestRelationFor(OBSERVATION o) {
 for each HEURISTIC h {
  for each past_OBSERVATION p
  within h's searchHorizon {
   if ( h.appliesTo(p,o) ) {
    tests = commonAttributes(p,o);
    if ( ! empty(tests) ){
     return createRelation(p,o,tests);
 }}}}
 return null;
}

commonAttributes(OBSERVATION p, OBSERVATION o) {
 set of TESTs s;
 for each ATTRIBUTE a of OBSERVATION p {
  for each ATTRIBUTE b of OBSERVATION o {
   if ( valueOf(a)==valueOf(b) ) {
   add TEST( "valueOf(a)==valueOf(b)" ) to s;
 }}}
 return s;
}
```

The heuristic's search horizon specifies how far back to search for a relating observation. It can be `none` (which will search until a relation is suggested or the first observation is reached), or it can be a predicate, such as `isIntention` (which will search backward up to the last intention). An example heuristic that searches for the relation connecting an action to an associated event is shown below:

```
heuristic1.appliesTo(p,o) {
 return ( IS_ACTION(p) and IS_EVENT(o) and
  !empty(commonAttributes(p,o)) );
}
```

The algorithm always terminates because it only looks at preceding observations and there is a finite number of past observations. The computational complexity of the algorithm in the worst case is $O(n^2)$, where n is the number of observations.

For each observation without an incoming relation, the algorithm suggests a relation and adds it to the list (if it has nor already been added) or appends it to an existing relation, as shown in Figure 3. For example, a relation is suggested for the belief observation *servicedTarget*. Each child of *servicedTarget* (in this case, there is only one) is a possible explanation (or cause) for *serviceTarget*. So, the *uavScan* event is a possible explanation for the *serviceTarget* belief, which is semantically correct for the UAV application. Other observations, such as the *flyToTarget* action, may have more than one possible explanation. Let each parent-child pair be called a *subrelation*. For example, *flyToTarget* action has 2 subrelations.

If the relation has already been suggested, then the corresponding observation IDs are added to the list of examples (seen on the right side of Figure 3). The examples are provided so that the user can investigate the example observations to determine if the suggested relation is semantically correct.

For each domain application, logging code for each agent concept was inserted into the source code. Table 1 shows the number of agent concepts logged for each agent system. The Tracing Method was iterated until **K** was complete. In other words, all observations were correctly linked to all appropriate observations as in Figure 2.

Table 1: Results for relation-suggesting algorithm.

	Simple	**UAV**
agent concepts	17	22
correct suggestions	12 (13)	11 (17)
incorrect suggestions	0 (2)	1 (2)
manually-created	4 (5)	2 (2)

Table 1 summarizes the number of relations that was correctly suggested, incorrectly suggested, and manually-created. For more precise measurements, the number inside parentheses is the number of subrelations. The results show that the relation-suggesting algorithm captures approximately 66% of the relations in the Simple system and approximately 80% in the UAV system with relatively few incorrect suggestions. Although further testing on other agent-based systems is required, the algorithm shows promise in automating the tedious task of associating observations with each other.

The relation-suggesting algorithm can be improved to identify more relations, but such improvements may not be generalizable to other domain applications. A major focus of the Tracer Tool is to remain domain-independent and allow users to specialize the tool for their own applications. Future work for the Tracer Tool includes allowing the user (1) to control which heuristics are used to determine if two observations are possibly related and (2) to define additional heuristics for their particular domain type.

5. SUMMARY

Software comprehension is essential for developing, maintaining, and redesigning complex software, such as agent-based systems. This research aims to remedy the drawbacks and limitations of existing techniques (i.e., reverse engineering and model-checking). Traditional reverse engineering tools produce large amounts of detailed documentation that the user must manually navigate, investigate, and decipher – time-consuming and inefficient tasks. Lange *et. al.* comment that "scanning through complex diagrams, whether on paper or GUI, is no efficient way to comprehend large software systems" [13]. Though model-checking offers conciseness and abstraction in its model representation and exhaustive search in behavior verification, the representativeness of its model with respect to the actual implementation is difficult to prove and maintain as the implementation evolves.

This paper describes three contributions that extend ideas from existing work to assist the user in comprehending agent software. First, a high-level representation (called the knowledge base **K**) that explicitly describes the user's growing knowledge of the software's behavior was defined. The representation uses agent concepts that are familiar to the designer, developer, and end-user. The explicit representation enabled the second contribution, which is the Tracing Method and Tracer Tool. The Tracing Method describes a process to create, refine, and verify **K** (the

user's understanding of the system) with respect to the actual implementation. With the aid of the Tracer Tool, many of the manual tasks, such as scanning for unexpected behavior, are automated. With a refined and complete **K**, the third contribution advances the state-of-the-art by automatically explaining why an agent performed some unexpected behavior, not only describing what is happening.

This research shows that some comprehension tasks can be automated to assist in developing the user's understanding of the system. In essence, the research approach is to imitate the learning cycle performed by humans. Specifically, a human collects and interprets data, forms hypotheses, and tests those hypotheses. The Tracer Tool generates interpretations from observations, suggests additions to the knowledge base, and collects observations to be interpreted. Additionally, this research leverages the semantics of agent concepts as abstractions and provides automated reasoning about the resulting abstracted representation. Future work includes behavior pattern recognition and anomalous behavior detection.

7. 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.

7. REFERENCES

[1] Agrawal, A., Du, M., McCollum, C., Systä, T., Wong, K., Yu, P., and Müller, H. A. Rigi - An End-User Programmable Tool for Identifying Reusable Components. In *Proceedings of Fifth International Conference on Software Reuse* (Victoria, British Columbia, 1998).

[2] Bruening, D., Devabhaktuni, S., and Amarasinghe, S. Softspec: Software-based Speculative Parallelism. In *Proceedings of 3rd {ACM} Workshop on Feedback-Directed and Dynamic Optimization* (Montery, California, 2000), ACM Press.

[3] Chikofsky, E. J. and James H. Cross, I. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7, 1 (1990), 13-17.

[4] Edmonds, B. and Bryson, J. The Insufficiency of Formal Design Methods. *Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, (2004), 938-946.

[5] Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müeller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. The Software Bookshelf. *IBM Systems Journal*, 36, 4 (1997), 564-593.

[6] Gasser, L., Braganza, C., and Herman, N. MACE: A Flexible Testbed for Distributed AI Research. In *Distributed Artificial Intelligence*, Huhns, M. N., ed. Morgan Kaufmann, San Mateo, CA, 1987. 119-152.

[7] Hindsight http://www.testersedge.com/hindsight.htm.

[8] Holzmann, G. J. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23, 5 (1997), 279-295.

[9] Jerding, D. and Rugaber, S. Extraction of Architectural Connections from Event Traces. In *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Montreal, Canada, 1998), ACM Press.

[10] Koskimies, K., Männistö, T., Systä, T., and Tuomi, J. Automated Support for Modeling OO Software. *IEEE Software*, 15, 1 (1998), 87-94.

[11] Kullbach, B. and Winter, A. Querying as an Enabling Technology in Software Reengineering. In *Proceedings of 3rd European Conference on Softward Maintenance and Reengineering* (Los Alamitos, 1999), IEEE Computer Society, 42-50.

[12] Lam, D. N. and Barber, K. S. Debugging Agent Behavior in an Implemented Agent System. In *Proceedings of 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), 45-56.

[13] Lange, C., Winter, A., and Sneed, H. M. Comparing Graph-Based Program Comprehension Tools to Relational Database-Based Tools. In *Proceedings of Ninth International Workshop on Program Comprehension* (Toronto, Canada, 2001), IEEE Computer Society, 209.

[14] Lucca, G. D., Fasolino, A., and Carlini, U. Recovering Use Case Models from Object-Oriented Code: A Thread-based Approach. In *Proceedings of 7th Working Conference on Reverse Engineering* (Brisbane, Queensland, Australia, 2000), 108-117.

[15] Mayrhauser, A. v. and Lang, S. On the Role of Static Analysis during Software Maintenance. In *Proceedings of 7th International Conference on Program Comprehension* (Pittsburgh, PA, 1999), IEEE Computer Society, 170-177.

[16] Poutakidis, D., Padgham, L., and Winikoff, M. Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols. In *Proceedings of First International Joint Conference on Autonomous Agents and MultiAgent Systems* (Bologna, Italy, 2002), ACM Press, 960-967.

[17] Robby, Dwyer, M. B., and Hatcliff, J. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Helsinki, Finland, 2003), ACM Press, 267-276.

[18] Schauer, R. and Keller, R. K. Pattern Visualization for Software Comprehension. In *Proceedings of 6th International Workshop on Program Comprehension* (Ischia, Italy, 1998), 4-12.

[19] Sim, S. E. and Storey, M.-A. A Structured Demonstration of Program Comprehension Tools. In *Proceedings of Seventh Working Conference on Reverse Engineering* (Toronto, Ontario, Canada, 1999), IEEE Computer Society, 184.

[20] Stroulia, E. and Systä, T. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review*, 10, 1 (2002), 8-17.

[21] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model Checking Programs. *Automated Software Engineering Journal*, 10, 2 (2003).