The Dissertation Committee for Dũng Ngọc Lâm
certifies that this is the approved version of the following dissertation:

Agent Software Comprehension:
Explaining Agent Behavior

Committee:

K. Suzanne Barber, Supervisor

Anthony P. Ambler

James C. Browne

Raymond J. Mooney

Dewayne E. Perry

Agent Software Comprehension: Explaining Agent Behavior

by

Dũng Ngọc Lâm, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2005

Dedicated to my wife Andrea, daughter Elena,
grandparents Rang and Tuyet,
parents Frances and Hao,
and siblings Yvonne, Frank, and Andrew.

Acknowledgments

I wish to thank the numerous people who helped me to complete this great achievement. I begin with my Supervisor, Dr. Suzanne Barber, who has been a constant and prominent pedestal for my research and who has encouraged me in my times of need. Next are the administrative staff for the LIPS Lab, Darla and Charlotte, without whom I would be amiss in the hoops and bureaucracy of UT. Then, there are my current labmates, Dave, Karen, Jisun, Jaesuk, Nishit, and DeAngelis, who make the days go by a bit faster. Also, there are the former labmates, Tom, Cheryl, Anuj, Ryan, Bill, who contributed to my research in some manner and who continue to be great friends to this day. Finally, of course, is my family to whom this dissertation is dedicated – my loving wife Andrea, my daughter Elena, my supportive parents Frances and Hao, and my understanding grandparents Rang and Tuyet.

The equipment and software used to perform this research was funded in part by the Texas Higher Education Coordination Board Advanced Technology Program (ATP) Grant #003658-0188-1999 and by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0588.

Agent Software Comprehension: Explaining Agent Behavior

Publication No. _____

Dũng Ngọc Lâm, Ph.D.
The University of Texas at Austin, 2005

Supervisor: K. Suzanne Barber

It is important for designers, developers, and end-users to comprehend (or explain) why a software agent acts in a particular way when situated in its operating environment. Comprehending agent behaviors in an agent-based system is a challenging task due to environmental uncertainty and the dynamics and multitude of agent interactions, which must be captured, processed, and analyzed by the human user. While traditional software comprehension answers "what is happening in the implementation?", this research takes a step further to facilitate comprehension by answering "why is the behavior happening in the implementation?". To explain agent behaviors in the implemented system, this research takes the model-checking approach for representing abstracted software behavior and the reverse engineering approach for verifying the expected behavior model against the implementation's actual behavior, while assimilating the terminology and framework from abductive reasoning. This research empirically shows that maintaining accurate background knowledge of how the implementation is expected to behave is crucial in generating accurate explanations of agent behavior. The resulting Tracing Method and accompanying Tracer Tool build on ideas from existing approaches and extend the

state-of-the-art to better assist human users (of various skill levels) in comprehending agent-based software by automating many reasoning tasks. The Tracing Method is applied to two domains to demonstrate the capabilities of the Tracer Tool in (1) suggesting background knowledge updates, (2) interpreting actual behaviors from implementation executions, and (3) explaining observed agent behaviors. This research aims to help designers who want to improve agent behavior; developers who need to debug and verify agent behavior; and end-users who want to comprehend agent behaviors.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

INTRODUCTION

> *Concurrency, problem-domain uncertainty, and non-determinism in execution together conspire to make it very difficult to comprehend the activity in a distributed intelligent system [...]. We urgently need graphic displays of system activity linked to intelligent model-based tools which help a developer reason about expected and observed behavior.*
>
> [Gasser et al., 1987]

Agents are distributed software entities that are capable of autonomous decision-making in order to achieve some defined goal. In multi-agent systems, agents can communicate in order to coordinate, negotiate, and collaborate during decision-making. Besides being motivated by its own initially-defined goals, an agent's behavior is influenced by interactions with other agents (i.e., their goals, beliefs, and relative roles), by events that have occurred in the past, and by the current situation. It is important for software designers, developers, and end-users to comprehend (or explain) why an agent acted in a particular way when situated in its operating environment, which is often unpredictable and uncertain. Currently, the process of comprehending (or generating an explanation of) agent behavior is done manually by observing actual behavior from the implementation executions and comparing them with expected agent behavior, which may have been derived from the software design, previous experience, intuition, etc. Considering the complexities of agent

software and the usual disparity between design and implementation, software comprehension is a difficult, time-consuming, and tedious process. To alleviate these issues, this research proposes an extension of software comprehension called *agent software comprehension* (ASC) and an innovative and practical approach to automatically generate explanations of agent behavior. The artifacts of this research include tools to help represent, build, and verify the user's comprehension of the implemented agent system.

Software comprehension, which historically has been associated with program comprehension and reverse engineering, involves extracting and representing the structural and behavioral aspects of the implementation in an attempt to recreate the intended design of the software. Software comprehension is motivated by the fact that designing and implementing a system is not always done right the first time and that maintenance is a major financial and temporal cost of the software life-cycle. The software may need to be (1) verified to ensure that the implementation is behaving as it was designed to behave; (2) maintained to fix bugs or make modifications; or (3) redesigned and evolved to improve performance, reusability, or extensibility (among other reasons). In order to perform these tasks, an understanding of the current implementation is required and is attained using software comprehension tools and techniques.

For agent software in particular, the difficulties in comprehension are exacerbated by characteristics of the individual agents and of the system itself. Agents are autonomous 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 why the agents made certain decisions or why it behaved in certain ways. Designers and developers must also consider the system's distribution, concurrency, domain uncertainty, and non-determinism [Gasser et al., 1987]. This research 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 by automating human reasoning.

Traditional software comprehension is synonymous with reverse engineering (RE) 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 programmatic components (e.g., objects and classes) and their dependencies, RE involves creating abstractions of the system design [Chikofsky and James H. Cross, 1990]. 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 [Agrawal et al., 1998; Finnigan et al., 1997]. RE tools analyze the implementation at a very low abstraction level (i.e., at the source code level) and, thus, are inappropriate for agent software because they produce models of the implementation that are too detailed (e.g., component dependence and class inheritance models). RE tools are useful for recreating designs that are detailed, where analyzing sections of code is appropriate. However, these tools do not provide abstracted views of the implementation as a whole in terms of high-level agent concepts (e.g., beliefs, tasks, goals, and communication messages). Wooldridge states that as software systems become more complex, more powerful abstractions and metaphors are needed to explain their operation because "low level explanations become impractical" [Wooldridge, 2002]. To attain an understanding of agent behavior, the models resulting from the comprehension process must be at the abstraction level where high-level agent concepts are the elemental or base concepts. By speaking

in terms of agent concepts (e.g., beliefs, goals, communications), designers, developers, and end-users can readily communicate with each other about the agent's behavior and can understand the system as a whole because agents concepts are familiar.

Additionally, since RE tools parse the source code directly, they are limited to programming languages that are parsable by the tool. For real-world applications, RE tools need to support software that use more than one programming language [Sim and Storey, 1999]. The Tracer Tool resulting from this research avoids parsing the implementation's source code so that the tool can operate with any software system implemented in practically any mix of languages. This is possible by having the user label certain parts of the source code with abstracted agent concepts, thus, providing a mapping between implementation constructs and high-level agent concepts (this process is described fully in Chapter 3). Such abstractions help to bring the ideas manifested in the implementation closer to familiar design concepts, thus, aiding the comprehension of the overall system behavior, independent of 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). For example, in SCED, detailed event trace information (e.g., method invocation) is used to create a set of sequence and state diagrams [Koskimies et al., 1998]. Other tools, such as Hindsight and Lemma, can create flowcharts and control-flow diagrams [Hindsight, 2004; Mayrhauser and Lang, 1999]. Such dynamic analysis tools (e.g., Tango [Stasko, 1990] and ISVis [Jerding and Rugaber, 1998]) also face the same problem of detailed representation of programmatic concepts such as process threads, remote procedure calls, and data structures, rather than agent-oriented models of goals, plans,

and interaction protocols. Dynamic analysis is particularly important for agent systems that operate in the presence of environmental dynamics and uncertainty. This research leverages agent concepts to build abstract representations of the agents' run-time behavior (i.e., relational graphs), which can be quickly understood by the user and can also be used for automated reasoning to further assist the user.

RE tools tend to produce a large amount of data for the user to browse and interpret (even for small systems). To deal with this, SoftSpec allows users to query a relational database of automatically gathered information about the software architecture [Bruening et al., 2000]. Alternatively, using a graph-oriented approach to more efficiently analyze and search through the large amount of data, the GUPRO toolset transforms source code into graphs according to a defined concept model [Kullbach and Winter, 1999]. Similarly, the research presented in this dissertation deals with large amounts of data by automating data interpretation for the user, except the data is at a higher abstraction level. Instead of a list of unconnected, detailed data that the user must relate manually, the presented solution automatically relates run-time observations together as a relational graph.

RE tools only produce representations of the implementation and have no model of the user's comprehension. It is the user's responsibility to digest the RE results (e.g., diagrams, charts, and databases). RE tools do not reflect how much the user understands and thus, cannot provide feedback to the user about the user's comprehension. However, in model-checking, the user expresses their understanding of the implementation as a model, which can be automatically checked for specified properties. Thus, model-checking tools have a representation of the user's comprehension of the system. Though useful due to the exhaustive state-space search employed, model-checking techniques in general do not verify the accuracy of the model with respect to the actual system. Hence, any checked properties may not

apply to the actual implementation. By combining model-checking with reverse engineering, this research maintains a model of the user's comprehension (as the user is learning about the implemented agent system) and also ensures that the model accurately represents of the actual system.

Acknowledging the value of comprehending agent behavior and the limitations of traditional software comprehension tools, this research proposes agent software comprehension (ASC) as a new research area for agent-oriented software engineering that addresses the issues described above. ASC is software comprehension for agent systems, or systems made up of software agents. In addition to representing the agent-oriented models of the implementation, ASC includes reasoning about those models; specifically, this research focuses on explaining agent behavior. While traditional software comprehension answers "what is happening in the implementation", this research takes a step further to facilitate comprehension by answering "why is it happening in the implementation". The ability to explain the system's behavior demonstrates comprehension of the system, answering both what is happening and why it is happening.

Traditionally, the reasoning process for comprehending software behavior is performed manually. First, events, agent actions, etc. are observed from running the implementation. Then, those observations are interpreted using models of how the particular agent is expected to behave (e.g., state charts). These expected behavior models are usually derived from design documents, experience with the agent system, and intuition about agent systems and are referred to as *background knowledge*. Since there is no direct way to measure how much the user comprehends, a person's comprehension of a subject is indirectly measured by how much the person can explain about the subject because the process of creating an accurate explanation demands correct comprehension of the system. Explanations bridge the gap

between expected and actual behavior – specifically, the explainer's background knowledge of the software's behavior and observations of the software execution (e.g., agent actions). Thus, explanations can be very important in designing, debugging, and trusting agent behavior.

Unfortunately, ensuring accurate explanations is difficult because the implementation evolves over time and there are many factors that can influence agent behavior. First, since comprehending the behavior of the implemented system relies on how accurately the background knowledge represents the implementation, the representative accuracy of the background knowledge must be maintained as the implementation changes. The second problem in manual explanation generation is that an explanation may be too difficult to conceive due to the sophistication (e.g., in reasoning or agent interaction) of the agent system or the amount of observed data to consider. In response to these difficulties, this research proposes an automated approach to agent software comprehension that can handle large amounts of observation data and can automate the generation of explanations to aid the user in comprehending the system as the implementation evolves over time.

The contributions of this research include models of agent behavior, a method to build agent-oriented models that are representative of the implementation, and a tool to automate comprehension tasks. This research combines and extends ideas from reverse engineering and model-checking. The result is a high-level, more scalable, practical, semi-automated solution for agent software comprehension called the *Tracing Method*. Comprehension is performed at the system-level using high-level agent concepts that are familiar to designers, developers, and end-users. Hence, all activities in the Tracing Method 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 con-

cepts, scalability is dependent on the number of agent concepts, rather than on code size or state-space complexity. The solution is practical in that (1) models of the user's comprehension accurately represent the implementation; (2) it can be applied to any programming language (that can log data to a file); and (3) the learning curve is low, unlike most RE and model-checking tools. The Tracer Tool assists the user by automating as much of the Tracing Method as possible (e.g., building the model, generating explanations, and detecting anomalies).

The following sections define the terminology of the research problem (Section 1.1) and state the research objectives in terms of research questions (Section 1.2). The remainder of this dissertation will focus on answering these research questions.

1.1 Problem Statement

To better describe the research problem of *comprehending agent behavior in a situated agent system*, the working definitions used in this research are given as follows:

(software) agent : a distributed software entity that is autonomous and goal-directed (makes decisions on its own about when to plan for a goal and what actions to take).

agent behavior : the relationship between agent actions and factors that influence those actions (including agent goals and beliefs, inter-agent interaction, and environmental events); behavior defines a mapping from the situation an agent is in and the state of the agent to the resulting actions performed by the agent.

(multi-)agent system : an executable implementation consisting of software agents and the infrastructure that enables interaction.

situated agent system : an agent system that has been deployed and executes within its intended environment (e.g., simulation or actual hardware).

to comprehend agent behavior : to be able to explain why an agent performed some particular action when situated in its environment.

explanation : relational statements (e.g., causal statements) that identify factors influencing an agent's action[1].

Given these definitions, the research problem can be precisely restated as *finding explanations for agent actions that occur during the execution of the implemented agent system*. An automated system that produces such explanations would be helpful to a wide range of users for comprehending the implementation, and consequently, helping to debug, test, maintain, and evolve the software.

The three major types of users targeted by this research are designers, developers, and end-users. Designers can utilize the capabilities of the Tracer Tool to compare the design against the implementation, thereby verifying which components have been implemented and whether the implementation's behavior is as expected from the design. This capability is crucial in situations where the design is not updated and the implementation's behavior evolves due to software requirements changes, maintenance tasks, feature additions, or performance enhancements. The generated explanations can help designers reason at an abstract level and track down why unexpected behavior occurs so that the design can be updated or the implementation can be corrected.

[1] In general, any observation can be the subject of an explanation.

For developers, this research provides a method to create a comprehensive visualization of the system. Both novice and experienced developers are offered a graphical depiction of the effects of their programming, at the level of abstraction that can be understood by other developers, as well as other types of users. Using the visualization, the developer can verify that the software behaves as intended. In particular, if the developer is involved with only a part of the software, then the developer can observe the effects of his/her part on other parts of the software. Understanding software behavior is essential for improving the software. Since software comprehension is often hindered by the large code size of the implementation, the approach taken in this dissertation is dependent not on the source code size, but on the number of agent concepts identified in the implementation. This is accomplished by only recording abstract agent concepts observed when executing the system and not directly analyzing the source code itself. As a result, this approach can be applied to software of any size to construct a system-level view of the agent system. Additionally, the Tracer Tool can generate explanations at a high abstraction level that also references specific observations that can be traced to a localized location in the source code for debugging purposes.

For end-users, who are not intimately involved in the design or development of the software, the Tracer Tool treats the implementation somewhat like a black-box. The end-user may have little or no knowledge of the expected behavior of the system. By recording high-level observations of what the agents are doing, the end-user begins to populate the background knowledge (to fill in the user's comprehension) of the system. Given these observations, the Tracer Tool can then suggests possible causal relations among those observations using general heuristics that characterize agent system behavior, thereby forming a more complete representation of the implementation's behavior. Once the causal relations have been defined

and the observations captured, an explanation of observed agent behavior can be readily generated and used for further comprehension. These explanations can help end-users determine whether to adopt the agent-based software solution.

Explanations employ familiar abstract terminology to help promote interaction among designers and developers to resolve differences or to encourage productive discussion between end-users and the development team (designer, developer, and project manager). In summary, the abstract agent concepts employed by the automatically generated visualization and explanations help to break down the language barrier among different types of users.

Before stating the hypothesis, a discussion of the technical aspects of the problem (i.e., generating explanations for agent actions) is required. An explanation for a manifestation (e.g., an agent action) relates relevant factors that influence the occurrence of the manifestation to the manifestation itself. To create an explanation, three inputs are needed – the manifestation, background knowledge, and observations [2].

manifestation m : the observed fact being explained, e.g., an agent's action or environmental event.

background knowledge K : the set of relevant relations between influential factors (e.g., causes) and manifestations (e.g., effects); describes how the system is expected to behave.

observations O : facts about aspects of the system gathered during execution (e.g., actions, events, and beliefs).

[2]The terminology is borrowed from abductive reasoning, which used to generate plausible explanations given an incomplete set of observations.

During the explanation process, background knowledge is used to associate relevant observations to the manifestation. The background knowledge specifies concepts (which are associated with observations), and relations between concepts (which help link observations together). Commonly, the links form a causal chain connecting observations to the manifestation being explained. For example, background knowledge may specify that if a switch is flipped up, the blades on the fan turn around, which causes the temperature to decrease. If an observation is that the fan blades are turning, then an explanation for the low temperature (the manifestation) would be that the switch was flipped up, causing the fan blades to turn, which causes the low temperature.

Note that the abstraction level of the explanation is at the same abstraction level as the background knowledge. Since an explanation using high-level agent concepts is desired, the background knowledge must be represented using agent-oriented models.

agent concepts : abstract concepts that are used to describe agents; for example, e.g., beliefs, tasks, goals, and communication messages.

agent-oriented model or **agent model** : a model that uses agent concepts; for example, belief dependence diagram, task decomposition, and communication protocol diagram.

Continuing with the example, since an abstract explanation for the low temperature is desired (rather than a detailed description of the switch mechanism, electron flow, air currents, etc.), the background knowledge only needs a causal relation between the switch and the fan and between the fan and the lower temperature. As long as the background knowledge accurately specifies the relationships among the concepts, an accurate explanation is possible. In general, the quality (i.e., accuracy

and completeness) of an explanation is dependent on (1) the number and relevance of the observations, (2) the accuracy of the background knowledge with respect to what is being explained, and (3) the competence of the explanation process.

Given the brief description of the issues encountered when generating explanations, the hypothesis of this research is stated as follows:

> *Accurate explanations of actual agent behavior can be generated if models of expected agent behavior being used as background knowledge are representative of the implementation's behavior.*

An explanation ϵ for a manifestation m is created using background knowledge K and a set of observations O from the implementation execution:

$$\epsilon = explain(m, K, O), \text{ where } m \in O \quad [1.1]$$

Since observations O are facts, the only other source of error when explaining manifestation m is background knowledge K. Thus, the accuracy of the explanation ϵ depends on how accurately the background knowledge K represents the actual system in which the manifestation m occurred. The issues addressed in this research are (1) building a K that accurately represents the implementation's behavior and (2) using K to generate explanations. These two issues are the two research questions posed in this dissertation.

Recall that ASC involves (1) representing the implementation using agent models and (2) reasoning over those representations to facilitate the user's understanding of the system. ASC is difficult because there is no common representation of agent behavior, and agents employ sophisticated reasoning processes and operate in unpredictable environments. The answers to the following research questions must address these difficulties.

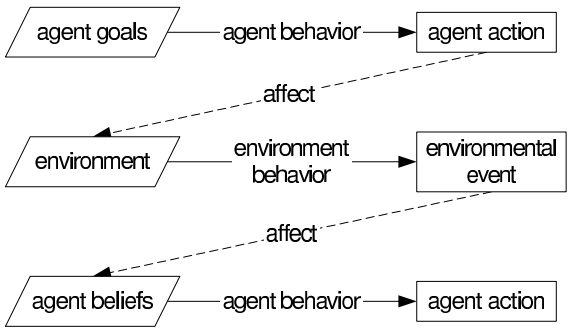


Figure 1.1: Some agent factors that can affect agent behavior

Research Question 1 : *How can background knowledge be created such that it accurately represents the implementation's behavior?*

Research Question 2 : *How can background knowledge be used to explain observed agent behavior?*

1.2 Research Questions

This section describes each research question in detail and the general approach taken to answer them.

1.2.1 Research Question 1: Building background knowledge

Research Question 1 (RQ1) is "*How can background knowledge be created such that it accurately represents the implementation's behavior?*" Background knowledge K represents the user's comprehension and also the expected behavior of the implemented system. For example, in model-checking, K is the model generated by the user that is to be checked. K also represents the behavior that the user expects from the system. K may have been derived from many sources, such

as specifications of the design, experience with the implementation, intuition from presentations, etc. K may be erroneous or inaccurate in that actual behavior of system is different from what is described by K. To generate accurate explanations, K must be accurate in that expected behavior should accurately reflect actual behavior. For software systems, actual behavior is exhibited as observations O from implementation executions. Due to complexities and uncertainties of some systems, exact behaviors cannot be predicted from only the design specification [Jennings, 2000]. Additionally, for software systems that must be updated and maintained over time, the design often becomes outdated.

The overall approach of this research is to build up the background knowledge K (representing the user's understanding) using observations from the actual implementation's executions, rather than relying on design specifications. As a result, everything in K is based directly on the actual implementation (similar to the RE approach). Modifications to K (e.g., addition of relations between agent concepts) are automatically suggested by the Tracer Tool. However, unlike RE, where detailed models are automatically created for the user to digest, this approach demands that the user confirms all modifications to K so that K reflects what the user comprehends. In other words, since the user is building K, there is nothing in K that the user does not already comprehend or at least seen.

Liedekerke and Avouris were one of the first to address the problem of debugging multi-agent systems, offering a tool that visualizes different aspects of the system at the agent concept level [Liedekerke and Avouris, 1995]. However, their approach does not consider what the user understands (the background knowledge) in order to identify unexpected or undesired behaviors from the user's perspective. The visualization/debugging tool suite provided by Ndumu et al. also does not account for what the user understands but instead collates, fuses, and presents in-

formation from disparate data sources [Ndumu et al., 1999]. In these approaches, the tools cannot provide feedback to update the user's comprehension when the implementation changes over time.

When the implementation's behavior changes (resulting from design changes or maintenance tasks) and is different from the expected behavior represented by K, the Tracer Tool alerts the user of the anomaly so that K can be updated. Since changes to the implementation can be propagated to K, the accuracy of K with respect to the implementation is maintained as the implementation evolves. The following briefly describes how this approach addresses the comprehension issues mentioned earlier in the Introduction:

low abstraction level : Background knowledge K is represented as a collection of high-level agent concepts familiar to designers, developers, and end-users.

language-dependent : The Tracing Tool records observations logged from the implementation's execution, rather than analyzing language-dependent stack traces and process threads.

large amount of data : The Tracer Tool automates the task of collecting, organizing, and interpreting the observations and can present the interpretation to the user as a relational graph that can be quickly understood.

human user must digest data : Given interpretations and K, automated reasoning can highlight new concepts and relations that the user has not yet modeled in K.

With this approach, (1) K accurately represents the actual system (i.e., the implementation) in terms of agent concepts familiar to the user and (2) K is a formal model of the user's comprehension that can be used for automated reasoning.

One type of automated reasoning that is explored in this research is explanation generation. Using the representative models as background knowledge K, explanations of actual agent behavior that are consistent with run-time observations can be produced, which leads to the next research question.

1.2.2 Research Question 2: Explaining agent behavior

Given accurate background knowledge resulting from RQ1, Research Question 2 (RQ2) is "*How can background knowledge be used to explain observed agent behavior?*" Agent behavior is defined as the relationship between agent actions and factors that influence those actions. Figure 1.2 illustrates the relationship between individual behaviors and the explanation linking the relevant behaviors. Explanations make connections among individual agent behaviors and are often used to determine causes for observed agent actions and, thus, to better comprehend the reasons for agent behaviors.

One popular way of generating explanations is to use abductive reasoning (or abduction). Abduction is a form of nonmonotonic reasoning where explanations of manifestations are generated based on knowledge of how the domain works and a given set of observations (which is usually incomplete). In the medical field, abductive reasoning was used to form a plausible diagnosis (the illness or cause of the illness), given the symptoms (or effects of the illness). The diagnosis is the explanation and the symptoms are the observations. The background knowledge used to generate the explanation is a compilation of illnesses and their symptoms. In addition to medical diagnosis [Console et al., 1991], abduction has been applied to other intelligent tasks, including natural-language processing [Stickel, 1989], plan recognition [Appelt and Pollack, 1992], prediction, classification [Mooney, 1997], and case-based reasoning [Leake, 1993]. The formulations of abductive reasoning

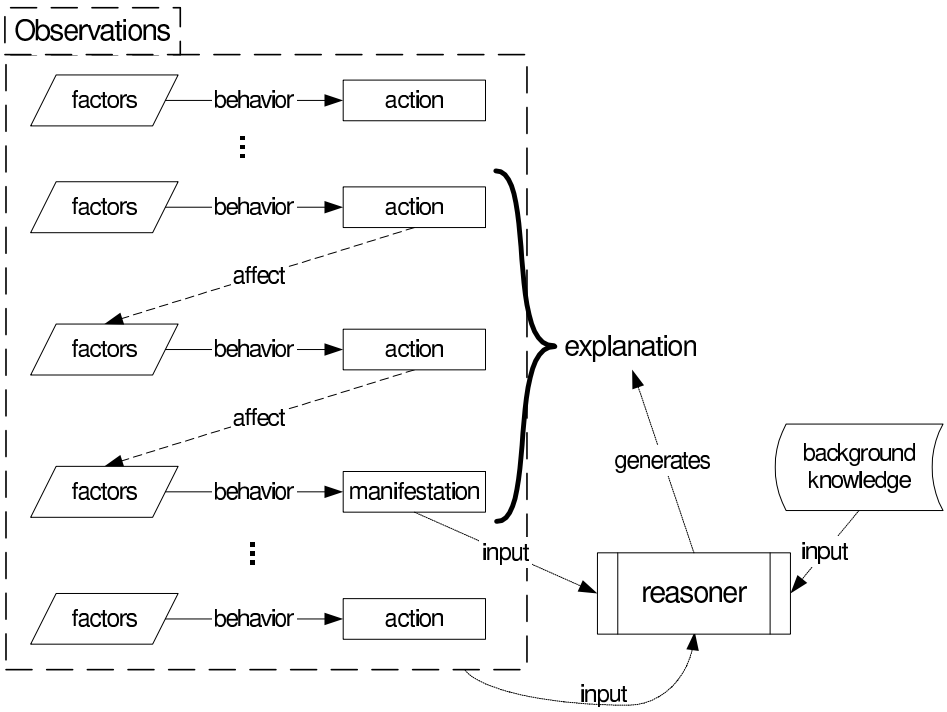


Figure 1.2: Relationship between behaviors and explanation

try to capture one aspect of how humans learn, specifically how humans rationalize observations by imposing causal relationships among observations. The approach of this research borrows the general conceptual model used in abduction and applies it to the software comprehension domain. The following enumerates the issues of existing abductive reasoning techniques and how this research addresses them:

- Many of the techniques depend on the background knowledge K being complete and provided by the user. This research incorporates the building of K in the comprehension process and guides the user in constructing an accurate representation of the domain. K is continually modified as the user gains comprehension and, thus, is never assumed to be complete.

- Abduction techniques try to capture causality (which in itself is difficult to define) using rules and conditional probabilities in their representation of K. Instead, this research uses semantic networks, which allows the user to relate observations in other ways, such as temporal or compositional relations, that make sense to the user.

- Propositional variables used in propositional logic and Bayesian networks to model observations lack the richer representation of predicates or composite variables. This research employs an object-oriented approach, whereby agent concepts (e.g., beliefs, intentions, and actions) are classes and observations are object instances of those classes. In so doing, an observation can have attributes that offer further details about the observation, such as when it occurred, pre- and post-conditions of the observation, agents involved in the observation, and references to previous observations.

- Bylander et al. analyzed the computational complexity of abduction and showed that it is NP-hard [Bylander et al., 1991]. In general, abduction is NP hard due in part to incomplete observations – there is no polynomial-time algorithm to determine if a hypothesis is the best explanation. Finding the best explanation involves iterating through all possible hypotheses. However, the process of generating plausible hypotheses involves making assumptions about occurrences or facts that were not explicitly observed, which may take longer than polynomial time. Additionally, as new observations are considered, the set of plausible hypotheses is likely to change. In this research, by ensuring that the set of observations is complete (i.e., that all observations of the implementation's execution are recorded), no assumptions about observations need to be made and generating the explanation is a matter of mapping

the observations to concepts in K and making appropriate relations among those observations.

In the domain of software comprehension, the user's comprehension is used as background knowledge K, and agent-relevant data extracted from the implementation execution is used as the set of observations O. Resulting from RQ1, background knowledge K accurately represents the implementation, and since all observations are systematically recorded, the observations O are true and complete. Given these, the explanation for a manifestation $m \in O$ is generated by (1) mapping the m to a concept in K, (2) looking up other concepts in K that influence (and relate to) m, and (3) finding observations in O that match those concepts. This process can be performed iteratively d times to produce a d-depth explanation (i.e., an explanation representing a causal chain of length d).

Using this approach, the explanation will be accurate in so far as the background knowledge K is accurate. Following from the hypothesis, if the explanation is not correct (i.e., does not "makes sense" to the user), then K may not be representative of the implementation. The explanation can be used to locate (1) which concept(s) or relation(s) in K is not accurate or (2) where in the implementation source code the erroneous observation(s) is occurring. This approach produces an explanation that is not only accurate with respect to the implementation but also reduces the number of error sources during the comprehension process, unlike abduction where assumptions (which may be incorrect) are made.

It is also possible that an explanation is correct, but it is not what the user desires. The explanations can help focus on and track down the cause of the undesirable behavior. With explanations for observations readily available to the designer, developer, or end-user, tasks such as redesigning, debugging, and understanding agent behavior, becomes a more manageable and less prone to human error.

♦◊♦

The remainder of this dissertation is organized into five chapters, where each chapter is guided by the two research questions. Chapter 2 reviews related work in the literature to set the background for this research. Chapter 3 describes the approach taken to answer the research questions. Chapter 4 and 5 details the Tracer Tool and experiments used to evaluate the approach in answering Research Questions 1 and 2. Finally, Chapter 6 summarizes the research objectives and enumerates the contributions of this research.

Chapter 2

BACKGROUND

> *Sadly, testing and debugging are still much neglected areas of agent development process. Developers need assistance with visualizing what is happening, and need debugging facilities to step through the execution and amend behavior where appropriate.*
>
> [Jennings and Wooldrige, 1998]

The objective of this research is to help users (i.e., designers, developers, and end-users) comprehend agent behaviors within agent-based software systems. As mentioned in Chapter 1, comprehending agent software is essential for using, creating, and advancing agent technology. However, sophisticated software such as agent systems presents obstacles that are difficult to overcome using current software comprehension and verification tools. In general, traditional software comprehension (or reverse engineering) tools are limited by their detailed abstraction level, their dependence on analyzing source code, their lack of automation to help decipher tremendous amounts of resulting data, and their lack of a model for how much the user understands. Taking the formal approach to modeling systems (and thus, understanding properties of systems), model-checking is limited by its demand for expert knowledge of the model-checking process, its high computational complexity, and the translation gap between the model being checked and actual system.

To remedy limitations of current techniques, this research offers a new approach to computer-aided software comprehension that involves:

(a) modeling the user's comprehension of the system,

(b) ensuring that the user's comprehension accurately reflects the actual system, and

(c) generating explanations as evidence of comprehension.

Research Question 1 (RQ1) addresses (a) and (b), while Research Question 2 addresses (c). This chapter reviews the background literature and related work in software agents, traditional software comprehension (or reverse engineering), and abductive reasoning. Limitations of existing work are described to motivate this research, and advantages are highlighted to introduce ideas that are used in this research.

Section 2.1 sets the general research context and scope for this dissertation. Section 2.2 discusses agent concepts and their utility in agent software engineering and comprehension. For RQ1, Section 2.3 describes traditional software comprehension (also known as program comprehension or reverse engineering), identifies the inadequacies of RE tools, and establishes the need for *agent software comprehension* (ASC). For RQ2, Section 2.4 describes the literature on abductive reasoning and how its foundations can be applied toward agent software comprehension.

2.1 Research Context and Scope

This research encompasses two major research areas: software engineering and artificial intelligence. Within software engineering, this research contributes to the field of agent-oriented software engineering (AOSE) and software comprehension. Though this research can be generalized for traditional software systems, this research focuses on agent-based software systems, where the distributed soft-

ware components (i.e., agents) can have sophisticated features (e.g., autonomous decision-making processes) and interaction dynamics (e.g., auction communication protocols). Because producing agent-based software is such a challenging task, there has been increasing activity in the last five years to advance AOSE. Though numerous research efforts have addressed the design and development of agent systems (e.g., Gaia [Wooldridge et al., 2000], MaSE [DeLoach et al., 2001], and JADE [JADE, 2000]), there has been little research for the maintenance and diagnosis of agent systems. Agent-oriented design uses agent concepts (e.g., agents, beliefs, and goals) to model the software system to be built. However, when the time comes to test and maintain the actual implementation, the agent concepts have been translated into the programmatic data structures in source code. To comprehend the implementation, agent concepts must be manually extracted from the behavior of the implementation's execution, which can easily become difficult as agent features become more sophisticated. To facilitate understanding, RQ1 aims to automate the process of agent concept extraction from the implementation.

Within the artificial intelligence area, this research tries to imitate human reasoning in the context of software comprehension. Since there is no direct method to measure human comprehension, the ability to generate correct explanations is used as a metric for comprehension. In order to generate explanations, the human's knowledge (or comprehension) of the system must first be represented (which is addressed in RQ1). In artificial intelligence, abduction is a popular form of nonmonotonic reasoning used to generate explanations despite incomplete information. The field of nonmonotonic reasoning aims to model "commonsense" reasoning in the real world, where the set of conclusions does not grow monotonically with increasing information [Minker, 1993] Abduction has been applied to several intelligent tasks, including natural-language processing [Stickel, 1989], plan recognition

[Appelt and Pollack, 1992], diagnosis [Console et al., 1991], prediction, classification [Mooney, 1997], and case-based reasoning [Leake, 1993]. RQ2 borrows the formulation of abduction and applies it to aid in software comprehension. By leveraging the controlled environments of software systems, any piece of data can be extracted or recorded from the software's execution. Thus, by using complete knowledge of the system (i.e., complete with respect to the user's comprehension of the system), the computational complexities of abduction (which is known to be NP-hard [Bylander et al., 1991]) can be avoided when generating explanations.

2.2 Agent Concepts

Agent system designers use the concept of agents to make building software for sophisticated, distributed domains easier because complex domain goals are decomposed into more manageable tasks that can be handled by individual autonomous entities, each with their own resources and decision-making capabilities. There are a number of agent-building tools (e.g., Concordia [Wong et al., 1997]), agent-based development environments (e.g., JADE [JADE, 2000]), and agent-oriented design methodologies (e.g., Gaia [Wooldridge et al., 2000] and MaSE [DeLoach et al., 2001]) that can be used to aid in the design phase (see Appendix A for a short survey of agent design methods). Different agent design methodologies produce different agent design models that are used as a blueprint to implement the agent system. Each agent design methodology utilizes its own set of agent concepts, from which design models are constructed. This research generalizes the set of agent concepts so that products of this research can be applied to most (if not all) agent-based systems.

Though agent concepts help in the design phase, there has been little research in leveraging them for the large maintenance phase of software engineering.

This research takes advantage of agent concepts to comprehend agent behavior in the implemented system. Agent software comprehension involves explaining how and why the agent makes certain decisions. Thus, the background knowledge K must include agent concepts and relations among those agent concepts, which are needed for generating explanations.

Agent concepts denote constructs used in agent-based systems (e.g., goals and beliefs) and are abstractions of low-level implementation constructs (e.g., data structures, classes, and variables). 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 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 [Stroulia and Syst, 2002]. 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 2.1 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

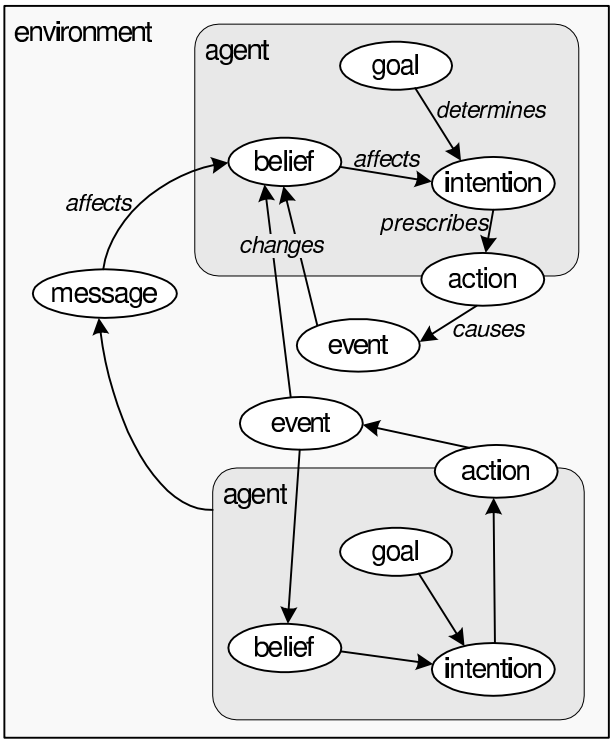


Figure 2.1: Agent concepts and some typical relationships

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 selected and performed.

To check the accuracy of the background knowledge K against the actual implemented system, a reverse engineering approach is used to compare expected behavior described by K against actual behavior. The next section discusses reverse engineering tools and argues for an extension to software comprehension called *agent software comprehension* (ASC).

2.3 Software Comprehension (for RQ1)

To enable generation of accurate explanations, background knowledge must be representative of actual behavior of the implementation. Software comprehension is used to check the representativeness of the background knowledge by comparing it to the reverse engineered models that result from reverse engineering the implementation. However, traditional software comprehension techniques and tools generally (1) are implementation language-dependent, (2) produce large amounts of low-level detailed data, (3) leave it up to the user to digest the data, (4) focus on static analysis of the source code, and (5) do not model what the user comprehends. Subsection 2.3.1 discusses the limitations of applying traditional software comprehension to agent software in further detail.

Section 2.3.2 discusses the use of model-checking as another approach for

software comprehension since the model being checked can be thought of as a formal design specification or a model of the user's comprehension of the system. The model 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. However, verifying and maintaining the model as the software changes has rarely been addressed. This research takes the model-checking approach for modeling the user's understanding of the software's behavior and the reverse engineering approach for verifying the model against the actual implementation.

Finally, Subsection 2.3.3 presents some arguments for extending software comprehension, and Subsection 2.3.4 proposes agent software comprehension to addresses the limitations of traditional software comprehension and model-checking.

2.3.1 Traditional software comprehension

Traditional software comprehension (or reverse engineering) has been used to study how a program performs certain operations, to improve performance, to fix bugs, to identify malicious content (e.g., viruses), or to adapt a program written for use in one domain for use in another domain. A recent application of reverse engineering is roundtrip engineering. Roundtrip engineering demands that the design and implementation are always consistent by iteratively reverse engineering the implementation and comparing it with the design [Systa, 2000]. A similar process is performed in this research, where the results of reverse engineering the implementation is used to build and verify with the background knowledge to ensure accuracy. As of yet, the use of software comprehension for agent software has not been

explored. In response, this research proposes to apply a software comprehension approach for comprehending agent behavior.

Software comprehension is motivated by the fact that implementing a system is not always done right the first time and that maintenance is a major (financial and temporal) cost of software engineering. The software may need to be (1) verified to ensure that the implementation is behaving as it was designed to behave; (2) maintained to fix bugs or make modifications; or (3) redesigned/evolved to improve performance, reusability, or extensibility (among other reasons). In order to do these things, an understanding of the existing implementation is required and is attained using software comprehension. Such an understanding can be acquired by reading design documents, examining source code, and analyzing execution traces of the system. This section describes a number of tools to aid the human user during software comprehension.

The goal of software comprehension is to extract the design (and its intentions) expressed as diagrams from the source code (or code execution). Computer-Aided Software Engineering tools (which were originally used to help with the initial construction of large software systems by employing graphical editors, consistency checkers, and code generators) can now automatically reverse engineer diagrams (e.g., structure charts, dataflow, entity-relationship, and component dependency diagrams) from the source code. These reverse engineering (RE) tools aim to offer the user a better understanding of the structure and relationship among software components of the system, and involves creating abstractions of the system design [Chikofsky and James H. Cross, 1990]. 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 [Agrawal et al., 1998; Finnigan et al., 1997]. The most common diagram is the structure chart, indicating class inheri-

tance, class methods, and their arguments. Other diagrams include dataflow, entity-relationship, and component dependency diagrams. Other tools used to understand the implementation include Gen++ [Devanbu, 1992], GRASPR [Wills, 1993], and DESIRE [Biggerstaff et al., 1994]. Because these tools analyze the source code, the resulting diagrams are at a low abstraction level and reference constructs in the source code.

Comprehending an agent means understanding what led to the agent's actions, or more abstractly, what motivated the agent's behavior. The low-level diagrams resulting from current comprehension tools do not reflect an agent's motivations. Agent concepts (e.g., beliefs, events, and goals) influence which actions the agent decides to take. To determine which factors from the agents and from the environment caused an agent's actions (as shown in Figure 1.1), the diagrams reflecting an agent's motivations should be modeled in terms of agent concepts (e.g., beliefs about the current environment, roles the agent is currently assign to, and intentions the agent is trying to satisfy), which are at a higher abstraction level relative to diagrams resulting from source code analysis.

RE tools tend to produce a large amount of data for the user to interpret and absorb. To deal with this, SoftSpec allows users to query a relational database of automatically gathered information about the software architecture [Bruening et al., 2000]. Alternatively, using a graph-oriented approach, the GUPRO (Generic Understanding of Programs) toolset transforms source code into graphs according to a defined concept model [Kullbach and Winter, 1999]. 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 [Lange et al., 2001]. This dissertation employs graphs to visualize the system's behavior for the user.

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 [Sim and Storey, 1999]. 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, their tool can only identify pre-defined patterns [Schauer and Keller, 1998]. This research also uses abstractions to help bring the ideas manifested in the implementation closer to familiar design concepts, thus, aiding the comprehension of the overall system behavior, independent of implementation constructs. Since agent behavior varies across different agent systems and domains, the tool from this dissertation must be flexible enough to perform reasoning for most (if not all) agent systems, despite varying agent behaviors.

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 [Koskimies et al., 1998]. such as Tango [Stasko, 1990] (which visualizes changes to values in data structures during execution) and ISVis (Interaction Scenario Visualizer) [Jerding and Rugaber, 1998] (which captures runtime elements and interactions in message sequence charts). Other tools, such as Hindsight and Lemma, can create flow charts and control-flow

diagrams [Hindsight, 2004; Mayrhauser and Lang, 1999]. Moreover, results of dynamic analysis can also be represented as use cases and recurring behavioral patterns, which brings the implementation closer to the design [Jerding and Rugaber, 1998; De Pauw et al., 2002; Lucca et al., 2000]. Since dynamic agent behavior is central to comprehending agent systems, the Tracer Tool focuses on elucidating reasons for agent behavior, rather than on the structural composition of the agent or system architecture.

RE tools only produce representations of the implementation and have no model of the user's comprehension. It is the user's responsibility to digest the RE results (e.g., diagrams, charts, and database entries). Since RE tools do not model what or how much the user understands, they cannot provide feedback to the user about the user's comprehension. For example, RE tools cannot alert the user about a misunderstanding about an agent's behavior. However, in model-checking, the user's understanding of the implementation is translated to a model, which can be automatically checked for specified properties. Thus, model-checking tools have a model of the user's comprehension of the system and can aid in verifying the model. Though useful due to its capacity to search the entire state-space, model-checking in general does not deal with the accuracy of the model with respect to the actual system. Additionally, the model must be made simple enough to be handled by the model-checker. The next section briefly describes the limitations of model-checking.

2.3.2 Model-checking

Model-checking is used to formally verify behavioral properties (e.g., deadlocks and assertion failures) of software systems. Model-checking tools, such as Spin [Holzmann, 1997] and Bogor [Robby et al., 2003], 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. Note: models the user's comprehension

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) [Visser et al., 2003], 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 [Edmonds and Bryson, 2004]. Applying the model-checking approach, this research models what the user comprehends. To extend the model-checking approach, this research also empirically verifies the model (called K) against the actual implementation using a RE approach and maintains the model so that it accurately represents the implementation.

2.3.3 Arguments for agent software comprehension

Agent software comprehension (ASC) extends traditional software comprehension to include not only the representation of the implementation but also the reasoning process for comprehending the implementation; specifically, reasoning about why agents decided to perform certain actions. As agent software becomes more sophisticated and complex, software comprehension becomes a more important topic for agent-oriented software engineering. The following are arguments for applying software comprehension to agent software.

Sophisticated agents Comprehending agent behavior is complicated by factors in the environment (e.g., dynamism, uncertainty, and nondeterminism) and sophisticated agent features. It is these features that make it difficult to (manually or automatically) comprehend the implemented agent system. The following describes agent features that make agent behavior difficult to comprehend and that must be considered in this research:

- Autonomy- An agent can make decisions without intervention based on its own internal state. The developer needs to know how the agent makes decisions in each of the agent's states.
- Proactivity- An agent takes initiative in order to achieve its goals. The developer needs to know when and why the agent takes initiative and for what goal.
- Reactivity- An agent must respond appropriately in a timely manner to changes in the environment. The developer needs to know what environmental changes affect the agent.

- Rationality- An agent must make decisions that are in its own best interest given its beliefs about its environment. The developer needs to know the beliefs (which may not be ground truth) relevant to the agent's reasoning process.

- Sociability- Agents may communicate to share beliefs and coordinate to achieve goals. Interaction can affect the internal state, beliefs, and decisions of participating agents.

To generalize, agents have control over their own behavior, which means the developer needs to have access to the agents' internal decisions to comprehend the whole system. To comprehend the system, the developer may also need to know the agent's beliefs at the time, the environmental state, and the situation the agent believes it was in. Acknowledging the difficulty of comprehending the implementation and the potential sophistication of agent features, this research aims to apply agent concepts beyond the design and implementation phases of software engineering and toward the testing and maintenance phases.

Unpredictability of agents Jennings explains two drawbacks associated with an agent-based approach: (1) the patterns and outcomes of interactions are inherently unpredictable due to agent autonomy and environmental uncertainty and (2) predicting system behavior based on its constituent components is extremely difficult because of the possibility of emergent behavior [Jennings, 2000]. Currently, agent developers circumvent these two drawbacks by adopting preset organizational structures, limiting nondeterministic environmental effects on the agents, and decreasing the number of factors that can affect agent behavior. Jennings stresses a need for a better understanding of how agent interaction affects an individual

agent's behavior. Since agent behavior cannot be predicted from only the design, being able to explain agent behavior resulting from the implementation is very important in agent development.

High software maintenance costs Despite efforts of automating the understanding process, program comprehension continues to be largely a manual task due to the significant amount of knowledge and analytical power required [Rugaber, 1995]. The comprehension activities of reading the documentation, scanning the source code, and understanding changes are reported to take 47% and 62% of the time spent on actual enhancement and correction tasks, respectively [Fjeldstad and Hamlen, 1983]. These maintenance tasks have been estimated to expend 50% to 75% of the time in the software lifecycle [Boehm, 1981]. For agent-based software, in which complexity is a common property, maintenance is a challenging problem. The implication is that if agent software development is to be improved, software maintenance needs to be improved, and if maintenance is to be improved, then the process of comprehending software programs should be facilitated. This research presents an automated reasoning process used to facilitate agent software comprehension.

Low end-user trust in agent decisions Agents are often used as assistants that make suggestions to the human user, who makes the final critical decision. One way to gain the user's trust in agent decisions is to enable the agent to explain its actions upon request or in unusual situations. In this manner, the user begins to understand why the agent behaves as it does and, over time, places more trust in the agent's decisions. In complex situations where the user is unsure of the appropriate actions to take, the user can rely on the agent to make a rational decision on behalf of the user and have some confidence in the agent's decisions.

Advancing agent research and development If it was possible to comprehend the implemented agent system at the agent design level, then there are a number of benefits that can be obtained and leveraged in future research. First, the implementation can be verified with respect to the design, and thus, help developers debug the agent system. Second, by examining agent behaviors, the motivation for those behaviors can be discovered and agent behaviors can be improved. Third, design patterns for agent behavior can be generalized and reused to reproduce agent behaviors in future agent systems. Fourth, agent concepts will not be limited to modeling the problem and designing the solution but also for talking about the implementation.

2.3.4 Agent software comprehension

Comprehending the software is important because of the unpredictability and dynamics of interacting autonomous agents. As agent software becomes more sophisticated and complex, software comprehension becomes a more important topic for agent-oriented software engineering. As previously described, traditional software comprehension tools and techniques are inappropriate for software comprehension of agent software. Agent software comprehension extends traditional software comprehension to include not only the representation of the implementation (e.g., structure charts, dataflow diagrams, and component dependency diagrams) but also the reasoning process for comprehending the implementation (i.e., explaining agent software behavior). While traditional software comprehension answers "what is happening in the implementation", ASC takes a step further to facilitate comprehension by answering "why is it happening in the implementation".

Traditionally, the reasoning process for comprehending software behavior is performed manually. First, events, agent actions, etc. are observed from running

the implementation. Then, those observations are interpreted using models of how the particular agent is expected to act (e.g., state charts). These expected behavior models are usually derived from design documents, experience with the agent system, and intuition about agent systems and are referred to as *background knowledge*. Since there is no direct way to measure how much the user comprehends, a person's comprehension of a subject is indirectly measured by how much the person can explain about the subject because the process of creating an accurate explanation demands correct comprehension of the system. Explanations bridge the gap between expected and actual behavior – specifically, the explainer's background knowledge of the software's behavior and observations of the software execution (e.g., agent actions). Thus, explanations can be very important in designing, debugging, and trusting agent behavior.

Unfortunately, ensuring accurate explanations is difficult because the implementation evolves over time and there are many factors that can influence agent behavior. First, since comprehending the behavior of the implemented system relies on how accurately the background knowledge represents the implementation, the representative accuracy of the background knowledge must be maintained as the implementation changes. The second problem in manual explanation generation is that an explanation may be too difficult to conceive due to the sophistication (e.g., in reasoning or agent interaction) of the agent system or the amount of observed data to consider. In response to these difficulties, this research proposes an automated approach to agent software comprehension that can handle large amounts of observation data and can automate the generation of explanations to aid the user in comprehending the system as the implementation evolves over time.

In summary, ASC involves (1) representing the implementation using agent-oriented structural and behavioral models and (2) reasoning over those representa-

tions to facilitate the user's understanding of the system. The ability to explain the system's behavior demonstrates comprehension of the system, answering both what is happening and why it is happening. Due to the issues with manual agent software comprehension, this research aims to automate agent software comprehension using the foundations of abductive reasoning.

2.4 Abductive Reasoning (for RQ2)

Being able to explain why an individual agent behaves as it does is an essential part of comprehending multi-agent systems. Agent interactions and environmental uncertainty make predicting agent behavior difficult. Currently, agent software comprehension is done manually, by associating run-time observations of the implementation to models of behavior from the design. This research aims to automate this the explanation process by using abductive reasoning. In general, abductive reasoning (or abduction) is "inference to the best explanation", a pattern of reasoning that occurs in such diverse problems as diagnostics, scientific theory formulation, and natural-language processing [Josephson and Josephson, 1994]. Abduction produces plausible explanations by using background knowledge (e.g., behavioral models) to link causes (e.g., goals and events) to observed effect (e.g., actions); thus, abductive reasoning would be beneficial in comprehending agent behavior. Abduction has been used for numerous AI applications, including natural-language processing [Stickel, 1989], plan recognition [Appelt and Pollack, 1992], diagnosis [Console et al., 1991], prediction, classification [Mooney, 1997], and case-based reasoning [Leake, 1993].

Though fruitful, these logical formulations have brought about several issues (e.g., the adequacy of deductive implication in representing causality) and disregarded some fundamental aspects (e.g., surprise) of Peirce's original formulation

of abduction. Aliseda-Llera believes that "these [logical] approaches have paid little attention to the elements of [Peirce's] formulation and none to what Peirce said elsewhere in his writings" [Aliseda-Llera, 1997]. Considering these issues along with promising ideas from existing abduction studies, this research designs an explanation generation mechanism based on abductive reasoning. In particular, this research represents causality (and other relationships helpful in an explanation of agent behavior) using semantic networks and identifies surprises as inconsistencies between the background knowledge and interpretations of the implementation execution. Contributions of this research include a method to build the background knowledge required for generating explanations and an approach to generate explanations for comprehending agent behavior despite complexities of agent systems.

Subsection 2.4.1 reviews Peirce's formulation of abduction, and Subsection 2.4.2 discusses issues with existing abductive reasoning research and how the design of the explanation mechanism addresses these issues.

2.4.1 Peirce's formulation of abduction

In Peirce's words, abduction is "inference to the best explanation" [Peirce, 1931-1958]. Abduction was Peirce's answer to a fundamental philosophical question of scientific discovery – how is synthetic reasoning (reasoning that creates new ideas) possible? He defines abduction as the process of forming (by employing some "guessing instinct") and selecting (by preferring "the simpler hypothesis in the sense of the more facile and natural, the one that instinct suggests") a plausible hypothesis for explaining observations. As a process of finding causes of observed effects, abduction is "the basis of interpretive reconstruction of causes and intentions, as well as of inventive construction of theories" [Peirce, 1931-1958].

Abduction is initiated as a consequence of some surprising observed fact or

phenomenon. The notion of surprise is relative with respect to some background knowledge that provides expectations. "The surprising fact, C, is observed; But if A were true, C would be a matter of course, Hence, there is reason to suspect that A is true" [Peirce, 1931-1958]. Abduction aims at discovering a plausible hypothesis, which is selected or invented, using the prior background knowledge. The surprising fact generates doubt in the background knowledge (what Peirce calls beliefs). When abduction is triggered by a surprise, the abductive process attempts to explain the surprising fact and calm the state of doubt (rather than nullifying doubt completely since an abductive explanation is only a hypothesis that needs to be confirmed) [Aliseda-Llera, 1997].

A surprise is relative to some background knowledge (current beliefs about how the world behaves), and there are two types of observations that cause surprise. Novel observations are completely new observations that are found to be consistent with the background knowledge, and anomalous observations are those that conflict with background knowledge. For example, if the background knowledge was flipping a switch up causes the light to turn on, then a novel observation could be the radio turns on and an anomalous observation could be the light turns off. Explanations resulting from these surprises can cause the background knowledge to be updated so that such observations are consistent with expectations.

In summary, Peirce tributes abduction as the creative process in scientific discovery and identifies the notion of surprise and its affect on the background knowledge. The notion of surprise is not fully captured in current abductive formulations. In this research, inconsistencies between the background knowledge and implementation executions are treated as surprises. This research borrows ideas from belief maintenance [Pagnucco, 1996] to incorporate surprises into abductive reasoning. The following describes this in further detail.

Incorporating surprises A surprise can cause the background knowledge (or background theory) to change, depending on what type of observation cause the surprise. For the sake of clarity and formalization, the following discussion is exemplified in logic. An observation m is a surprise if the current background theory K does not entail the observation. For a novel observation, since m is consistent with K, then an explanation e can be added to K so that m is deducible from K.

Given $K \cup m \nvDash \bot$, find e such that $K \cup e \models m$.

This is also known as *abductive expansion* because a new explanation is added to the background theory. For an anomalous observation, since m is inconsistent with K (i.e., K entails the negation of m), K can be revised to become K' so that $K' \cup e$ entails m.

Given $K \cup m \models \bot$, find e such that $K' \cup e \models m$, where $K' \nvDash \neg m$.

This is known as *abductive revision*. Commonly, statements in K that contribute to the negation of the surprise are removed (a.k.a., abductive contraction) so that inconsistency does not occur and abductive expansion is performed to produce the revised K', which is used to generate a consistent explanation. However, if there is enough confidence in the existing background theory, the anomalous observation may be ignored and identified as an error (in our case, an implementation error).

In this research, surprises are addressed in RQ1 as the background knowledge is being constructed and verified against the implementation's behavior. When surprises occur, the Tracer Tool can suggest how the background knowledge can be updated or generate an explanation for the user to explore the reason for the surprise observation. In addition, explanations of non-surprising observations can be used to verify and further support beliefs in the background theory, making those beliefs more grounded and less likely to be removed later on.

2.4.2 Issues with approaches to abductive reasoning

Though most approaches to abduction are formulated in logic, there is no unique formalism for abductive reasoning. Thagard and Shelley criticize two formal logical models of abduction for their inadequacy as general characterizations of abduction [Thagard and Shelley, 1997]. However, logical formulations captured the central aspects of abduction and made possible computational complexity analysis and precise comparison with other reasoning model – Bylander et al. analyzed the computational complexity of abduction and showed that it is NP-hard [Bylander et al., 1991], and Konolige related abductive reasoning with Reiter's model of diagnosis [Konolige, 1991]. Thagard and Shelley affirm that "the results must be viewed as relative to the abstract model of abduction proposed" and identify several issues that must be considered in a formulation of abduction. Three relevant issues from [Thagard and Shelley, 1997] are reviewed and discussed below – explanation is not deduction; hypotheses may be revolutionary; and completeness is elusive.

Explanation is not deduction (Probability and causality) Konolige's formulation of abduction uses deductive inference to deduce observations from plausible hypotheses. Though deductive models provide a good approximation for many explanations, deductive models fail to provide "either necessary or sufficient conditions for explanation" [Thagard and Shelley, 1997]. Konolige's formulation of abduction as a deductive formulation excludes domains where effects probabilistically occur.

It is important to note that not all inferences of the form "(if a b) and b, therefore a" are considered abductions because not all such inferences create an explanation that relates cause with effect. For example, the statement "(if AlexIsInRoomA

AlexIsSick) and AlexIsSick, therefore AlexIsInRoomA" is not an explanation because AlexIsInRoomA is not the cause of AlexIsSick. The actual form of abduction is "(cause a b) and b, therefore a is explanation" [Charniak and McDermott, 1985]. The notion of causality is essential for explanations, but causality cannot be expressed using deduction.

Alternatively, Hempel discusses statistical explanations, where the observations follow probabilistically, not deductively, from the background theory and the explanation [Hempel, 1965]. Poole argues that Bayesian conditioning (used to find causes for observed effects) can be seen as abduction in probabilistic logic programs [Poole, 1993]. Bayesian networks is an established research field and offer a general method (i.e., Bayesian conditioning) for finding explanations. However, the representation of nodes in the Bayesian network is propositional, thus they lack the rich representations of predicates used by logical approaches.

This research addresses this issue by explicitly modeling causal relations (as well as other relations) in the background theory as [Charniak and McDermott, 1985] suggests. Semantic networks will be used to model relationships among agent concepts. As a result, the proposed explanation generator can distinguish causality from other relations. In addition, reasoning can be enhanced using non-causal relations (e.g., compositional, hierarchical, and temporal relations).

Association-based (or semantic) networks have long been studied as a knowledge representation method in A.I. [Quillian, 1968; Findler, 1979]. Nodes represent objects, concepts, and events, and links represent interrelation or association between nodes. "This is in contrast to statistical or rule-based models where relationship between entities are often implicitly embedded in conditional probabilities or conditional rule and often interwoven." Semantic networks' expressiveness makes them suitable for representing causal associations between causes and ef-

fects. Unlike statistical pattern classification and rule-based systems which have firmly established underlying theories (i.e., probability theory and deductive logic, respectively), association-based abductive models have lacked theoretical base to support them until Parsimonious Covering Theory [Peng and Reggia, 1990].

Hypotheses may be revolutionary (Inconsistent observations) Konolige's formulation requires that a hypothesized explanation be consistent with the background theory (or background knowledge). This requirement does not allow for abductive revision, where anomalous observations conflicting with background theory result in changing the background theory. "While this requirement may be acceptable for mundane applications, it will not do for interesting cases of belief revision where the introduction of new hypotheses leads to rejection of previously held theories" [Thagard and Shelley, 1997].

In this research, when an anomalous observation occurs, either the implementation is incorrect with respect to the background knowledge K, K incorrectly predict the behavior of the implementation, or both. As a result, one or both are modified so that K is consistent with the implementation's behavior.

Completeness is elusive (Imperfect background theory) Bylander et al. require that the background knowledge be complete with respect to all observed data [Bylander et al., 1991]. Thagard and Shelley believe that "this is a laudable goal, but does not justify building completeness into the definition of an explanation, since the goal is so rarely accomplished in realistic situations" [Thagard and Shelley, 1997]. Even when given a perfect background theory, an abductive reasoner can never include all factors that may potentially be relevant in a situation. This qualification problem [McCarthy, 1980] states that any hypothesis

(car starts when ignition is turned on) must in principle depend on an infinite set of conditions (no potato in tail pipe) [Leake, 1993].

This research does not strive for completeness since the background knowledge K will usually require modification as the implementation evolves over time. Any incompleteness in K is handled by the notion of surprises as described in the previous subsection. In accordance with [Dupre and Rossotto, 1995], this dissertation avoids unnecessarily detailed and involved explanations and takes advantage of abstractions.

2.5 Related Work

In accordance with Edmonds' emphasis on the need for empirical analysis of agent behavior, such as scenario-based analysis and field testing [Edmonds and Bryson, 2004], this research analyzes the implemented system rather than a model of the implemented system. The tools presented in this dissertation is able to help verify agent behavior, detect anomalous observations, and assist the user in diagnosing the causes of such anomalies. Similar research by Kalech and Kaminka [Kalech and Kaminka, 2003] focuses on diagnosing inter-agent failures, where agents have models of other agents and uncertainties must be considered. Using a different approach, the tools in this paper take advantage of the fact that actual values can be recorded as true observations as the system is executing. In effect, the only uncertainties are those related to the user's comprehension of the implemented system, which are addressed by the Tracer Tool's ability to suggest updates to the user's comprehension.

In Garcia-Molina et al., several tools are suggested for the collection of trace files (consisting of important low-level events and corresponding time param-

eters) in a distributed system [Garcia-Molina et al., 1984]. In our approach, traces of observed agent activity are recorded using agent concepts, which are at a level of abstraction familiar to designers, developers, and end-users of the agent system. Liedekerke and Avouris were one of the first to address the problem of debugging multi-agent systems, offering a tool that visualizes different aspects of the system at the agent concept level [Liedekerke and Avouris, 1995]. However, their approach does not consider what the user understands (the background knowledge) in order to identify unexpected or undesired behaviors from the user's perspective. The visualization/debugging tool suite provided by Ndumu et al. also does not account for what the user understands but instead collates, fuses, and presents information from disparate data sources [Ndumu et al., 1999]. The approach taken by this research is to leverage background knowledge to (1) make anomalous behavior easily detectable as shown by the visualization of the Tracer Tool and (2) investigate observations by generating explanations in terms of familiar agent concepts. Additionally, the integrated ability of the TTL Checker to check system properties further aids in identifying anomalous behavior.

The motivations and approach of this dissertation has similarities with research focusing on the explanation of expert systems by Swartout and Moore [Swartout and Moore, 1993]. As summarized in [Memmel and Roth-Berghofer, 2004], Swartout and Moore stress the importance of fidelity (having accurate explanation of the knowledge-base system), understandability (the use of abstraction and terminology), sufficiency (having enough knowledge to generate explanations), and efficiency (the explanation facility should not degrade run-time performance of the expert system). These features were considered and addressed in this dissertation by using abstract agent concepts, building an accurate and sufficent background

knowledge, and providing a non-intrusive approach used to gather observations of run-time data.

Though similar, Swartout and Moore's work to explain expert systems differs from this dissertation research in the following ways. First, their research requires the expert system to be developed or adapted to the EES (Explainable Expert Systems) framework ([Swartout et al., 1991]) to support explanations. The logging approach employed in this dissertation does not require any changes to the subject system, only logging code (that does not interfere with the software architecture or behavior) needs to be inserted. Second, their tool, called EXPECT, analyzes the rules and knowledge base of the actual expert system. This suggests that EXPECT must be able to parse the language and architecture of the expert system. Since this dissertation does not analyze the implementation directly, the Tracer Tool does not depend on the programming language or architecture of the implementation, as long as agent concepts can be identified and logged. Thirdly, EXPECT relies on the backchaining of rules and the reformulation of goals to activate those rules. In this dissertation, the causal relation between observations are separated from the implementation. These relations are stored in the background knowledge (which represents the user's comprehension) and are formed as the user's knowledge of the system grows with the help of the Tracer Tool.

Memmel discusses that one of the difficulties of generating explanations is that the user may not understand the explanation because the explanation system does not have a model of the user's knowledge [Memmel and Roth-Berghofer, 2004]. This dissertation addresses this issue by explicitly modeling what the user comprehends about the implementation in the form of background knowledge. With this approach, the generated explanations will use concepts defined in the background knowledge and the Tracer Tool is able to provide feedback to the user,

such as suggestions to update the user's comprehension or visualizations of actual system behavior with respect to the user's expected behavior.

Studies by Doyle *et al.* points out that novice users prefer higher-level explanations mixed with background information and low-level elements, whereas experts, who are more interested in anomalies, tend to prefer low-level explanations [Doyle et al., 2003]. This dissertation aims to target a wide range of users, from designers to end-users. As a result, the generated explanations employ abstract agent concepts and references specific observations from system execution. Each observation and its associated data can be mapped to some location in the source code, thus, facilitating traceability to the implementation.

This dissertation adds to the numerous studies on the use of explanations. The differences lie in the modeling of the user's comprehension, the practical approach for recording actual system behavior (which de-couples the Tracer Tool from the implemented system), and the automated reasoning used to generate visualizations and explanations of system behavior.

2.6 Summary

Software comprehension is essential for developing, maintaining, and redesigning complex software, such as agent-based systems. Software comprehension (a.k.a. reverse engineering) is performed by the human user to understand the structure and behavior of the software. To emphasize the contributions of this research, this chapter reviews background literature and work related to this research topic. The current suite of reverse engineering 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 di-

agrams 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/or events that occur during execution) and leave it up to the user to interpret and reason about the software's behavior. Additionally, 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 system" [Lange et al., 2001].

Another approach for software comprehension is for the user to specify a formal behavioral model of the system and verify certain properties using model-checking tools, which focuses on testing and verification of software behavior. The behavioral model 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. 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 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, reverse engineering and model-checking have their limitations 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 must consider their own goals, resources, and constraints while making

decisions and negotiating with other agents. In trying to comprehend the system, the user may also have to consider interactions among agents and how the agents handle the uncertain and unpredictable environment. This research developed a method and tool that builds on the ideas from the reverse engineering and model-checking approaches and extends the state-of-the-art to better assist the human user (of various skill levels) in comprehending agent-based software.

In Section 2.4, the original conceptualization of abductive reasoning and issues with existing research on abductive reasoning were presented. This research focuses specifically on the issues of (1) incorporating surprise into the abduction process (2) formulating the background knowledge and the abduction process, (3) dealing with imperfect background knowledge, and (4) selecting the correct hypothesis.

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.

Chapter 3

APPROACH

explanation (n): (1) the act of explaining, expounding, or interpreting; the act of clearing from obscurity and making intelligible; (2) that which explains or makes clear; ...

[1913 Webster's Revised Unabridged Dictionary]

This chapter describes the approach taken to answer each research question presented in Chapter 1. Section 3.1 describes the general formulation of the approach. Section 3.2 introduces the representation of agent concepts, background knowledge, and observations used throughout this research.

The next chapters, Chapter 4 and 5, expound on the approach and present the Tracer Tool along with experiments for Research Questions 1 and 2 respectively.

3.1 Formulation of the Approach

The objective is to produce accurate explanations of agent behavior in the implementation, which is possible only if the background knowledge accurately represents the implementation. The following describes the formulation of the approach taken by this research to ensure the representativeness of the background knowledge and to explain agent actions. The notation used by the following equations is that functions begin with a lowercase letter (e.g., $interpret(K, O_s)$)

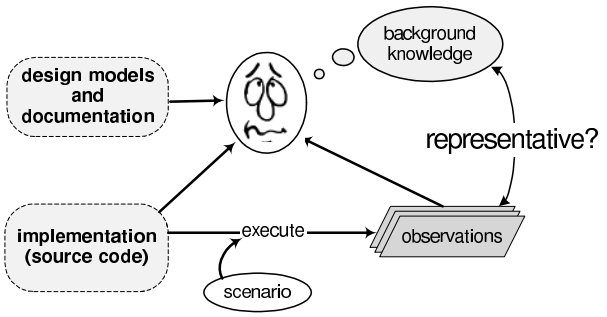


Figure 3.1: Manual software comprehension

and predicates, which return true or false, begin with an uppercase letter (e.g., $Consistent(K, N_s)$).

Research Question 1 (RQ1) asks how background knowledge is created such that it accurately represents the implementation's behavior. Traditionally, background knowledge K is constructed by the user and describes how the agents are expected to behave in terms of the agent concepts listed in Section 2.2. Background knowledge K represents the user's comprehension of the system, which is commonly derived from many sources, such as specifications of the design, experience with the implementation, and intuition from presentations. In model-checking, K is a model that is to be checked and it is manually specified by the user. As illustrated in Figure 3.1, the manual procedure for building comprehension can be expressed as

$$K' = update_{manual}(K, D, I, O_s)$$

where K is the previous background knowledge, D is the design models and documentation, I is the implementation expressed in source code, and O_s is a set of observations resulting from executing the implementation I in some scenario s:

$$O_s = observe(execute(I, s)) \quad [3.1]$$

Note that since comprehension is an iterative process, construction of K' involves modifying and updating the previous background knowledge K. To build up comprehension, the user has the tedious task of gathering, organizing, and relating the data from the design D, the implementation I, and the observations O_s. Additionally, as the implementation changes over time, the user must verify that the expected behavior expressed in K is representative of the actual behavior from the implementation.

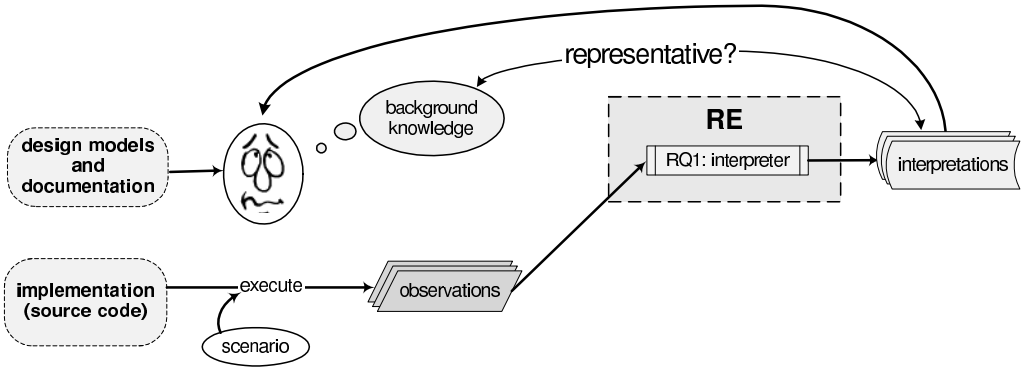


Figure 3.2: Reverse engineering approach

As seen in Figure 3.2, the reverse engineering approach helps the user by analyzing O_s to produce interpretations N_s, which consists of models representing actual system behavior:

$$N_s = interpret_{RE}(O_s)$$

However, the user still has the task of ensuring that K accurately represents N_s (i.e., that the user's comprehension of the system is accurate with respect to the actual system). When a user tries to comprehend agent behavior in the implemented system, the user is essentially building an interpretation by observing and examining

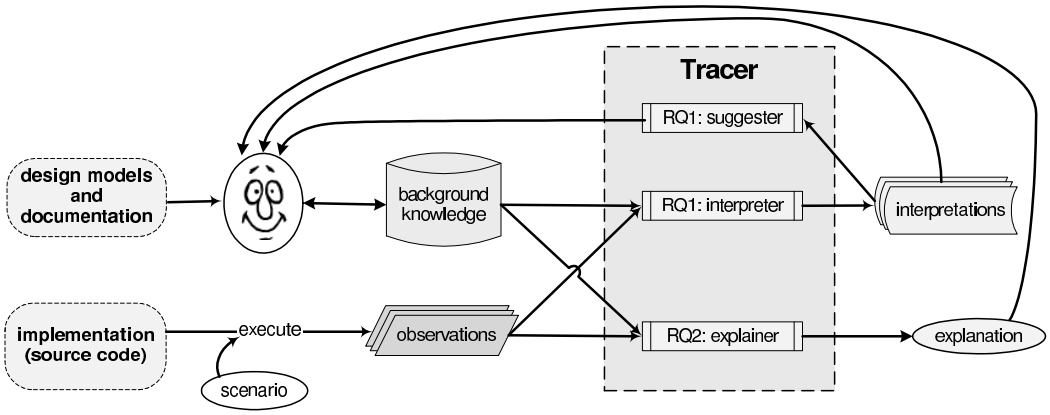


Figure 3.3: Overall approach of this research

agent actions, communicated messages, environmental events, and any other run-time data that can be acquired from the implementation. As shown in Figure 3.1 and Figure 3.2, background knowledge about the expected behavior of the implemented system is required to build the interpretation. The problem that prevents further automation is that there is no explicit representation of the background knowledge that exists in the user's mind.

To aid the user in software comprehension, this research answers RQ1 by automating the tasks of interpreting the observations *with respect to* K (and in the process, verifying K) and suggesting modifications to K (see Figure 3.3). This is possible by explicitly modeling the user's background knowledge K and using it as input to the Tracer Tool. Thus, the new $update$ function is

$$K' = update(K, D, N_s, k) \quad [3.2]$$

where interpretation N_s are derived by mapping the observations O_s to agent concepts in K:

$$N_s = interpret(K, O_s) \quad [3.3]$$

and suggestions k consists of relations that can be added to the background knowledge K:

$$k = suggest(N_s) \quad [3.4]$$

Since background knowledge K should accurately model the user's comprehension, the user remains in control of K and must confirm all suggestions before K is modified. However, the user no longer needs to directly analyze the observations O_s from the implementation execution or verify that K accurately reflects the implementation's behavior, as these tasks are automated by the Tracer Tool. With the interpretations N_s readily available, the user can modify K as seen fit. Through each iteration of building up K, the Tracer Tool verifies K against the observations O_s in case the user introduced errors into K. If the implementation's behavior changes and differs from K, new or inconsistent behaviors can be readily detected in N_s and suggestions for updating K are generated by the Tracer Tool.

Formally stated, the background knowledge K is representative of the implementation I if and only if K is consistent and complete with respect to interpretations N_s for each execution scenario s in a set of scenarios S:

$$Representative(K, I, S) \iff \forall s \in S \, (Complete(K, N_s) \wedge Consistent(K, N_s)) \quad [3.5]$$

where $Complete(K, N_s)$ is true if there is no suggestions (i.e., $k = \emptyset$) for updating K and $Consistent(K, N_s)$ is true if there are no contradicting behaviors. Section 4.4.2 lists how the different types of completeness and consistency errors are detected. Ideally, S would be a *complete* set of scenarios covering all possible situations the implementation would encounter. Since this is not generally feasible, a set of scenarios that covers a reasonable number of situations is assumed to be given.

Once background knowledge K has been checked for representative accuracy over the chosen set of scenarios S, Research Question 2 (RQ2) asks how K can be used to accurately explain a manifestation m (e.g., an agent action) that occurred in a specific scenario s using observations O_s (resulting from the scenario in which m occurred). An explanation ϵ consists of a subset of observations from O_s and relations among those observations that contributed to (i.e., caused or influenced) the occurrence of m. The relations are derived from K, which defines relations among agent concepts. Thus, explanation generation involves mapping observations to agent concepts and following the relations (backward) from m to observations that caused m. Based on the approach illustrated in Figure 3.3, an explanation ϵ for manifestation m can be generated using K and O_s. If an interpretation N_s of the scenario exists, the same explanation can be generated faster using N_s because $interpret(K, O_s)$ has already done the work of mapping and relating the observations.

$$\epsilon = explain(m, K, O_s) = explain(m, N_s) \quad [3.6]$$

Since background knowledge K is represented using agent concepts, the generated explanations will be expressed in terms of the same high-level agent concepts, understandable by anyone with a general knowledge of agents.

In summary, since the accuracy and validity of the explanation is dependent on how accurately the background knowledge K represents the implementation I, accuracy between K and I is addressed in Research Question 1 (RQ1). Given an answer for RQ1, Research Question 2 addresses how explanations are generated using background knowledge K and observations O_s from the system execution. To clarify the approach to answer the research questions, the next section describes the representations used for agent concepts, K, k, O_s, and N_s.

3.2 Representations

All representations used in this research are based on agent concepts. Agent concepts and relationships between agent concepts are used in background knowledge K and suggestions k. Observations O_s are instantiations of agent concepts that has additional run-time data, such as actual belief values and timestamps. Interpretations N_s consist of observations and relations among observations (as defined in K). Finally, an explanation ϵ is a subset of observations and relations from N_s that is structured as a tree, where child nodes are factors that explain the occurrence of the parent node. Note that though agent concepts are pervasive in all the representations, the set of agent concepts can be easily expanded with little modification to the other representations. The following describes the representation used for agent concepts, observations, and background knowledge. Interpretations, suggestions, and explanations are better described in the context of answering Research Questions 1 and 2 (in Chapters 4 and 5).

3.2.1 Agent concepts and Observations

The idea for agent concepts is to provide a guideline for observing and analyzing agent behavior without being bogged down with implementation-specific details. The set of agent concepts include $goal$, $belief$, $intention$, $action$, $event$, and $messages$ (see Section 2.2 for a general description). This set is adequate for explaining agent behavior at a high level of abstraction. In practice, developers may want to include other application-specific concepts that help to better understand the agent system.

Each agent concept contains a set of attributes (as shown in Figure 3.4). These attributes are essential for many of automated tasks performed by the Tracer Tool, such as suggesting possible relations between agent concepts, as the attributes

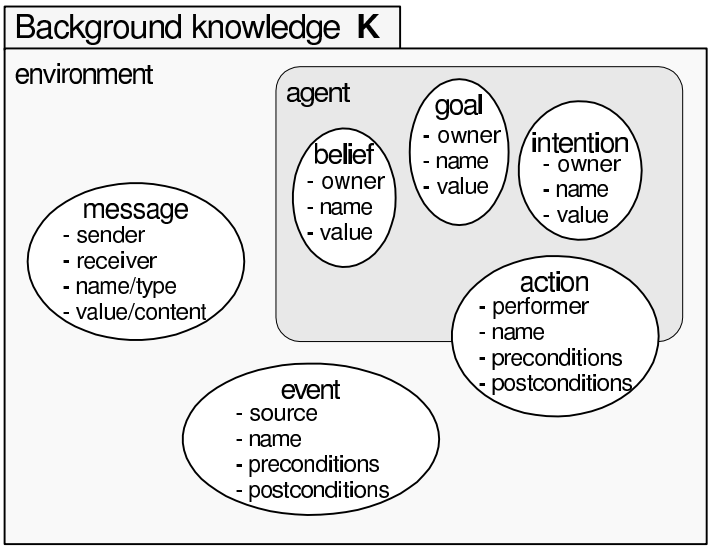


Figure 3.4: Agent concepts and their pre-defined attributes

are used to distinguish and relate observations in O_s. Each attribute is composed of a name and a value. Each agent concept includes some pre-defined attributes, such as `name`, `value`, or `preconditions`. Usually, the user adds additional attributes for application-specific details. The value of each attribute is either a single value or a list of name-value pairs.

Some attribute values may be static such as for the attribute `name`, while most attributes are assigned at run-time such as `value` or `timestamp`. When values are assigned to attributes of an agent concept during run-time, the concept is called an observation. Note that agent concepts only exist within the user's background knowledge K because those concepts represent abstract constructs that help describe agent behavior.

3.2.2 Background knowledge

The background knowledge (or 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 best viewed as a semantic network (or more closely, a casual network [Sowa, 1984]) where nodes denote agent concepts and directed arcs denote relations between agent concepts. These relations represent direct 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 directed arcs that point to that concept. The agent concept at the origin of the arc potentially caused the agent concept at the destination of the arc. Continuing the example, the aforementioned event would have an incoming relation originating from the action – meaning the action potentially caused the event to occur. 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.

A semantic network was chosen to represent background knowledge because it can be quickly understood by users and enables the generation of expressive explanations consisting of not only causal relationships but also temporal and hierarchical relationships among agent concepts. This additional information can be useful during development, maintenance, and presentation of the agent system.

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. Observations can hold an additional set of attributes (e.g., a stack

trace and a timestamp of the realtime) that is not agent-relevant but is helpful in debugging the source code. Details about the process of building background knowledge K with the aid of the Tracer Tool are described in Section 4.2 for Research Question 1.

Chapter 4

Research Question 1: Building Background Knowledge

An explanation is a statement which points to causes, context and consequences of some object (or process, state of affairs etc.), together with rules or laws which link these to the object. Some of these elements of the explanation may be implicit.
Explanations can only be given by those with understanding of the object which is explained.

[2005 Wikipedia]

This chapter focuses on the contributions and products resulting from answering Research Question 1 (RQ1). Section 4.1 describes the approach taken to answer RQ1. Section 4.2 describes the interpretation process as implemented in the Tracer Tool. Experiments with the Tracer Tool in Section 4.4 demonstrates the utility of the approach. Finally, 4.5 summarizes the experimental results and contributions within the scope of RQ1.

4.1 Approach to Research Question 1

To generate accurate explanations, the background knowledge should be representative of what is being explained (i.e., agent behavior in the implementation). This implies that the background knowledge (representing expected agent

behavior) must be complete and consistent with the implementation's behavior (see Equation 3.5). Research Question 1 (RQ1) asks this question – *How can background knowledge be created such that it accurately represents the implementation's behavior?* By explicitly modeling the user's comprehension as background knowledge K, the accuracy of K can be verified against the actual implementation during the interpretation process, which has been automated by the Tracer Tool.

The following describes this solution in more detail and answers RQ1 by focusing on Equations 3.1, 3.3, and 3.4 presented in the previous chapter.

- Equation 3.1 shows how actual behavior is captured from the implemented system;
- Equation 3.3 looks at determining the representative accuracy of K; and
- Equation 3.4 examines how K can be built up and updated by analyzing the observations.

Equation 3.1: $O_s = observe(execute(I, s))$:

Since K and N_s are modeled at the agent concept abstraction level, only agent concepts are extracted from the implementation – more detailed concepts (e.g., data structures and method calls) are not needed. To acquire only the agent concepts from the implementation, the approach is to instrument the source code (i.e., add extra code to log data). The extra logging code is inserted at locations where agent concepts occur or change. When the implementation is executed in a scenario s, only agent-relevant data is logged as observations O_s, which are collected via the Tracer Tool. This reverse engineering approach requires only a high-level structural understanding of the implementation and encompasses the entire

agent system implementation rather than just portions of the code. The scalability of this approach is better than reverse engineering because only relevant data about the system is analyzed, not every method call or data structure change. This approach translates run-time data and occurrences from the implementation execution into observations of agent concepts. Since the observations are coming from numerous agents and may be out of order, the Tracer Tool sorts and organizes the observations (during run-time) for the next step, which is creating the interpretations.

Equation 3.3: $\boldsymbol{N_s = interpret(K, O_s)}$:

To produce interpretations N_s from the implementation, observations O_s of the implementation execution and background knowledge K are used as shown in Figure 3.3. Traditionally, observing and relating observations together is performed manually by the user. However, by leveraging K, the Tracer Tool can automatically interpret the observations for the user by linking appropriate observations together, building interpretations (i.e., behavioral models) of the observations based on order, origin, and other run-time attributes.

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 [Barber and Lam, 2003a].

In the case of this research, the interpretations are semantic graphs with observations as nodes. 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. As shown in Figure 3.1, background knowledge about the expected behavior of the implemented system is required to relate the otherwise unconnected observations together. For example, in Figure 4.1, the background knowledge for an agent's behavior denotes an $intention$ that is influenced by two different beliefs (*belief* and *belief-2*) and a *goal*. The *intention* causes an *action* to occur, which results in an *event-2*, which in turn affects *belief-2*. Note that K represents a behavioral pattern and, thus, can have cycles in the graph. However, the interpretation, which consists of actual observations and their relationships, does not have cycles.

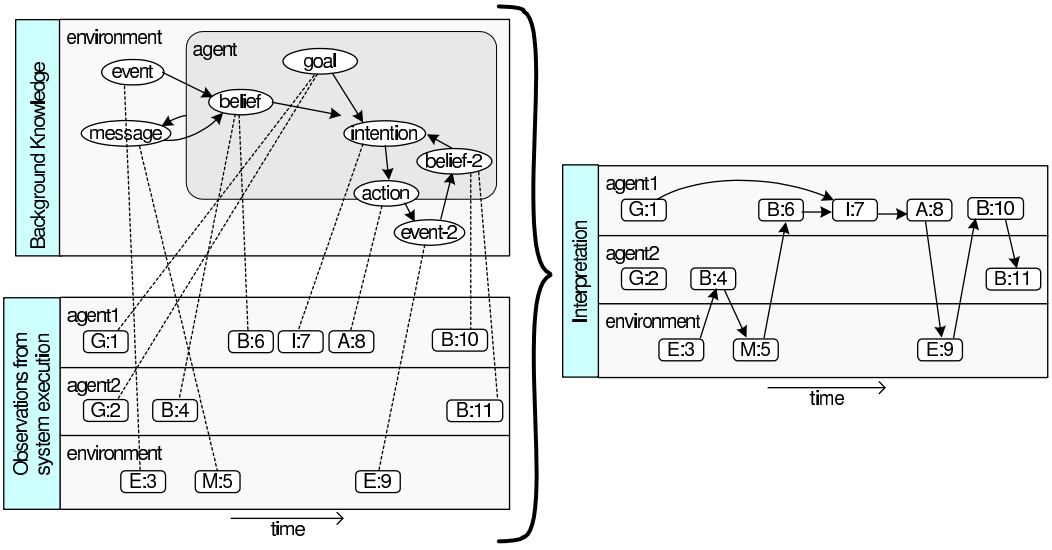


Figure 4.1: An interpretation created from the background knowledge K and observations O_s

To build an interpretation, observations are mapped to agent concepts in K and are linked together using relations defined in K. In Figure 4.1 for example, ob-

servations *B:4* and *B:6* are mapped to agent concept *belief* because the observations are beliefs about a target's state; *B:10* and *B:11* are mapped to *belief-2* because the observations are beliefs about the target's location; the initial goals *G:1* and *G:2* are mapped to *goal*; *I:7* is mapped to *intention*; etc. Since *intention* is causally influenced by *goal* and *belief*, directed edges are added between the appropriate nodes (e.g., from *G:1* and *B:6* to *I:7*) to relate the observations together. In other words, since the user expects the *goal* and beliefs about a target's state *belief* to influence the agent's *intention*, the user will create an interpretation where the corresponding observations for that agent are causally linked.

This is repeated for each observation, and the resulting interpretation is visualized as a semantic graph 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. Suggestions can help the user modify K appropriately.

Equation 3.4: $\boldsymbol{k = suggest(N_s)}$:

Since the interpretation process performs the mapping between the observations O_s and agent concepts in K, K is verified against the implementation I. If there exists an observation $o \in O_s$ that cannot be related to some other observation based on defined relations in the current K, then a suggestion is offered by the Tracer Tool to update K so that o is a consequence of some other observation. This occurs when K has no incoming relation for the agent concept corresponding to observation o. Starting from o, the relation-suggesting algorithm searches backwards (temporally) through the observation list, using heuristics to determine

if a previous observation is related in some way to o. The heuristics leverage the typical relationships among agent concepts. For example, if o is an action, then the algorithm searches for the last observed intention p that has some similar attribute as those of the action o. If such an intention is found, a relation from intention p to action o is suggested.

Experiment 1 tests the capability of the relation-suggesting algorithm. If the suggestion list is empty (i.e., $k = \emptyset$), then K is complete – all actual behaviors are modeled by the expected behavior representation of the background knowledge. Experiment 2 investigates other potential representation accuracy problems and their causes.

4.2 Tracer's Interpreter and Suggester

To address Research Question 1 (*How can background knowledge be created such that it accurately represents the implementation's behavior?*), the Tracing Method was developed to help check the consistency between the background knowledge and the observations made from executing the implementation. Once the background knowledge is checked, the automated reasoners can employ the background knowledge to facilitate software comprehension, i.e., explain agent behavior. Another use of the Tracing Method is to gather the scenario observations O_s used by the explainer.

The approach to RQ1 is analogous to a human's learning process. 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 to collect the observations (i.e., executing the implementation in several scenarios), and (3) interpreting the observations (i.e., verifying K against the actual imple-

mentation) [Lam and Barber, 2005*b*]. The Tracing Method imitates this cycle and the Tracer Tool automates some of the traditionally manual tasks involved in each of these steps, such as analyzing run-time data and relating it to models of agent behavior in the background knowledge.

The Tracing Method can be used repeatedly throughout the software life-cycle from the first skeleton code to the final system. The Tracing Method and Tracer Tool has been developed for agent systems that are implemented in procedural programming languages (e.g., Java, C, and C++), but they can also be used in declarative agent-oriented programming languages (e.g., AF-APL [Ross et al., 2004] and Suna et al.'s mobile agent language [Suna and Fallah-Seghrouchni, 2004]) to visualize and clarify agent behavior in the system. Currently, the Tracer Tool includes Tracing clients that allow Java and Lisp implementations to send logs (i.e., observations) to the Tracing server.

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 [Jennings, 1999]. 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 behavior, and to help explain these behaviors. As shown in Figure 4.2, the Tracing Method involves logging agent behavior during execution, interpreting the log entries and run-time data into agent concepts (i.e., creating interpretations), and comparing those interpretations with the background knowledge. The Tracing Method produces interpretations (that represent the actual agent behavior in terms of agent concepts) from 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 inheri-

tance models or state-chart and process-flow diagrams [Barber and Lam, 2003*b*]), or the interpretations can be relational graphs linking observations together (as is the case in this dissertation).

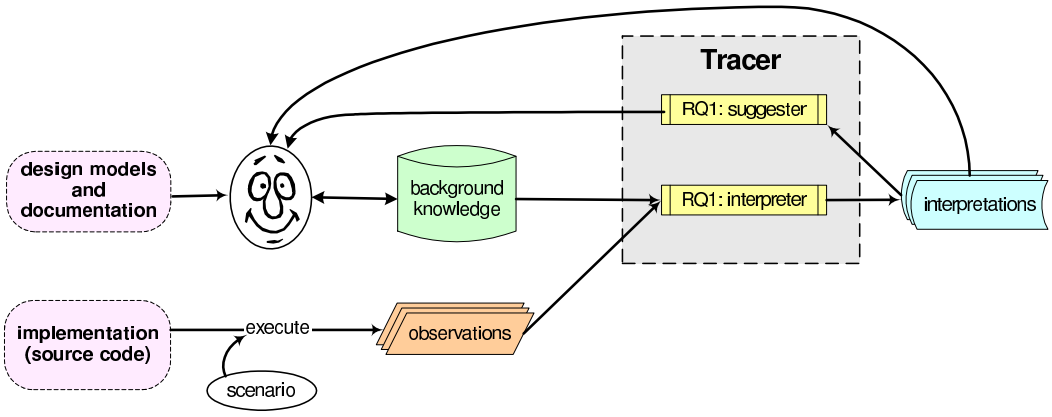


Figure 4.2: Approach for Research Question 1

This section describes and demonstrates the *interpreter* and *suggester* of the Tracer Tool (shown in Figure 3.3). The Tracer Tool addresses the comprehension issues (described in Section 2.3) in the following ways:

low abstraction level : Background knowledge K is represented as a collection of high-level agent concepts familiar to designers, developers, and end-users.

language-dependent : The Tracing Tool records observations logged from the implementation's execution, rather than analyzing language-dependent stack traces and process threads.

large amount of data : The Tracer Tool automates the task of collecting, organizing, and interpreting the observations and can present the interpretation to the user as a relational graph that can be quickly digested.

human user must digest data : Given interpretations and K, automated reasoning can highlight new concepts and relations that the user has not yet modeled in K and ignore already modeled relations.

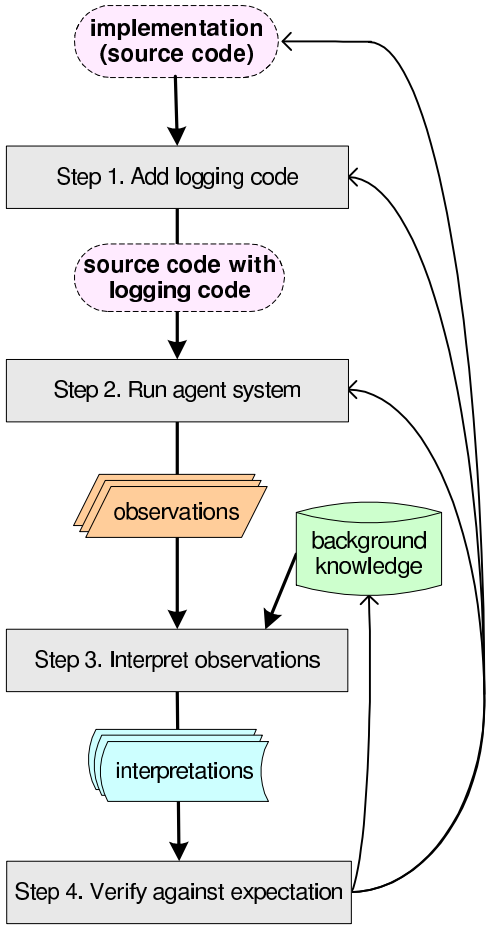


Figure 4.3: Process diagram of the Tracing Method

Each step of the Tracing Method shown in Figure 4.3 is described in the following subsections, accompanied by screenshots from the Tracer Tool. 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). The Tracing Method involves (Step 1) instrumenting the source code so that agent behavior is logged during execution, (Step 2) compiling and executing the agent system to capture run-time data, (Step 3) transforming those observations into a more unified interpretation(s), and (Step 4) comparing the interpretations with models of expected agent behavior in the background knowledge.

To clarify which tasks in the Tracing Method have been automated and where additional interpreters and analyzers can be integrated into the Tracer Tool, Figure 4.4 illustrates each component of the Tracer Tool. In Step 1, Tracer has a logging code generator that produces lines of code to be inserted into the implementation's source code, along with a `LoggingHelper` class which acts as a mediator between the agent system and the Tracer Tool. In Step 2, when the system is executed, the `LoggingHelper` employs the `TracingClient` to send logs (possibly over the network) to the `TracingServer`, which may be on another computer. In Step 3, the `Interpreter` and possibly other interpreters create different interpretations or views of the system execution based on observations gathered by the `TracingServer` and the background knowledge. Finally, in Step 4, the `Explainer` and possibly other analyzers can help the user compare actual and expected behaviors.

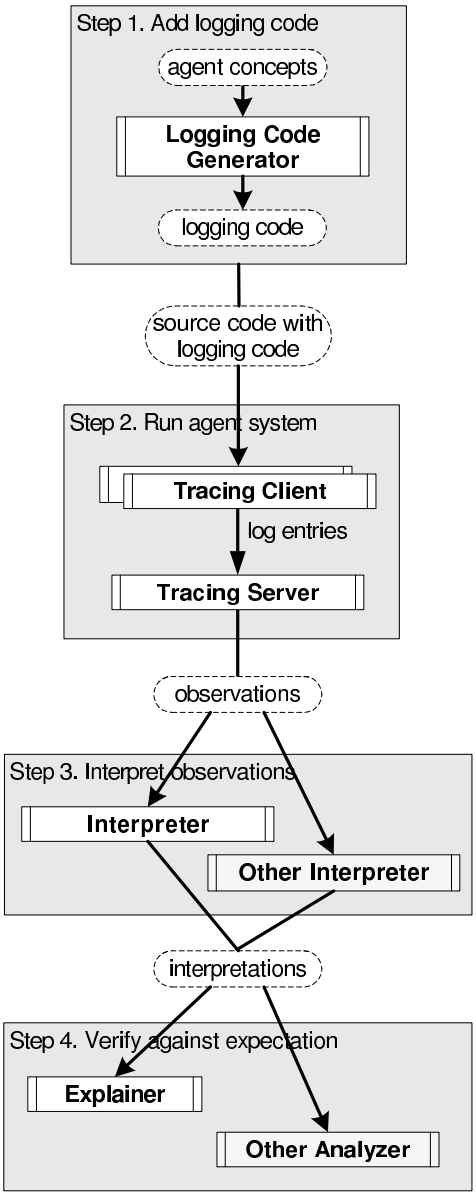


Figure 4.4: Components of the Tracer Tool

4.2.1 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 mentioned in Chapter 2 (e.g., Gen++ [Devanbu, 1992] and DESIRE [Biggerstaff et al., 1994]), the Tracing Method does not analyze code in detail, thus, it is not dependent on any specific language. Instead, run-time data about agent concepts is acquired by explicitly logging the data. This involves instrumenting the implementation by adding *logging code* to specific locations in the source code. 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.

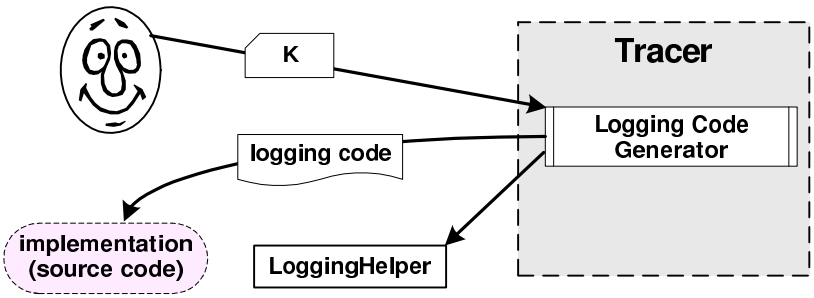


Figure 4.5: Step 1: User assistance from the Tracer Tool

Figure 4.5 shows the interaction between the user and the Tracer Tool. The Tracer Tool aids the user by generating the appropriate logging code to be inserted into the implementation, given some background knowledge specified by

the user. For example, Listing 4.1 shows the background knowledge for a basic agent system. For each agent concept specified in the background knowledge, the Tracer Tool generates a corresponding one-line logging code in Java (see Listing 4.2) to be inserted into the appropriate location in the implementation. The logging code calls the appropriate method in the `LoggingHelper` class (code snippets of which are in Listing 4.3), which is also generated by the Tracer Tool. The `LoggingHelper` class then uses the `TracingClient` class to send the logs to the `TracingServer`, as shown in Figure 4.4.

In the code listings, the variable `me` is some identifier for the agent that "owns" the agent concept. In the case of an action or event, `me` refers to the performer of the action or source of the event; and in the case of a message, `me` is the sender or receiver of the message. For further reference, see Figure 3.4 for other agent concept attributes. When executed, each piece of logging code will fill in all attributes specified in the background knowledge specification.

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 Figure 4.6, 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, [Barber and Lam, 2003*a*] describes how such contextual logging code can result in state-chart and process-flow diagrams (other representations of software behavior), which are then verified.

Listing 4.1: Background knowledge specification used to generate logging code

```
loggingClass="tracer.TracingClient";

// AGENT CONCEPTS

// belief syntax
// id: B(<name>, {<key-value pairs>});
1:  B("currentLocation", {"location", point});
2:  B("visitedLocations", {"locations", points});
3:  B("visitedLocationByOthers", {"location", visitedLocation});
4:  B("acceptedLocation", {"location", point});
5:  B("foundObject", {"location", point});

// event syntax
// id: E(<name>, {<preconditions>}, {<postconditions>});
// preconditions and postconditions are a list of key-value pairs
11: E("moveTo", null, {"location", point});
12: E("foundObject", null, {
      "object", envObject,
      "location", point
    });

// message syntax
// id: M(<name>, <from>, <to>, {<key-value pairs>});
// messages to me
21: M("propose", agent2, me, {"location", point});
22: M("reject", agent2, me, {"location", point});
23: M("accept", agent2, me, {"location", point});

// messages from me
24: M("propose", me, agent2, {"location", point});
25: M("reject", me, agent2, {"location", point});
26: M("accept", me, agent2, {"location", point});

// intention syntax
// id: I(<name>, {<key-value pairs>});
31: I("nextLocation", {"location", point});

// action syntax
// id: A(<name>, {<preconditions>}, {<postconditions>});
41: A("moveTo", null, {"location", point});
```

Listing 4.2: Generated logging code to be inserted into implementation

```
/* loggingCodeID=1 */
LoggingHelper.logBelief_currentLocation(me, "comment", point);

/* loggingCodeID=2 */
LoggingHelper.logBelief_visitedLocations(me, "comment", points);

/* loggingCodeID=3 */
LoggingHelper.logBelief_visitedLocationByOthers(me, "comment",
  visitedLocation);

/* loggingCodeID=4 */
LoggingHelper.logBelief_acceptedLocation(me, "comment", point);

/* loggingCodeID=5 */
LoggingHelper.logBelief_foundObject(me, "comment", point);

/* loggingCodeID=11 */
LoggingHelper.logEvent_moveTo(me, "comment", point);

/* loggingCodeID=12 */
LoggingHelper.logEvent_foundObject(me, "comment",
  envObject, point);

/* loggingCodeID=21 */
LoggingHelper.logMessageReceive_propose(me, "comment",
  agent2, me, point);

/* loggingCodeID=22 */
LoggingHelper.logMessageReceive_reject(me, "comment",
  agent2, me, point);

/* loggingCodeID=23 */
LoggingHelper.logMessageReceive_accept(me, "comment",
  agent2, me, point);

/* loggingCodeID=24 */
LoggingHelper.logMessageSend_propose(me, "comment",
  me, agent2, point);

/* loggingCodeID=25 */
LoggingHelper.logMessageSend_reject(me, "comment",
  me, agent2, point);

/* loggingCodeID=26 */
LoggingHelper.logMessageSend_accept(me, "comment",
  me, agent2, aPoint);

/* loggingCodeID=31 */
LoggingHelper.logIntention_nextLocation(me, "comment", point);

/* loggingCodeID=41 */
LoggingHelper.logAction_moveTo(me, "comment", point);
```

Listing 4.3: Code snippets from LoggingHelper.java

```
package tracer.generated;
import edu.utexas.lips.tracer.TracerLogRecord_Interface;

/** Implementation of methods called by the logging code **/
public class LoggingHelper {
  public static void logBelief_currentLocation(String ownerVar,
    String comment, Object point) {
    tracer.TracingClient.log(ownerVar, "currentLocation",
      TracerLogRecord_Interface.BELIEF, comment,
      new Object[]{"location", point});
  }
// ...
  public static void logEvent_foundObject(String ownerVar, String
    comment, Object envObject, Object point) {
    tracer.TracingClient.log(ownerVar, "foundObject",
      TracerLogRecord_Interface.EVENT, comment,
      new Object[][]{null, {"object", envObject, "location", point
        }});
  }
  public static void logMessageReceive_propose(String ownerVar,
    String comment, String agent2, String me, Object point) {
    tracer.TracingClient.log(ownerVar, "propose",
      TracerLogRecord_Interface.MESSAGE_RECEIVE, comment,
      new Object[]{agent2, me, new Object[]{"location", point}});
  }
// ...
  public static void logIntention_nextLocation(String ownerVar,
    String comment, Object point) {
    tracer.TracingClient.log(ownerVar, "nextLocation",
      TracerLogRecord_Interface.INTENTION, comment,
      new Object[]{"location", point});
  }
  public static void logAction_moveTo(String ownerVar, String
    comment, Object point) {
    tracer.TracingClient.log(ownerVar, "moveTo",
      TracerLogRecord_Interface.ACTION, comment,
      new Object[][]{null, {"location", point}});
  }
}
```

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

Figure 4.6: Log entries for agent `Bot15` in the UAV domain

Since only agent concepts are logged, this logging approach requires only high-level functional and structural knowledge of the implementation (e.g., entity relationship diagrams, data flow diagrams, and process diagrams). If there is insufficient or erroneously-placed logging code, these errors will manifest themselves as specific representative errors (e.g., inconsistencies) between the interpretations and the background knowledge. The developer can use the identified inconsistency to determine if logging code should be added or modified. Experiment 2 (Section 4.4.2) exemplifies the types of representative errors and how they can be remedied.

This first step begins to address the *translation gap* problem by identifying agent concepts in the implementation and mapping those to agent concepts in the design. In software engineering, a translation gap occurs whenever a more abstract representation is translated into a more detailed representation, for example from design to architecture and from architecture to implementation. Automated tools help alleviate the translation gap by providing traceability between abstract concepts and detailed concepts such that, for example, every design concept (that has been implemented) can be traced to some piece of the source code. Model-checking has the translation gap problem because concepts in the model being checked is not explicitly mapped to concepts in the implemented system. In other words, the modeled concepts are not grounded in some real system. Using the Tracing Method, every modeled concept in the background knowledge that occurs in the implementation is logged as an observation, assuming the user has inserted the logging code in the appropriate location in the source code. Each observation holds information, e.g., the stack trace and class method in which the observation occurred, that helps the user locate the corresponding agent concept in the source code.

4.2.2 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. A logging mechanism based on the Java Logging Framework [*Java Logging APIs*, 2002] 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 (via CORBA^{TM}[CORBA, 1999] or DARPA's CoABS Grid [CoABS, 2003]).

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, Figure 4.6 displays all types of log entries on a single display. Figure 4.6 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) is acquired at run-time and appended to the log entry. The idea is to capture dynamic information that makes agent systems unpredictable (see Chapter 2) and to incorporate that information in the explanation.

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 Figure 4.6, the log entry with ID 94 was logged by thread 12, while most of the other entries were logged by thread 13.

4.2.3 Step 3: Interpret logging results

The third step is to interpret the run-time data into concise models (called interpretations). There are several ways to automatically interpret the observations listed in Figure 4.6, 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 [Barber and Lam, 2003*a*] 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. Based on 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. These interpretations can be used to help the developer identify critical points in system (e.g., decision-making and coordination events and data and computational bottlenecks) and locate bugs in the system.

To generate semantic graphs, rules that relate agent concepts to each other are applied to the observations. The rules are a mechanism to form relations as defined in the background knowledge. The Tracer Tool generates rules based on the relations defined in the background knowledge. The rules compare the constituent attributes of agent concepts (e.g., name, subject, preconditions, and/or postconditions) in order to associate one observation with another. For example, a rule may state 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 may state that if belief b was observed before intention p was created and after the intention prior

Listing 4.4: Interpreting algorithm

```
interpret(OBSERVATION o, GRAPH N_s) {
  for each RULE r {
    int numOfRelationsFormed=0
    if ( r.appliesTo(o) ){
      for each past_OBSERVATION p within r.
        searchHorizon {
        if ( r.passesTests(o,p) ) {
          N_s.createRelation(p,o);
          numOfRelationsFormed++;
        }
        if (numOfRelationsFormed == r.
          numOfRelationsToFind){
          break; // go back to the outer-most for-loop
        }
      }
    }
  }
}
```

to p, then belief b affects intention 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. Essentially, the background knowledge is being used as a template to relate otherwise unconnected observations.

Listing 4.4 describes in pseudocode how the Tracer Tool's interpreter processes each observation to find related observations. For each rule derived from the background knowledge, check if the rule is applicable to the observation o. If it is, all past observations within the rule's search horizon are examined (in reverse chronological order) with respect to o. The search horizon can be none (where all past observation are examined) or an observation matching certain attributes. For example, if the search horizon matches an intention observation, then all observations between the last intention and the current observation are examined. If a past

observation p passes all tests for the rule, a relation from p to o is created. If a sufficient number of relations have been formed as declared by the rule, then applying the rule is complete and the next rule is checked for applicability on the observation o. The computational complexity of the interpreting algorithm is O(n^2), where n is the number of total observations or the number of nodes in the graph.

The idea behind the interpreter is to automatically generate informative representations of agent behaviors from unconnected observations using (potentially reusable) rules. To help build up the background knowledge, from which the rules are generated, rules for unconnected observations can be automatically suggested using heuristics about the general relationships among agent concepts. The relation-suggesting algorithm is described in the next step.

4.2.4 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, interpretations in the form of 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. Traditionally, due to the lack of a formal specification of agent behavior, verifying the interpretations is a manual step – a human must compare the interpretations against formal and informal expected behavior models.

To address this problem, Tracer facilitates the identification of anomalous or incorrect behavior. For example, given the relational graph interpretation of the observations, nodes that are unconnected to the graph indicate anomalous behavior. Incorrect behavior can also be automatically detected by specifying relations that should not occur.

To allow for other analyses of different interpretation types, additional analysis tools can be plugged into the Tracer Tool (see Figure 4.4). 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 background knowledge, the cause of the errors may be one or more of the following:

1. logging code may need to be moved 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.

These different types of errors are the reason for arrows pointing from Step 4 to previous steps or objects in Figure 4.3. Verification error 1 is due to human error (it is assumed that the user understands the source code well enough to insert the code in the correct places). Verification error 2 is beyond the scope of this dissertation (it is assumed that the set of execution scenarios sufficiently covers the range of possible code executions). Correcting verification error 3 is also beyond this dissertation scope. However, the interpretation and recorded observations provide information that can be useful in correcting these errors. 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.

To further aid in correcting verification errors, 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 discover specific factors influencing an agents decisions and to ensure that an agent is performing the action for the right reasons. Section 5.2 demonstrates how an explanation is created.

To aid in correcting the cause for verification error 4, suggestions for updating the background knowledge can be generated by the Tracer Tool. The relation-suggesting algorithm (shown in Listing 4.5) is initiated for an observation o when the Tracer Tool cannot create an incoming relation for observation o based on the

Listing 4.5: Relation-suggesting algorithm

```
suggestRelationFor(OBSERVATION o) {
  for each HEURISTIC h {
    for each past_OBSERVATION p within h.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("valueOfAttrib(a)==valueOfAttrib(b)")
           to s;
      }
      // This can get complicated: XX ...
      if ( isTypeOfCollection(valueOfAttrib(a))
           && valueOfAttrib(a).contains(valueOfAttrib(
             b)) ){
        add TEST("valueOfAttrib(a).contains(
          valueOfAttrib(b)") to s;
      }
      if ( isTypeOfCollection(valueOfAttrib(b))
           && valueOfAttrib(b).contains(valueOfAttrib(
             a)) ){
        add TEST("valueOfAttrib(b).contains(
          valueOfAttrib(a)") to s;
      }
    }
  }
  return s;
}
```

Listing 4.6: Example heuristic testing if an action is related to an event

```
heuristic_1{
  appliesTo(p,o) {
    return IS_ACTION(p) and IS_EVENT(o) and
      !empty(commonAttributes(p,o));
  }
}
```

current K. This occurs when K has no incoming relation for the agent concept corresponding to observation o that originates from some other observation. In other words, the relational graph does not show a cause for observation o. Starting from o, the algorithm searches backwards (temporally) through the observation list, using heuristics to determine if a previous observation (this is within the heuristic's search horizon) 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 search horizon of the heuristic specifies how far back to search for a relating observation. It can be none (where the search does not stop until a relation is suggested or the first observation is reached), or it can be a predicate, such as `isIntention` (where the search stops before the first intention is found). An example heuristic that searches for the relation connecting an action to an associated event is shown in Listing 4.6. A list of all heuristics is provided in Section 4.4.1.

The algorithm always terminates because it only looks at preceding observations and there is a finite number of past observations. The computational com-

plexity of the suggesting algorithm for each observation is O(n), where n is the number of past observations.

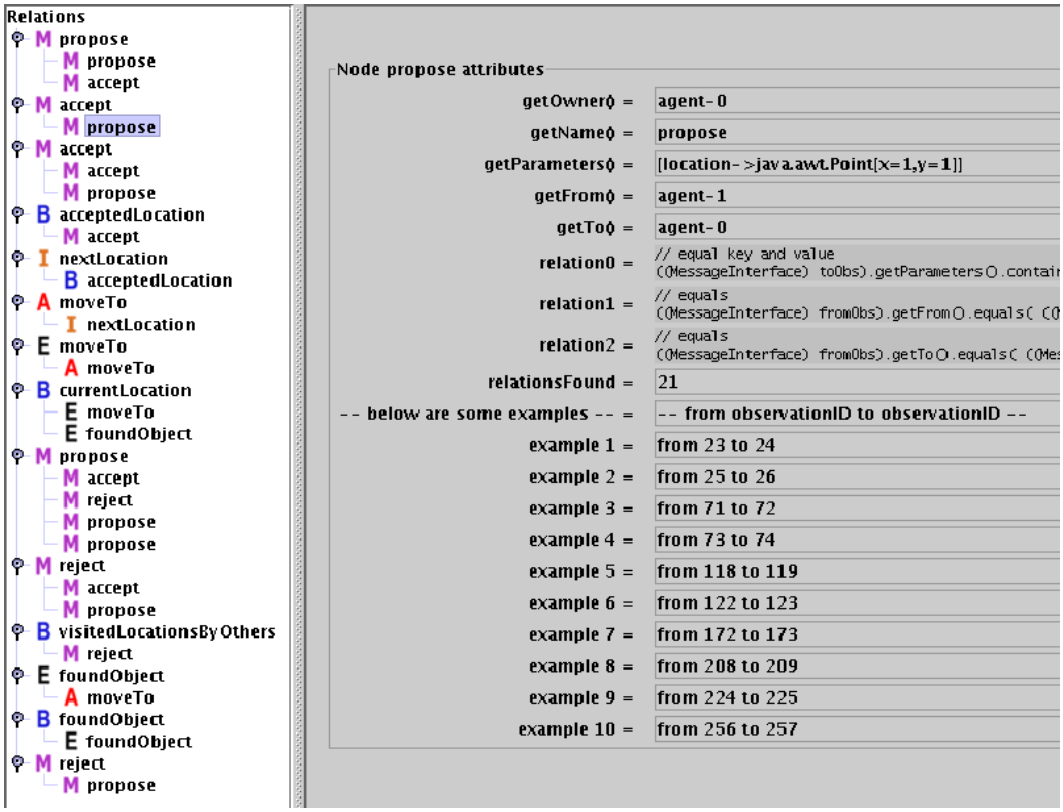


Figure 4.7: Suggested relations from the Tracer Tool for the UAV application

For each observation without an incoming relation, the algorithm suggests a relation and adds it to the list (if it has not already been added) or appends it to an existing relation, as shown in Figure 4.7. 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 4.7). The examples are provided so that the user can investigate the example observations to determine if the suggested relation is semantically correct.

♦◊♦

The result of the Tracing Method is verified background knowledge and interpretations of the implemented agents' behaviors in terms of agent concepts. Thus, the user's comprehension of how the agents behave accurately reflects the actual implemented agent system. As a side effect of the Tracing Method, the source code is documented (though sparsely) with logging code that identifies important points in the code for understanding the implemented agents' behaviors. Chapter 5 describes how the verified background knowledge can be used to further automate comprehension tasks. Before delving into the experiments related to RQ1, a case study in the Unmanned Aerial Vehicle domain is described in the next section to set up the context for the experiments and to provide an example of the Tracing Method and the Tracer Tool.

4.3 Case Study: Tracing the UAV Domain

To demonstrate the Tracing Method, the Tracer Tool will be used to trace agents in an implemented agent system for the Unmanned Aerial Vehicle (UAV) target-monitoring domain. Each agent receives input from and controls a single UAV. 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) while minimizing the 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, verifying agent behavior is facilitated by using the Tracing Method and Tracer Tool.

Figure 4.8 shows a screen capture from the UAV Simulation software. There are three UAVs shown as solid black circles labeled 15, 16, and 17; their radar or scanning range is indicated by the larger circle around each UAV. Each UAV's projected path is shown as a dotted line to the 'x' destination or way-point. The small gray circles labeled 1 through 14 are the targets to be monitored. Each target has concentric rings around it that indicate relatively how long it has been since that target was last scanned. For example, since target 5 was just scanned by UAV 15, it has a small ring around it. On the other hand, target 10 (UAV 15's next destination) has several large rings around it, indicating that it has not been scanned recently.

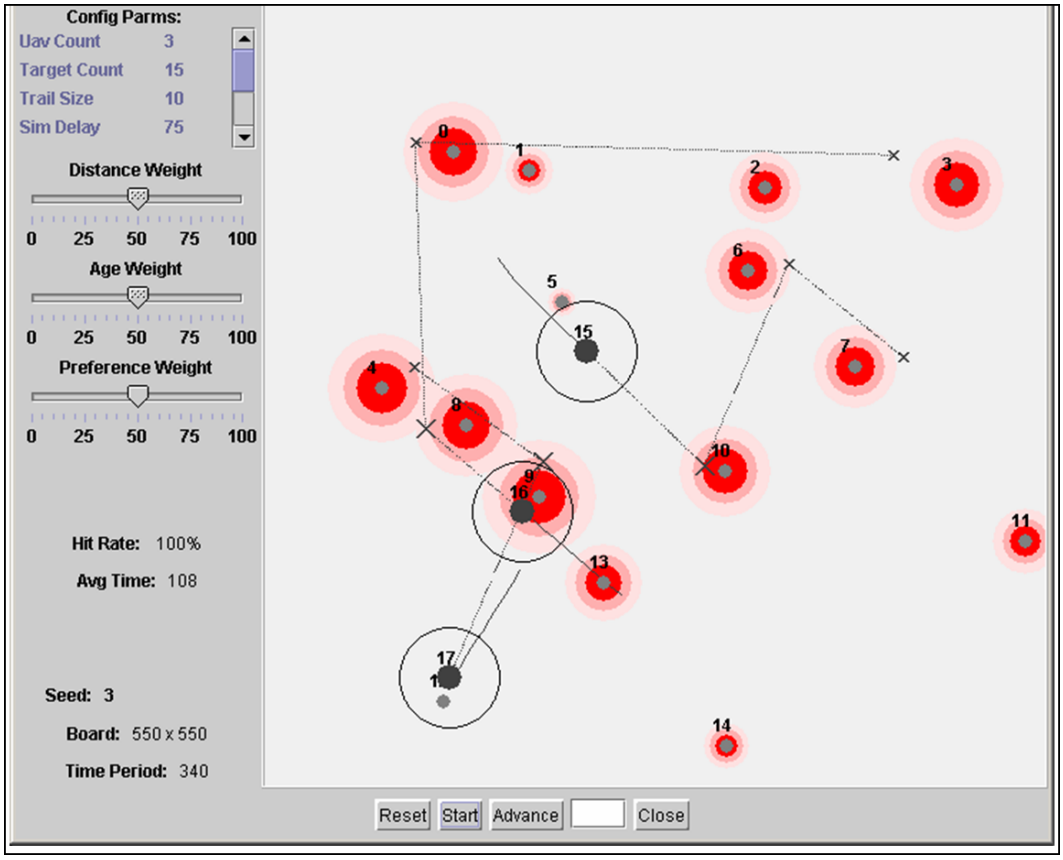


Figure 4.8: UAV Simulation Viewer

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 (1) the simulation was coded by Metron, Inc., as a contribution to the DARPA TASK project [TASK, 2001], (2) the agents were programmed by a developer in our lab, and (3) the logging code was added to the agents and simulation by the author

of this dissertation. The following lists specific agent concepts that are logged as observations 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;

goal : (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 Figure 4.6) to the Tracing Server, which translates the log entries into observations. Table 4.1 lists some of 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 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

Table 4.1: 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)
...	...	...	...

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 4.1, 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. Initially, the background knowledge K holds the set of agent concepts that were logged – no relations between the concepts were defined. Thus, the observations are not yet connected or related to each other. For each unconnected observation, the Tracer Tool will automatically suggest a relation to be added to K. Experiment 1 (Section 4.4.1) discusses the details of the relation-suggestion algorithm and the construction of K further.

Provided with relations that connect agent concepts together in the background knowledge, the relations are converted into rules. In future executions of the system, the rules are applied to the observations in order to create the graphs.

The interpreting algorithm applies the appropriate rule(s) to each new observation that appears and searches backward (temporally) to find the latest previous observation that satisfies the rule antecedent. Following is an example of a rule used in the UAV domain that represents the relation between a message and a belief with the same name and value attributes.

- If belief b occurs after message m and their 'name' attributes are equal and their 'value' attributes are equal, then m affects (or caused) b.

More complicated application-specific rules can be created to relate more than two observations together. In a rule, there are three major aspects used in the interpreting algorithm shown in Listing 4.4 – the search horizon, the matching criteria, and the maximum number of matching observations to find. The search horizon specifies how far back in the observation list, e.g., the previous intention observed. Much of the effort goes into the matching criteria which determines if two observations are related based on the observations' type and attribute values. The relation-suggesting algorithm is meant to reduce the amount of effort the user must invest in building the background knowledge K.

Figure 4.9 illustrates the relational graph generated from the interpretation of the observations in Table 4.1. 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 4.1. 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.

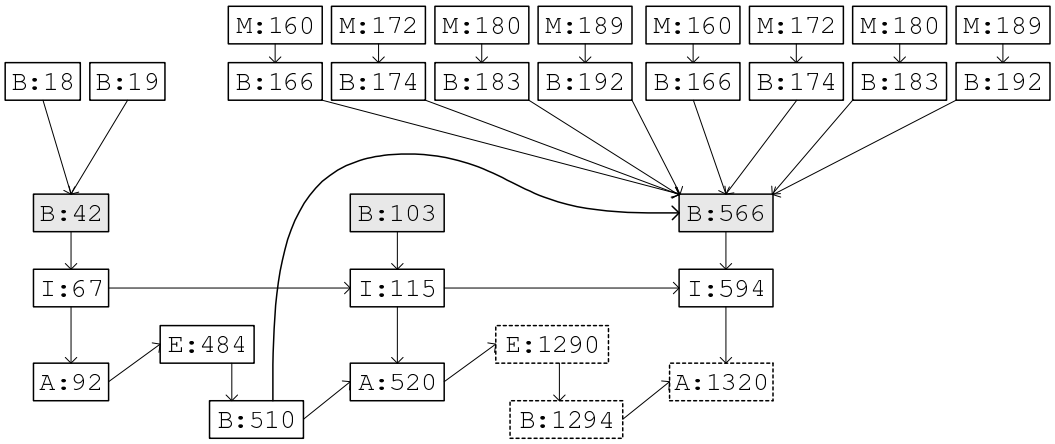


Figure 4.9: Relational graph for agent `Bot15`

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 Figure 4.9 and detailed in Table 4.1.

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. 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` flies to `target 1`'s believed location). This series of observations can be easily followed in Table 4.1. 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 4.1) 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.

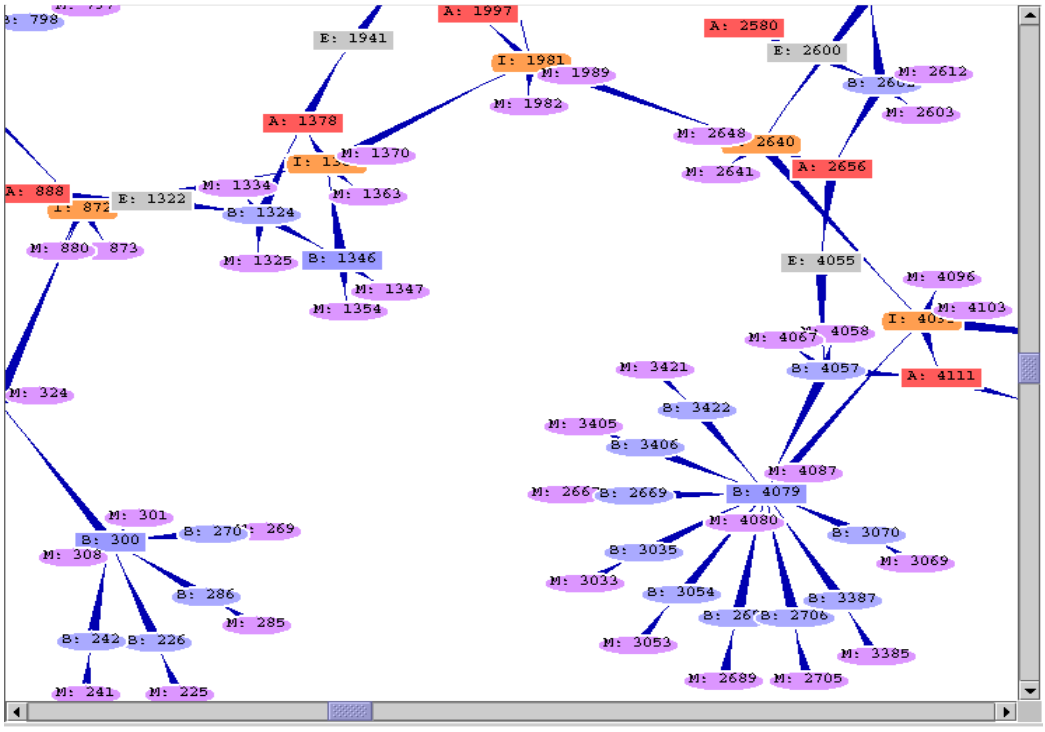


Figure 4.10: Relational graph from the Tracer Tool

A relational graph (such as one produced by the Tracer Tool in Figure 4.10) can present important information that may not be as obvious when presented as a

list or table. As demonstrated, the graph provides a quick way to understand (and ask questions about) the operations of the agent without having to understand the source code 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. Given this causal graph, automated explanation generation (discussed in Chapter 5) is simplified. Additionally, some patterns of behavior are more easily discovered by the human user in graph form. For example, in Figure 4.9, 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 Figure 4.11, 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, Figure 4.12 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.

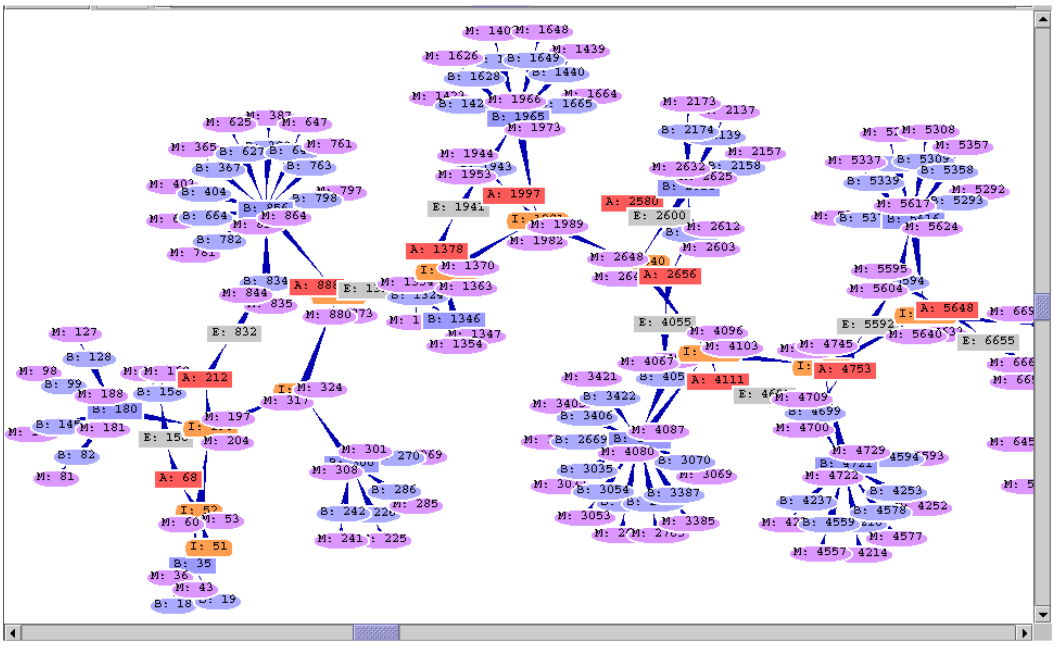


Figure 4.11: Patterns in relational graph

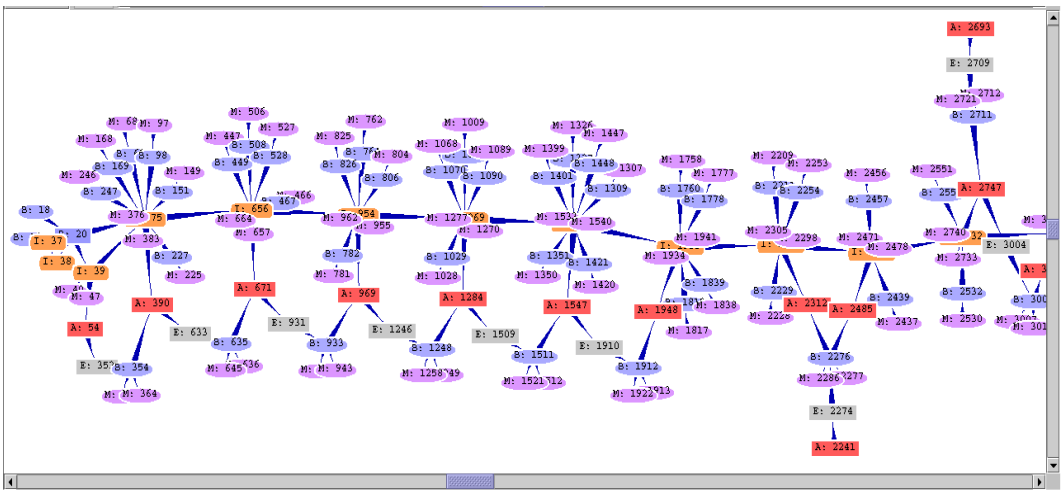


Figure 4.12: Anomalies in the relational graph (on right-side) for a different agent

4.4 RQ1 Experiments

This section discusses two experiments for Research Question 1, which asks *How can background knowledge be created such that it accurately represents the implementation's behavior?* The following experiments test Tracer's ability to help build background knowledge K and to detect (and help resolve) accuracy errors in K with respect to the implemented system.

Experiment 1 (EX1) : tests the correctness and completeness of the relation-suggesting algorithm in Tracer;

Experiment 2 (EX2) : tests what types of accuracy errors can be detected in the interpretation and how they can be resolved;

4.4.1 EX1: Relation-suggesting algorithm performance

The relation-suggesting algorithm (shown in Listing 4.5) can be used to help the user build up the background knowledge K. The purpose of this experiment is to gauge how well the algorithm (or more specifically, the heuristics used by the algorithm) performs in automatically building an accurate background knowledge K. The results of this experiment implies the capability of the Tracer Tool to automate comprehension tasks – the more tasks that are automated, the less effort the user has to invest. The following two metrics will be used to evaluate the algorithm:

correctness : the percentage of suggested relations that are correct or partially correct

completeness : the percentage of relations in K that were correctly suggested (as opposed to manually-created or modified)

The experimental setup involved applying the Tracer Tool on two multi-agent systems: "Simple" (approximately 500 lines of Java code) and "UAV" (ap-

proximately 20,000 lines of Java code). For each domain application, logging code for each agent concept was inserted into the source code. For the Simple domain, 17 agent concepts were logged; for the UAV domain, 22 agent concepts were logged. Each system was then executed with an initially empty background knowledge of the system, analogous to understanding nothing about the system. Initially, observations were recorded but no cohesive interpretation can be created due to the empty background knowledge K. Since none of the observations are related to any other observation, the relation-suggesting algorithm is enabled and performed on each observation. The following heuristics are used by the algorithm to suggest possible relations among agent concepts.

H.1. If the observation is an intention, suggest relations from all previous beliefs (that have common attributes) that occurred after the previous intention and doesn't already have an outgoing relation. If no belief is found, link to a previous goal with common attributes.

H.2. If the observation is a belief, suggest a relation from the latest message or event with common attributes that doesn't already have an outgoing relation.

H.3. If the observation is an event, suggest a relation from the latest action with common attributes that doesn't already have an outgoing relation.

H.4. If the observation is an action, suggest a relation from the latest action or intention with common attributes that occurred after the previous intention and doesn't already have an outgoing relation.

H.5. If the observation is a received message, suggest a relation from the latest message (that has common attributes) that was sent to the sender of the received message and doesn't already have an outgoing relation.

H.6. If the observation is a sent message, suggest a relation from the latest observation with common attributes and that doesn't already have an outgoing

relation. When searching, skip all received messages whose sender is not the receiver of the sent message.

H.7. If the observation is an intention, suggest relations from all previous beliefs (*regardless of the lack of common attributes*) that occurred after the previous intention and doesn't already have an outgoing relation. If no belief is found, link to a previous intention or goal (*regardless of the lack of common attributes*).

The following lists all suggested relations resulting from using the above heuristics in the relation-suggesting algorithm for the UAV domain. The suggestions are shown to the user as a tree representation where child nodes are the cause of parent nodes as seen in Figure 4.13. Note that Suggestion S.1 requires modification and that Suggestion S.4 is incorrect. Both are discussed following the suggestion list.

S.1. If an 'addCommitment' intention i occurs after another 'addCommmitment' or 'removeCommitment' intention i_2 whose 'flyToTarget' attribute value is equal to i 'flyToTarget' attribute value, then i_2 affects i. (from Heuristic H.7)

S.2. If a 'commitments' message m is sent after an 'addCommitment' intention i, i has a 'commitments' attribute whose value is equal to m's 'commitments' value, then i affects m. (from Heuristic H.6)

S.3. If a 'flyToTarget' action a occurs after an 'addCommmitment' or 'removeCommitment' intention i that has a 'flyToTarget' attribute and has a's 'target' precondition attribute value for one of its attributes values, then i affects a. (from Heuristic H.4)

S.4. : If a 'commitments' message m is received from agent a after a 'commitments' message m_2 was sent to a and both messages have a 'commitments' attribute, then m_2 affects m. (from Heuristic H.5)

S.5. If an 'other_commitments' belief b occurs after a 'commitments' message m and both have equal 'commitments' attribute values, then m affects b. (from Heuristic H.2)

S.6. If an 'other_servicedTarget' belief b occurs after a 'scannedTarget' or 'servicedTarget' message m and both have equal 'target' attribute values, then m affects b. (from Heuristic H.2)

S.7. If an 'uavScan' event e occurs after a 'flyToTarget' or 'spiralSearch' action a and a has a 'target' precondition attribute value that is equal to e's 'target' postcondition attribute value, then a affects e. (from Heuristic H.3)

S.8. If an 'servicedTarget' belief b occurs after a 'uavScan' event e and e has a 'target' postcondition attribute value that is equal to b's 'target' attribute value, then e affects b. (from Heuristic H.2)

S.9. If a 'removeCommitment' intention i occurs after a 'servicedTarget' (must also have i's 'removeTarget' attribute value as one of b's attribute values), 'other_servicedTarget', or 'other_commitments' belief b, then b affects i. (from Heuristic H.7)

S.10. If a 'scannedTarget' message m is sent after a 'servicedTarget' belief b and both have equal 'target' attribute values, then b affects m. (from Heuristic H.6)

S.11. If a 'servicedTarget' message m is sent after a 'servicedTarget' belief b and both have equal 'target' attribute values, then b affects m. (from Heuristic H.6)

S.12. If an 'spiralSearch' action a occurs after a 'flyToTarget' action $a2$ and both have equal 'target' precondition attribute values, then $a2$ affects a. (from Heuristic H.4)

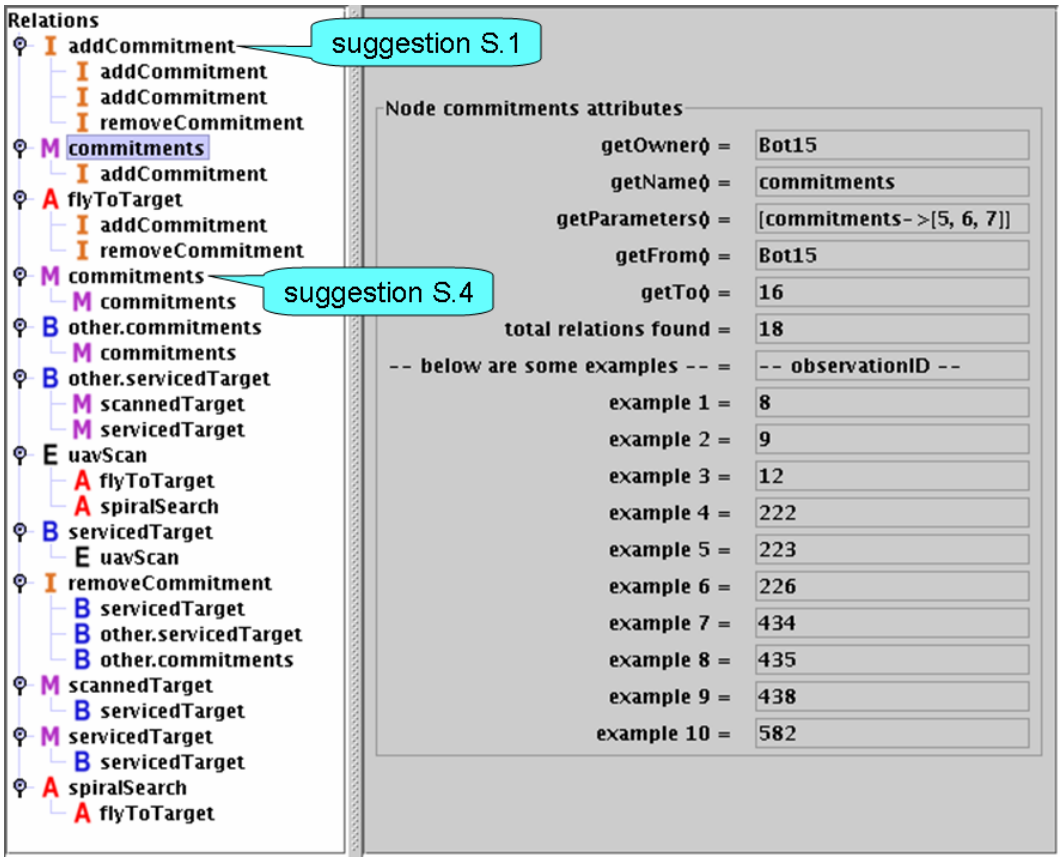


Figure 4.13: Relations suggested by Tracer

For each relation, if the suggested relation is judged correct by the user, the relation is added to K. In this experiment, the correctness of a relation was also verified by the developer of the agent software.

Some incorrect relations can be corrected by the user. For example, Suggestion S.1 relates 'addCommitment' and 'removeCommitment' intentions to future 'addCommitment' intentions with the same 'flyToTarget' attribute values. However, when relating the 'addCommitment' intention i to a 'removeCommitment'

intention i_2, the observations i and i_2 will not have the same 'flyToTarget' attribute values since the target has already been removed from the agent's intention. Figure 4.14 shows an interpretation when using the erroneous rule. The error is clearly apparent because intention nodes do not lead to (i.e., cause) future observations, whereas it is expected that all intentions should lead to some future intention or action. The corrected suggestion is:

> S.1' : If an 'addCommitment' intention i occurs after another 'addCommitment' i_2 whose 'flyToTarget' attribute value is equal to i's 'flyToTarget' attribute value or after a 'removeCommitment' intention i_2, then i_2 affects i.

Figure 4.15 shows an interpretation after Suggestion S.1 was corrected. Intention `I:414` now correctly leads to `I:429`, instead of ending the causal chain as in Figure 4.14.

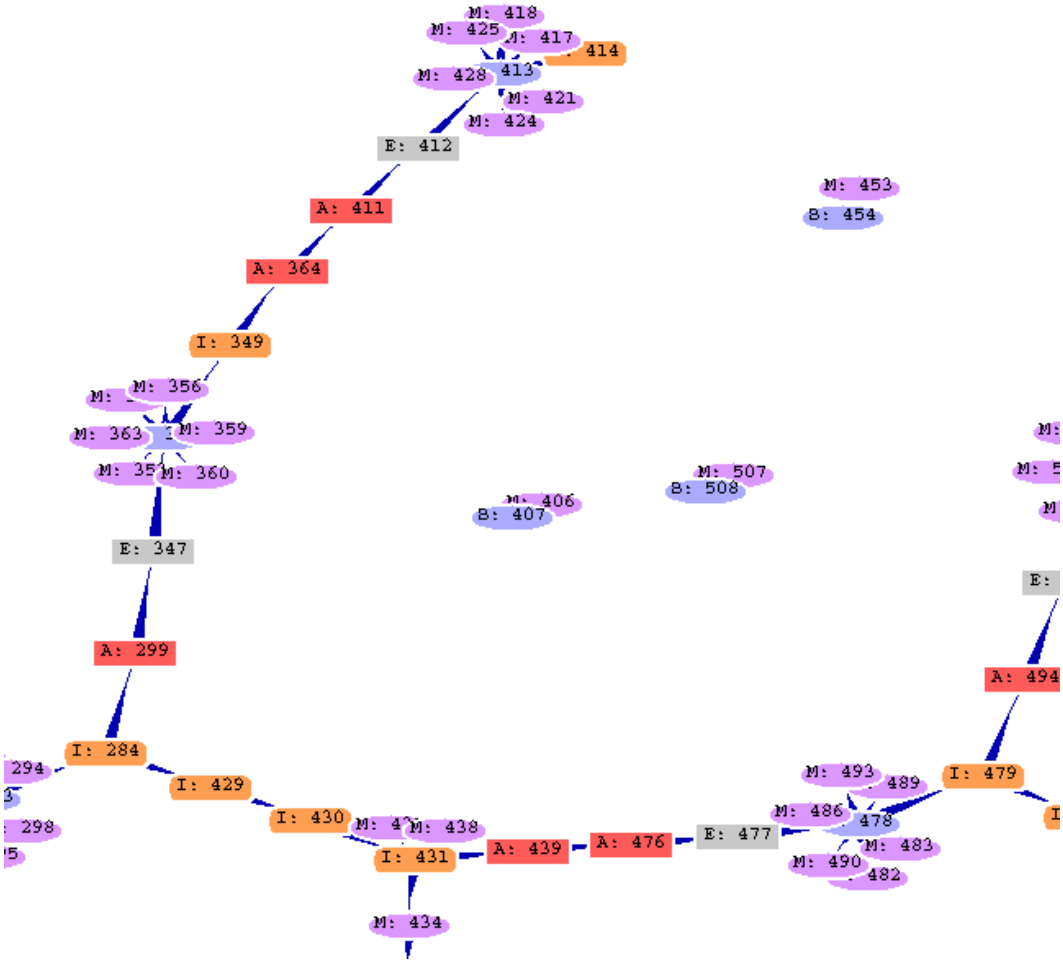


Figure 4.14: Before correction of Suggestion S.1

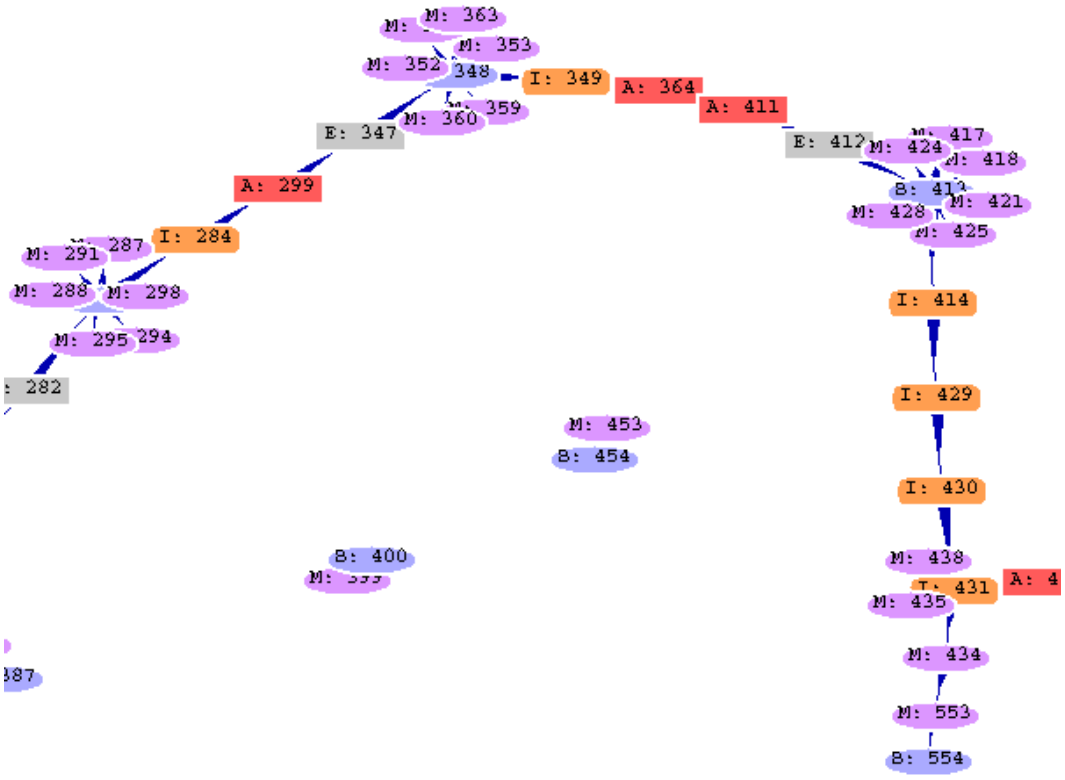


Figure 4.15: After correction of Suggestion S.1

While some suggestions can be corrected, other suggestions simply do not make sense in the application domain and are not added to K by the user. For example, Suggestion S.4 (*If a 'commitments' message m is received from agent a after a 'commitments' message m_2 was sent to a and both messages have a 'commitments' attribute, then m_2 affects m.*) defines a relation from a received 'commitments' message to a sent 'commitments' message – in other words, a reply to the received message. The relation S.4 was suggested because two messages with the common attributes (i.e., their 'commitments' name) happen to occur consecutively. However, 'commitments' message are sent only because an agent is telling other agents the intentions to which it has committed. So the relation to the 'commitments' message should originate from an intention; the relation of which has been defined in Suggestion S.2 (*If a 'commitments' message m is sent after an 'addCommitment' intention i, i has a 'commitments' attribute whose value is equal to m's 'commitments' value, then i affects m.*). Since S.2 creates the correct relation, Suggestion S.4 can be removed.

Though suggestions greatly facilitate the construction of K, some relations that are specific to the application domain must be manually created to build a complete K. For example, the following manual relation was added to relate all beliefs observed after the previous intention to the current intention. In other words, all new beliefs observed after the last intention were used to create the current intention.

- For all beliefs after the previous intention q, if the belief b occurs before an intention p, then b affect p.

As a result of this manually created relation, all the unconnected belief nodes in Figure 4.15 are connected to the appropriate intention, as shown in Figure 4.16.

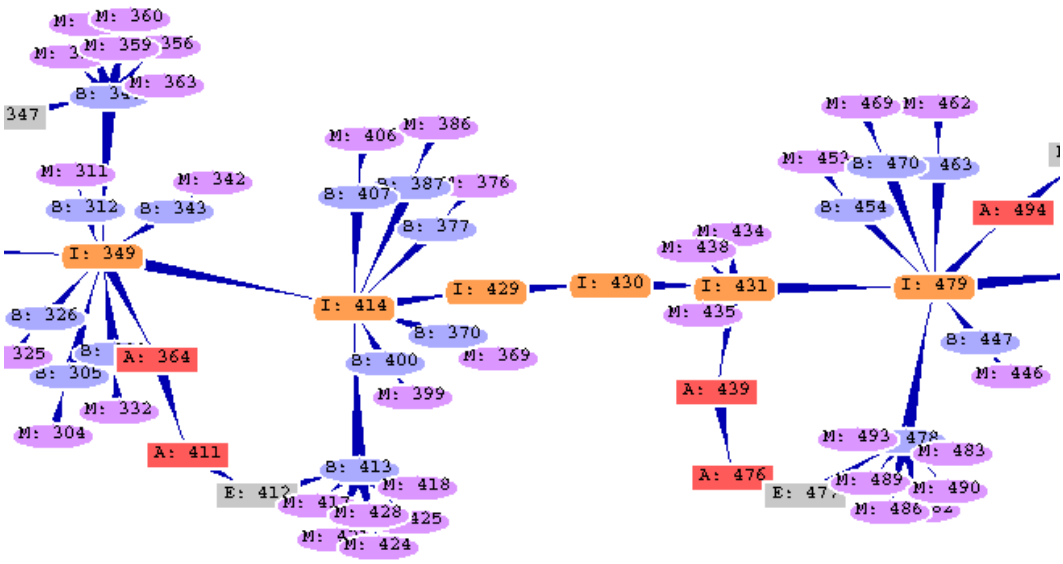


Figure 4.16: Interpretation with complete and correct K

An interpretation with the final K is shown in Figure 4.16. In total, one relation (actually, a subrelation) was modified because it was too restrictive; one relation was removed because it did not make sense; and one relation was manually added because domain knowledge was needed to create the relation.

Table 4.2 summarizes the number of relations that was correctly suggested, incorrectly suggested, and manually-created for each agent system. For more precise measurements, the number inside parentheses is the number of subrelations. A set of subrelations define several causes for a single effect. The results show that the relation-suggesting algorithm captures approximately 71% of the relations in the Simple system and approximately 83% in the UAV system with relatively few incorrect suggestions. The algorithm shows promise in automating the tedious task of associating observations with each other.

	Simple	**UAV**
number of agent concepts	17	22
correctly suggested relations	10 (11)	10 (16)
partially correct suggested relations	0 (0)	1 (1)
incorrectly suggested relations	1 (1)	1 (2)
manually-created relations	4 (4)	1 (1)
total suggested relations	11 (12)	12 (19)
total relations	14 (15)	12 (18)
correctness = correct/(total suggested)	91% (92%)	83% (84%)
completeness = 1 - (partial + manual)/total	71% (73%)	83% (89%)

Table 4.2: Results for the relation-suggesting algorithm

Note that automated relation-suggesting is possible because of the attribute names for each agent concept. For example, if an attribute name for an belief is 'target' while an attribute name for an event is 'vehicle', then a relation for these two agent concept will not be suggested based solely on those different attributes – a manual relation must be created.

Additional heuristics can be added to the relation-suggesting algorithm so that more relations can be identified automatically, 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. By adding extensions to the Tracer Tool, the user can (1) control which heuristics are used to determine if two observations are possibly related and (2) define additional heuristics for their particular domain type.

The results of Experiment 1 shows the extent of automation for building a correct background knowledge using the prescribed heuristics. This experiment suggests that the heuristics can sufficiently identify most relations in the background knowledge. For the domains used in this experiment, most of the background

knowledge can be correctly suggested, thus reducing the time and effort invested in comprehending the system. The relations that were manually modified or created were specific to the domain application and, thus, heuristics for suggesting such relations should not be included in the heuristics list.

4.4.2 EX2: Detecting representative inaccuracies

The interpreting algorithm (shown in Listing 4.4) creates an interpretation by relating observations to each other based on the relations defined in the background knowledge K. This experiment addresses the representative accuracy of K and is designed to identify the types of errors that can be detected using the interpretation and how such errors can be remedied.

The difficulty of diagnosing apparent representative errors in the background knowledge K is that K may not be the actual source of the error. For example, if K has a relation connecting two concepts and the relation does not exist in the interpretation, then the user's comprehension K or the implementation or both may be incorrect. If K represents the developer's comprehension of how the system *should* behave from the design specification, then K is correct and the implementation needs to be modified to produce the expected behavior.

Within the scope of this research, there are several sources of errors, which can result in faulty reasoning (e.g., inaccurate explanations of software behavior). Using the variables in Equation 3.1 ($O_s = observe(execute(I, s))$) and Equation 3.3 ($N_s = interpret(K, O_s)$) as guidelines, the possible error sources include the following:

E.1. incomplete or incorrect K: missing or incorrect agent concepts or relations in the background knowledge;

E.2. missing observation in O_s: wrong logging code placement (the location where the logging code was inserted by the user is incorrect);

E.3. incomplete or buggy I: the implementation is incomplete or has a bug; and

E.4. insufficient S: the set of execution scenarios does not provide adequate coverage of agent activities.

To determine which errors can be detected using interpretations, different type of errors were implanted and the interpretation was examined. By studying the effects of various errors in the interpretation, error detection can be automated and possible solutions can be presented to the user. This experiment begins with a background knowledge K that accurately represents the implementation I in a set of scenarios S. Thus, K, I, and S are assumed to be correct and complete as far as this experiment is concerned. Based on the possible error sources, the following errors were implanted into the otherwise accurate K, correct I, and complete S:

P.1. $K + node$: add node to K; add an agent concept that doesn't occur in the implementation in any scenario. This imitates the user having incorrectly assumed knowledge of agent concepts about the system.

P.2. $K + edge$: add edge to K; add a relation that doesn't occur in the implementation in any scenario. This imitates the user having incorrectly assumed knowledge of relations between agent concepts about the system.

P.3. $O - node$: remove observation from O_s; remove logging code from agent system source code. This imitates the user inserting logging code in a non-executing portion of the source code (for a particular scenario or for all scenarios).

P.4. $misplace(o)$: reorder the sequence of observations in O_s; move logging code to a different location in the source code such that the observation is out of

order and may have incorrect data. This imitates the user inserting logging code in the wrong location in the source code.

P.5. $I - behavior$: remove behavior in I; remove whole sections of source code to disable agent behavior. This imitates an incomplete implementation of the agent system.

P.6. $mixup(I)$: change I's behavior; rearrange source code to cause incorrect agent behavior in the implementation. This imitates an incorrect implementation of the agent system.

P.7. $S - scenario$: remove scenario from S; do not execute the agent system in a particular scenario. This imitates an insufficient scenario set that doesn't cover all expected agent behaviors.

Other possible errors, such as removing a node or edge in K and adding an observation to O, were not implanted because they were implicitly addressed in Experiment 1 (EX1). In EX1, K was initially empty and as new observations were logged, agent concepts were added and relations were suggested to update K. The Tracer Tool offers suggestions for all observations without a relation coming from another observation – nodes with no incoming edge. Therefore, such "errors" can be directly remedied by the Tracer Tool.

For each implanted error, the agent system was executed and interpretations were created. Since the interpreter uses K and O_s as input, any errors in K or O_s will be exhibited in the interpretation N_s. Table 4.3 shows the effect of each implanted error on the interpretation. Occurrence counts are the number of times an agent concept or relation occurs in an execution scenario. All errors had some identifiable effect on the interpretation.

Implanted Error	**Effect**
P.1 ($K + node$)	The interpretation is unaffected. Tracer shows occurrence count is 0 for added agent concept.
P.2 ($K + edge$)	The interpretation is unaffected. Tracer shows occurrence count is 0 for added relation.
P.3 ($O - node$)	The observation and all edges connecting the observation are missing in the interpretation. Tracer makes a suggestion if a related observation does not have an incoming edge.
P.4 ($misplace(O)$)	All edges connecting the reordered observations are missing in the interpretation. Tracer makes a suggestion if a related observation does not have an incoming edge.
P.5 ($I - behavior$)	Same effects as P.3 if associated logging code was also removed. Otherwise, no effect.
P.6 ($mixup(I)$)	Same effects as P.4 if associated logging code was also moved. Otherwise, no effect.
P.7 ($S - scenario$)	Tracer shows occurrence counts is 0 for agent concepts and/or relations that occur only in the removed scenario.

Table 4.3: Effects of implanting errors

Note that some errors can cause the same effect. For example, P.3 and P.5 have the same effect on the interpretation if the associated logging code was modified along with the source code. The solution to both these errors is to look at the same place in the source code where the observation was recorded. Implanted errors P.4 and P.6 can also have the same effect. This is not surprising since observations in O are logs of the implementation I's behavior. If the implementation changes significantly, then the sequential order, frequency, and content of the observations will be affected. However, it is important to distinguish the sources of the errors. In other words, the effect of P.3 does not imply that P.5 was the error source. P.3 imitates incorrect insertion of the logging code (performed by the user

of Tracer), whereas P.5 imitates an incomplete system implementation (performed by the software developer).

To diagnose problems detected in the interpretations, Table 4.4 enumerates possible problems that can be seen in the interpretation, along with their respective cause(s). *Node* refers to an observation in the context of an interpretation N_s and refers to an agent concept in the context of background knowledge K; and *edge* refers to a relation between nodes.

problem	**cause**
node in K is missing in N_s	user incorrectly added the concept to K (error E.1); OR wrong logging code placement (error E.2); OR implementation is incomplete (error E.3); OR observation does not occur in the scenario s (error E.4).
edge in K is missing in N_s	user incorrectly added the relation to K (error E.1); OR wrong logging code placement (error E.2); OR implementation is incomplete/incorrect (error E.3); OR the relation does not occur in the scenario s (error E.4).
node in K is missing in all N_s	user incorrectly added the concept to K (error E.1); OR wrong logging code placement (error E.2); OR implementation is incomplete (error E.3); OR the observation does not occur in any scenario in S (error E.4)
edge in K is missing in all N_s	user incorrectly added the relation to K (error E.1); OR wrong logging code placement (error E.2); OR implementation is incomplete/incorrect (error E.3); OR the relation does not occur in any scenario in S (error E.4)

Table 4.4: Possible completeness and consistency problems between K and N_s

Note that it is not possible to have a node in N_s that has no corresponding agent concept in K since every inserted logging code originate from an agent concept in K. The result is that there cannot be anything in N_s that is not already in K. This is also true for edges (or relations) between nodes since edges are created only

Cause	Solution
E.1 (incomplete or incorrect K)	K models a superset of behaviors exhibited in N_s. Remove from K agent concept or relation with occurrence count=0 (as shown by Tracer).
E.2 (missing observation in O_s)	Make sure the inserted logging code is in the correct section of code that it represents and that the code is executing for the current scenario.
E.3 (incomplete or buggy I)	I is incomplete (with respect to K) or incorrect (buggy). Add the feature to I, or use Tracer's explanation generator to help locate the bug and fix it.
E.4 (insufficient S)	S does not have sufficient coverage. Add a scenario in which the concept or relation occurs in S.

Table 4.5: Solutions to causes of problems in Table 4.4

if a corresponding relation exists in K. Thus, such problems in the interpretation do not arise. For example, if a person does not have the comprehension that pollen is in the air and that pollen can cause sneezing, then that person's interpretation of the environment and his sneezing will not consider the existence of pollen and its relation to sneezing.

Table 4.4 shows the possible problems that can occur, along with their possible causes. As seen in the table, the source of the problem cannot be easily determined from the problem. For example, if the problem is that a node in K is missing in N_s (i.e., an expected agent concept is not observed), then the cause of the problem could be (E.1) the agent concept should not be in K, (E.2) logging code for the agent concept was misplaced, (E.3) I does not implement the agent concept, or (E.4) the scenario s does not induce the observation to occur. If an edge in K is missing in N_s, then similar causes may be possible. If a node or edge in K is missing in all interpretations N_s for $s \in S$, then it is possible that (E.4) none of the scenarios in S contain a scenario that induces the observation to occur.

Some further investigation is required on the part of the user to find the source of the problem through the process of elimination. Table 4.5 shows solutions for the causes listed in Table 4.4. For the cause E.1, the Tracer Tool helps the user identify the extraneous concept or relation in K by showing an occurrence count of 0 for the particular agent concept or relation. As a solution, the user can remove the offending agent concept or relation from K. In other cases, the user can ensure that the logging code is correct (for E.2), add a feature corresponding to the concept or relation in I (for E.3), or add a new scenario to S (for E.4).

This analysis shows that Tracer can detect representative inaccuracies of the user's comprehension (or background knowledge K) by creating an interpretation N_s of actual behavior and analyzing N_s. In other words, the user's expected behavior of the agent software system is verified against the implementation's actual behavior. This research provides a method to maintain the representative accuracy of K as the implementation I changes throughout the software lifecycle by suggesting updates to K and detecting unexpected or anomalous behavior in the implementation's behavior. As a result of automating the software comprehension process, the Tracer Tool can be used to build the background knowledge and check its representativeness against the implementation.

4.5 Contributions from RQ1

Generating explanations is dependent on the quality of the background knowledge, specifically on how accurately the background knowledge reflects the context of what is being explained. The contribution from answering RQ1 is a method to produce a formal model (called the background knowledge K) that (1) accurately represents the actual system (i.e., the implementation, even as it changes) in terms of agent concepts familiar to the designer, developer, and end-user, (2) ex-

plicitly represents the user's growing knowledge of the software's behavior, and (3) can be used for automated reasoning.

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 verifying expected behavior and detecting unexpected behavior, are automated. With the resulting background knowledge K, explanations of actual agent behavior that are consistent with run-time observations can be produced, which leads to the next research question.

4.6 Assumptions and Limitations

This research makes some assumptions about the user's ability to instrument the implementation, provide a sufficient scenario set, and determine the extent of agent concepts to record. Additionally, this research approach does not perform an exhaustive verification of system behavior. The following briefly describes and comments on these points.

- The method is dependent on the user's ability to correctly insert logging code in the appropriate source code location. To assist the user in identifying relevant parts of the source code, a set of design-level agent concepts (e.g., beliefs, intentions, and actions) are provided. Since high-level information (e.g., entity relationship diagrams, data flow diagrams, and process diagrams) about the implementation is usually provided in some software design, the user can study the design instead of the implementation directly. Using agent concepts that are referenced in the software design, the user has a better handle on where to insert logging code in the implementation. Since it is not cur-

rently possible to automate logging-code insertion, this assumption must be made.

- The approach of this research does not perform an exhaustive state-space search to verify software behavior (as model-checking does). Instead, a more practical approach is provided for complex systems whose state-space is too large or not feasible to be verified. In accordance with Edmonds' emphasis on the need for empirical analysis of agent behavior, such as scenario-based analysis and field testing [Edmonds and Bryson, 2004], the Tracer Tool analyzes the implemented system (within the scope of the scenario set) rather than a model of the implemented system. Additionally, this approach focuses on whether the implementation and the comprehension of the system are consistent, rather than on whether the model of the system behaves correctly. Collaborative work has been recently performed to integrate a property checker called TTL Checker with the Tracer Tool [Bosse et al., 2006]. The TTL Checker provides automated checking of agent or system behavioral properties, similar to model-checking except the input is a set of observations from the system's execution rather than a formal model of the system. The execution traces are checked against properties about the agents' behaviors, written in the predicate logic Temporal Trace Language (TTL) [Jonker and Treur, 2002] using agent concepts terminology. Thus, a user can discover behavioral anomalies from Tracer's graphical interface or be alerted by the TTL Checker.

- To focus the scope of this research, the user is relied upon to provide the set of scenarios S in which the agent system is executed. This is a typical and onerous problem faced by software testers and is not directly addressed in this research. It is only within this set that the background knowledge K

can be said to be representative of the implementation's behavior (as represented by the scenario's interpretation N_s). The Tracer Tool cannot know whether the scenarios provide adequate coverage of agent behaviors unless those behaviors (in the missing scenarios) are represented in K, in which case non-covered behaviors are those that involve agent concepts or relations with no occurrence counts (as discussed in Table 4.5 for Error E.4). In other words, the Tracer Tool can only check what the user has modeled in K. The Tracer Tool can aid in populating K based on observations from the implementation's executions through all scenarios in S. However, the user must determine whether K sufficiently models all agent behaviors that are relevant to the user.

- On a related issue, there is no metric to determine the appropriate amount of agent concepts for representing K, the user's comprehension of the system. The amount depends on the reason that software comprehension is being performed. For example, if the purpose is to have a general understanding about what the agents are doing and why they are performing certain actions, then every agent belief may not need to be logged. However, if the purpose is to debug a specific action, then the user may want to log all agent beliefs that could have affected the decision-making process which resulted in that action. The guideline is to log all agent concepts mentioned in the software design such that there exists a mapping for every agent concept in the design to an implementation construct.

Chapter 5

Research Question 2: Explaining Agent Behavior

explanation (n): (1) a statement that explains; (2) thought that makes something comprehensible

[1997 WordNet 1.6]

5.1 Approach to Research Question 2

How can background knowledge be used to explain observed agent behavior?

Explanations of agent actions offer an understanding of why agents behave in a certain way in a given scenario. An explanation of agent behavior answers a question like "Why did agent action m occur?" A desirable explanation could be "Action m was performed by agent n_1 because n_1 held belief b, which was due to the occurrence of event e, which was an expected consequence of agent n_1 performing action a, which was planned as a result of negotiations with agent n_2 about n_2's goal g." Other relevant agent concepts can include details about the negotiations, such as the communication messages and updated beliefs resulting from the messages.

Since there is no direct way to measure how much the user comprehends, a person's comprehension of a subject is indirectly measured by how much the person can explain about the subject because the process of creating an accurate

explanation demands correct comprehension of the system. Explanations bridge the gap between expected and actual behavior (i.e., between the explainer's background knowledge and the implementation's execution). Thus, explanations can be very important in designing, debugging, and trusting agent behavior.

Unfortunately, ensuring accurate explanations is difficult because the implementation evolves over time and there are many factors that can influence agent behavior. First, since comprehending the behavior of the implemented system relies on how accurately the background knowledge represents the implementation, the representative accuracy of the background knowledge must be maintained as the implementation changes. The second problem of manual explanation generation is that an explanation may be too difficult to conceive due to the sophistication (e.g., in reasoning or agent interaction) of the agent system or the amount of observed data to consider. In response to these difficulties, this research proposes an automated approach to agent software comprehension that can handle large amounts of observation data and can automate the generation of explanations to aid the user in comprehending the system as the implementation evolves over time.

Since background knowledge K has been checked for representative accuracy over the chosen set of scenarios S as described in Chapter 4, K can be leveraged to accurately explain any observation (called the manifestation $m \in O_s$), such as an agent action. An explanation ϵ consists of a subset of observations from O_s and relations among those observations that contributed to (i.e., caused or influenced) the occurrence of m. The relations among those observations are derived from K, which defines relations among agent concepts. Thus, explanation generation involves mapping observations to agent concepts and following the relations (backwards) from m to observations that caused m.

From Equation 3.6 ($\epsilon = explain(m, K, O_s) = explain(m, N_s)$), generating explanations is dependent on the quality of the K, specifically on how accurately K reflects the context of what is being explained – thus, stressing the importance of maintaining representative accuracy between K and I as described in Chapter 4. Based on the approach illustrated in Figure 5.1, an explanation ϵ for manifestation $m \in O_s$ (e.g., agent action) can be generated using the checked background knowledge K and observations O_s. To generate an explanation for a manifestation m, the explainer uses the same technique as in interpretation – mapping observations to agent concepts specified in K and using relations in K to link observations together. If an interpretation N_s of the scenario has been created, the same explanation can be generated faster using N_s because $interpret(K, O_s)$ has already done the work of mapping and relating the observations. Starting from the manifestation m in the interpretation N_s, the explanation is generated by identifying observations that cause or influence the occurrence of m by following edges pointing to m. This can be performed recursively to an arbitrary depth to find causes of causes.

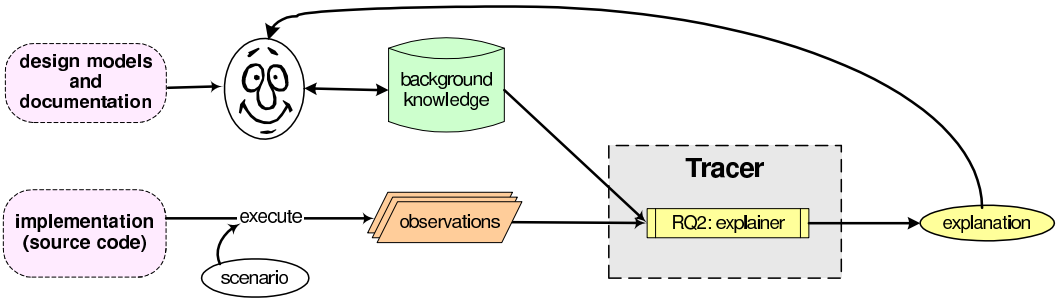


Figure 5.1: Approach for Research Question 2

Since the background knowledge is expressed in terms of agent concepts listed in Section 2.2, the resulting explanations will be expressed in terms of the

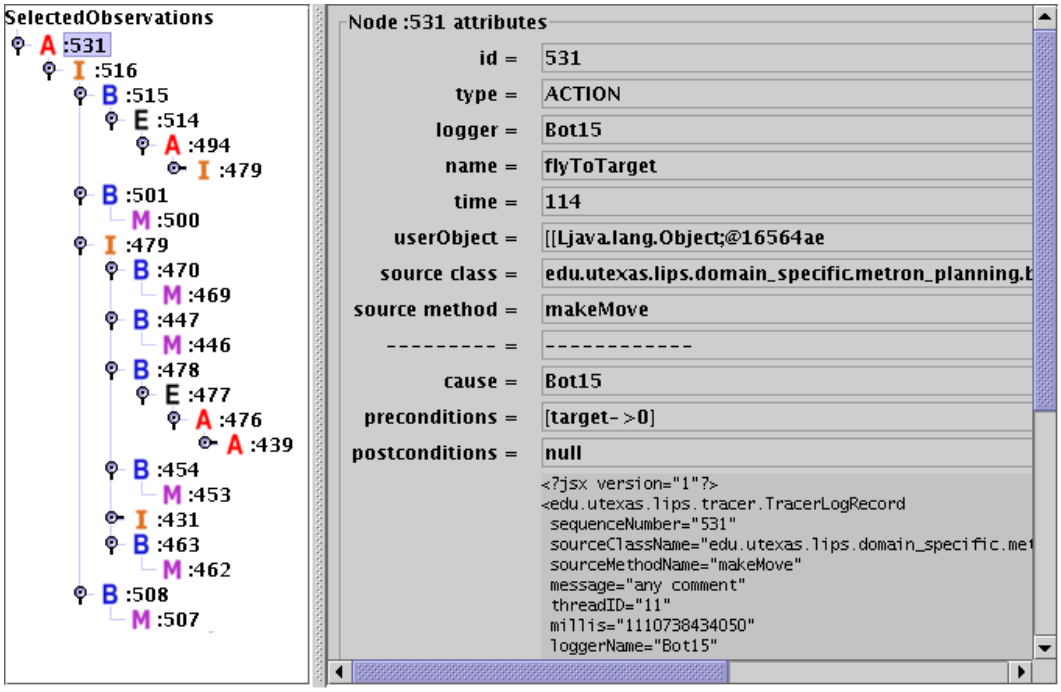


Figure 5.2: Explanation in Tracer

same agent concepts. The explanation can be expressed as a tree (as shown in Figure 5.2) where the root of the tree is m. Child nodes are observations that influenced or caused the parent node observation to occur. The depth of the explanation tree can continue 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 tree does not have a leaf node with one of these observations, then K may be incomplete and may require relations to be added. In the previous chapter, Section 4.4.2 demonstrates the relation-suggesting algorithm used to help build a complete K.

Explanations can help focus on and track down the cause of a particular

undesirable behavior. With automated explanation generation readily available to the user, tasks such as redesigning, debugging, and understanding agent behavior becomes a more manageable task and less prone to human error.

5.2 Tracer's Explainer

In the process of building, verifying, and refining the background knowledge K, anomalous behavior may be discovered. Anomalous behavior manifests itself as (1) observations (or nodes) without incoming relations or (2) relations between observations that do not semantically make sense in the particular application domain. The source of these inconsistencies could be any of the errors mentioned in Section 4.4.2. 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 assist in tracking down bugs or sources of anomalous behavior in the implementation, the Tracer Tool can generate explanations for a specified manifestation m (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 N_s for the scenario s in which m occurred. Starting from the manifestation m being explained, incoming edges are followed to other observations that caused or preceded m.

Listing 5.1 shows the algorithm for generating an explanation ϵ using interpretation N_s. The algorithm is essentially a depth-first search starting from observation m, following the directed edges backwards. Each observation has information about when the observation occurred, where in the source code it executed, and other run-time data about the observation. Such data is collected to map agent con-

Listing 5.1: Explanation generation algorithm

```
buildExplanation(OBSERVATION m, GRAPH N_s, EXPLANATION
  ε) {
  for each RELATION r within N_s thatPointsTo m {
    OBSERVATION o = getOriginOfRelation(r);
    ε.addObservation(o, m); // add observation o under
        m
    buildExplanation(o, ε)
  }
}
```

cepts to specific source code fragments and, thus, aid the user in locating errors in the implementation.

As seen in Figure 5.3, 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`.

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 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 [Lam and Barber, 2004]. In that paper, relations in K

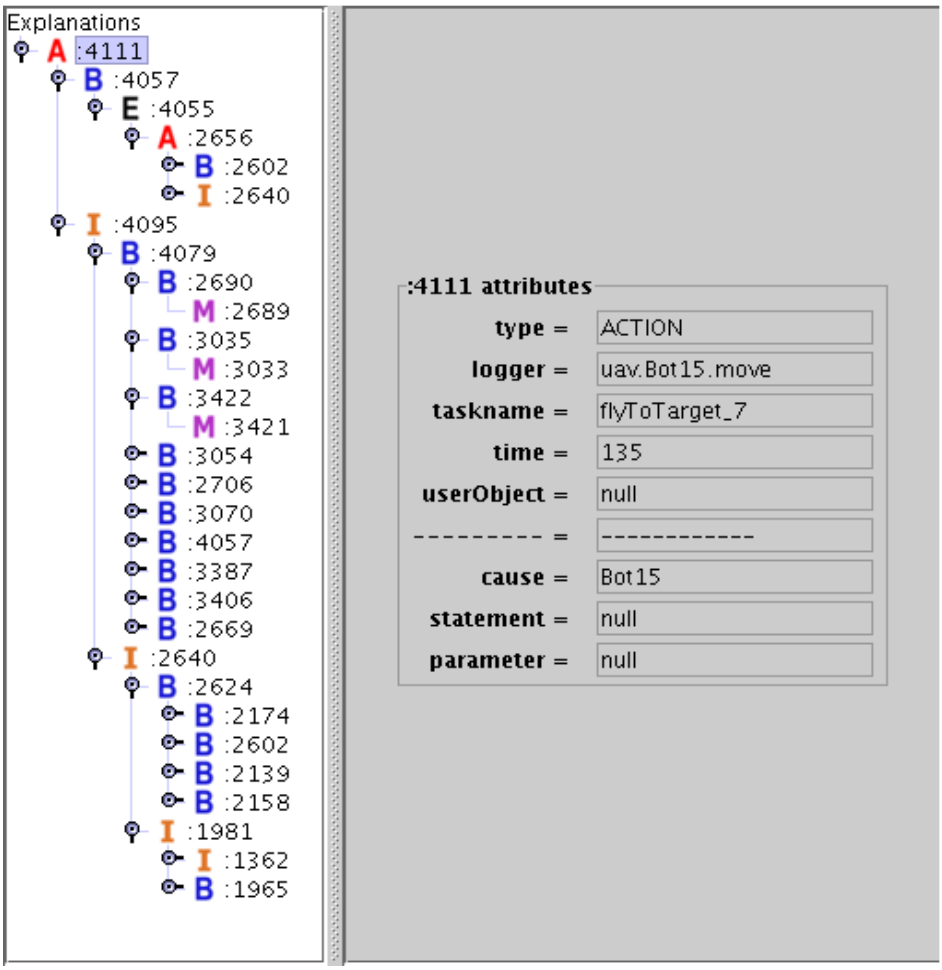


Figure 5.3: Explanation of action A:4111 from Figure 4.10

were manually specified as rules, each defining the relation between observations. In this dissertation, relations are suggested automatically as shown in Experiment 1 (EX1).

5.3 RQ2 Experiment

With the explanation generation capability of the Tracer Tool available, this section discusses an experiment that directly addresses the hypothesis – *accurate explanations of actual agent behavior can be generated if models of expected agent behavior being used as background knowledge are representative of the implementation's behavior*. Experiment 3 (EX3) shows that if the background knowledge is not representative of the implementation's behavior, then the explanations may not be accurate. This experiment also demonstrates the utility of explanations to clarify and correct the user's understanding of the system.

5.3.1 EX3: Generating explanations

The objective of Experiment 3 is to investigate the affect of representative inaccuracies of background knowledge on the accuracy of explanations. The experimental method is similar to EX2. Errors are implanted (in varying types and degrees) in the background knowledge K to simulate inaccurate background knowledge. In order to focus on changes in the background knowledge, the implementation is assumed to be correct, just as if an end-user is trying to comprehend what and why the implemented agents are behaving the way they are. Thus, any inaccuracies in the explanation are not a result of implementation errors. According to the hypothesis, the explanations should be inaccurate if the background knowledge is inaccurate.

The following lists several ways to introduce errors into the background knowledge K. Note that the implementation's source code was not modified – no logging code was added or removed.

K.1. add extra agent concept: This simulates a user mistakenly believing there exists an extra agent concept in the implementation's behavior. For example,

the user may believe (for some reason) that an agent performs the 'attack target' action, but that action is not incorporated into the implementation. Since there is no part of source code that corresponds to the agent concept, no observations will be recorded from the implementation execution. Thus, the observation will not appear in the explanation and there will be no affect on the explanation. This is the same as implanted error P.1 in EX2 (adding a node to K).

K.2. remove agent concept: This simulates a user being unaware of the agent concept. Thus, the explanation will not have any reference to the observation corresponding to the agent concept. By removing an agent concept, all connected relations must also be removed. Thus, this type of error is reduced to the 'remove relation' error K.4 below. If the agent concept has no relations to other agent concepts, then observations corresponding to the agent concept will never be referenced as a cause (of another observation) in the generated explanation.

K.3. add extra relation: This simulates a user mistakenly believing a relation exist between two agent concepts (e.g., due to coincidence or misunderstanding) when there is no actual relation.

K.4. remove relation: This simulates a user being unaware of a relation between two agent concepts.

Table 5.1 summarizes the expected effect of these types of errors on the any generated explanation. Errors K.1 and K.2 are not used in this experiment as they have no effect on the explanation or can be reduced to another type of error, respectively. Consequently, only *relations* will be added or removed from K – only errors K.3 and K.4 will be implanted. In addition to the different types of errors, there are

implanted error	**effect on explanation ϵ**
K.1 ($K+$ *concept*)	None. Since no observation corresponding to the agent concept is recorded, it will not appear in the explanation.
K.2 ($K-$ *concept*)	None, directly. By removing an agent concept, all connected relations must also be removed from K. Thus, the explanation will not have any reference to the observation corresponding to the agent concept.
K.3 ($K+$ *relation*)	More relations will be created in ϵ if observations referenced by the added relation occur.
K.4 ($K-$ *relation*)	Relations will be removed in ϵ if observations referenced by the removed relation occurred in the baseline explanation.

Table 5.1: Expected effect of erroneous K on explanations ϵ

different degrees to which these errors are implanted. This experiment shows the affect of increasing number of errors in K on the generated explanation.

The UAV domain was used in this experiment, however, the conclusions from this experiment are domain-independent. There are a total of 16 relations in the background knowledge K for the UAV domain, as shown in Figure 5.4. The chosen execution scenario includes agent behaviors that use all agent concepts and relations in the K so that all elements in K are referenced in the generated explanation.

Three trials with different selected relations are performed. Each trial involves adding and removing relations (chosen at random) to K. To determine what relation to add, two agent concepts in K are randomly selected and a new relation is created from the first to the second agent concept. No duplicate relations were added since it would not affect K. Within a trial, the same random seed is used to maintain repeatability over executions that use different versions of K. An explanation of the same observation is generated for each execution. In particular, an action observation was chosen to be explained. By making the UAV simulation

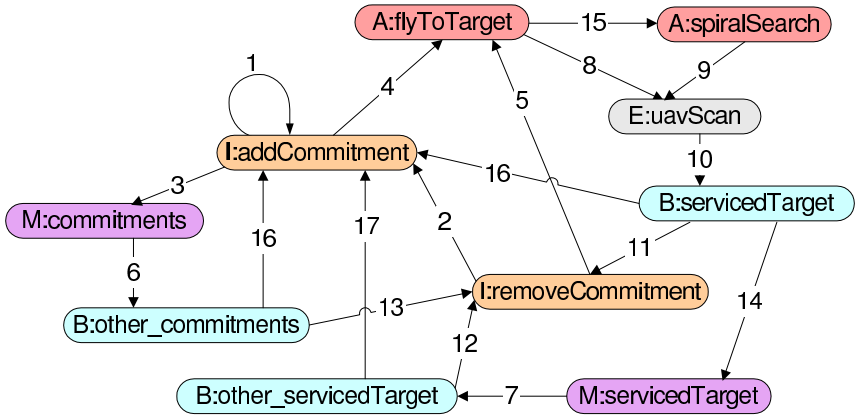


Figure 5.4: Background knowledge for the UAV domain

deterministic (i.e., using the same random seed and removing race conditions) and by explaining the same observation, explanations across different executions can be automatically compared since the size of the explanations can be very large. A count of the number of differences for each type of difference (either missing or extra relation) was recorded for each change to K. Otherwise, if the simulation was non-deterministic, then equivalent observations from different executions usually have different identifiers and automated comparison would not be more difficult.

There was no bound set for the depth of the explanations. This resulted in explanations referencing large numbers of observations (e.g., 213 and 404). The evaluation metric used in this experiment is the number of incorrect relations (i.e., either missing or extra relations) in the explanation.

The following describe the randomly chosen relations and parameters of the different trials. The relations labeled with negative numbers were removed one after the other in the order of descending integer value (e.g., -1, then -2, then -3, etc) until the generated explanation has no correct relations. For example, if the

baseline explanation (the completely accurate explanation) has 213 relations, then relations are removed from K until all 213 relations are missing from the resulting explanation.

The relations labeled with positive numbers were added sequentially in increasing order. Six relations were sufficient to show the trend in the explanation accuracy. Table 5.2 summarizes each trial's timesteps for which the execution was ran, the random seed used, the number of relations in the baseline explanation, and the id of the relations that were removed (in the order they were removed).

Trial	**Timesteps**	**Random seed**	**Baseline**	**Relations removed**
Trial 1	200	3	213	8,1,9,10,2,7,6, 11,14,12,3,4,16
Trial 2	400	29	404	4,1,10,9,5
Trial 3	400	29	404	10,1,9,5,8,2, 15,13,3,12,4

Table 5.2: Experiment 3 trials

Note that Trial 3 has the same setup as Trial 2, except that a sequence of different relations were removed to demonstrate the sensitivity of explanation generation due to inaccuracies in K. The following details each trial's experimental setup.

Trial 1 was executed for 200 timesteps with random seed 3. The baseline explanation has 213 relations.

-1. Relation 8: A:flyToTarget → E:uavScan
-2. Relation 1: I:addCommitment → I:addCommitment
-3. Relation 9: A:spiralSearch → E:uavScan
-4. Relation 10: E:uavScan → B:servicedTarget
-5. Relation 2: I:removeCommitment → I:addCommitment
-6. Relation 7: M:servicedTarget → B:other_servicedTarget
-7. Relation 6: M:commitments → B:other_commitments
-8. Relation 11: B:servicedTarget → I:removeCommitment

-9. Relation 14: B:servicedTarget → M:servicedTarget
-10. Relation 12: B:other_servicedTarget → I:removeCommitment
-11. Relation 3: I:addCommitment → M:commitments
-12. Relation 4: I:addCommitment → A:flyToTarget
-13. Relation 16: B:any → I:any

1. A:flyToTarget → I:removeCommitment
2. B:servicedTarget → M:commitments
3. B:other_servicedTarget → M:servicedTarget
4. E:uavScan → I:addCommitment
5. B:other_commitments → B:other_servicedTarget
6. M:scannedTarget → A:spiralSearch

Trial 2 was executed for 400 timesteps with random seed 29. The baseline explanation has 404 relations.

-1. Relation 4: I:addCommitment → A:flyToTarget
-2. Relation 10: E:uavScan → B:servicedTarget
-3. Relation 1: I:addCommitment → I:addCommitment
-4. Relation 9: A:spiralSearch → E:uavScan
-5. Relation 5: I:removeCommitment → A:flyToTarget

1. B:other_servicedTarget → E:uavScan
2. M:servicedTarget → M:servicedTarget
3. B:other_servicedTarget → I:removeCommitment
4. B:servicedTarget → B:servicedTarget
5. M:servicedTarget → I:addCommitment
6. M:servicedTarget → A:flyToTarget

Trial 3 was executed for 400 timesteps with random seed 29, same as Trial 2 except with different relations. The baseline explanation has 404 relations.

-1. Relation 10: E:uavScan → B:servicedTarget
-2. Relation 1: I:addCommitment → I:addCommitment
-3. Relation 9: A:spiralSearch → E:uavScan
-4. Relation 5: I:removeCommitment → A:flyToTarget

-5. Relation 8: A:flyToTarget → E:uavScan
-6. Relation 2: I:removeCommitment → I:addCommitment
-7. Relation 15: A:flyToTarget → A:spiralSearch
-8. Relation 13: B:other_commitments → I:removeCommitment
-9. Relation 3: I:addCommitment → M:commitments
-10. Relation 12: B:other_servicedTarget → I:removeCommitment
-11. Relation 4: I:addCommitment → A:flyToTarget

1. M:commitments → B:other_servicedTarget
2. B:other_servicedTarget → E:uavScan
3. A:flyToTarget → B:other_servicedTarget
4. M:servicedTarget → E:uavScan
5. M:commitments → A:flyToTarget
6. B:servicedTarget → E:uavScan

Results

Table 5.3 shows the number of relations in the explanations for the 3 trials. The first column shows the number of relations that were removed (negative) or added (positive) to K. Each row represents an execution of the system using the modified K. The row with 0 added relations represents the baseline for comparison. For each trial and each execution, there may be extra or missing relations in the generated explanation.

The data in Table 5.3 is shown as stacked bar charts in Figure 5.5, Figure 5.6, and Figure 5.7 for each respective trial. The charts show the number of relations added (positive direction) or removed (negative direction) along the X-axis and the number of incorrect (extra or missing) relations along the Y-axis.

	Trial 1 Explanations		**Trial 2 Explanations**		**Trial 3 Explanations**	
relations added to K	extra	missing	extra	missing	extra	missing
-13 (13 removed)	2	213				
-12 (12 removed)	7	116				
-11 (11 removed)	8	106			0	404
-10 (10 removed)	8	106			0	84
-9 (9 removed)	1	106			0	84
-8 (8 removed)	1	106			0	84
-7 (7 removed)	0	106			0	84
-6 (6 removed)	0	97			0	84
-5 (5 removed)	0	47	0	404	18	84
-4 (4 removed)	10	47	1	376	18	84
-3 (3 removed)	10	32	1	376	18	84
-2 (2 removed)	14	26	1	376	18	84
-1 (1 removed)	4	26	1	371	0	84
0 (baseline)	-	-	-	-	-	-
1 added	15	0	27	0	108	0
2 added	15	0	96	0	135	0
3 added	15	0	102	0	235	0
4 added	30	0	150	0	268	0
5 added	80	0	179	0	316	0
6 added	82	0	211	0	342	0

Table 5.3: Experiment 3 results: number of incorrect relations in the explanations

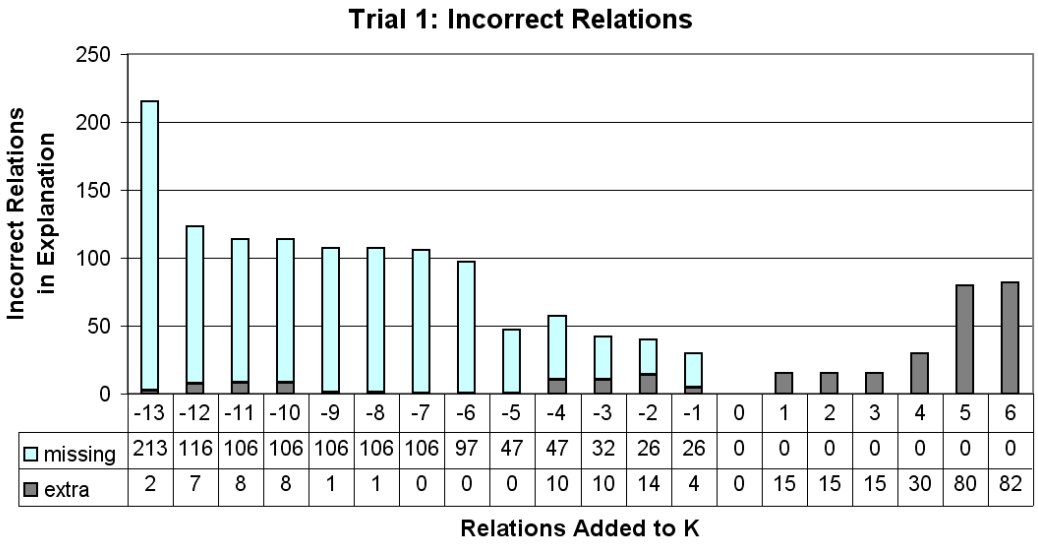


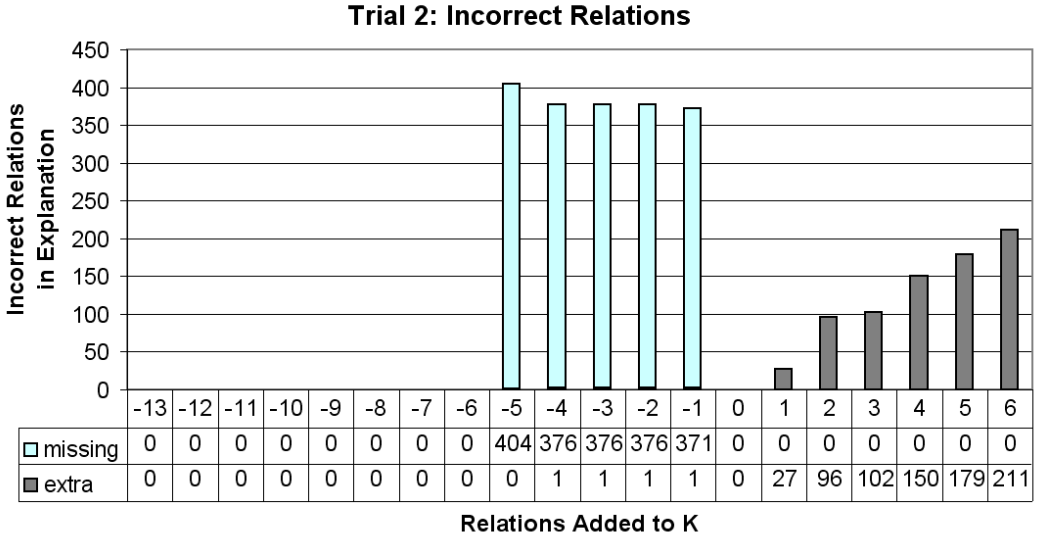
Figure 5.5: Trial 1 results

As seen in the results in Figure 5.5, as relations in K are added, those relations erroneously appear in the explanation, and thus the explanation becomes progressively inaccurate and overly complex. The degree of inaccuracy is dependent on the type of erroneous relation added, specifically, on how often that relation occurs in the explanation. For example, when the first relation was implanted, 15 extra relation was seen in the explanation because the added relation (i.e., A:flyToTarget → I:removeCommitment) occurred 15 times within the 200 timesteps and was referenced in the explanation. However, when the second relation (i.e., B:servicedTarget → M:commitments) was implanted, no additional extra relations were found in the explanation since the 'commitments' message is not referenced in the explanation, which is because the message is a broadcasted message and does not influence the agent's own activities. Likewise for the third relation (i.e., B:other_servicedTarget

→ M:servicedTarget), where the 'servicedTarget' message is also a broadcasted message that has no influence on the observation being explained.

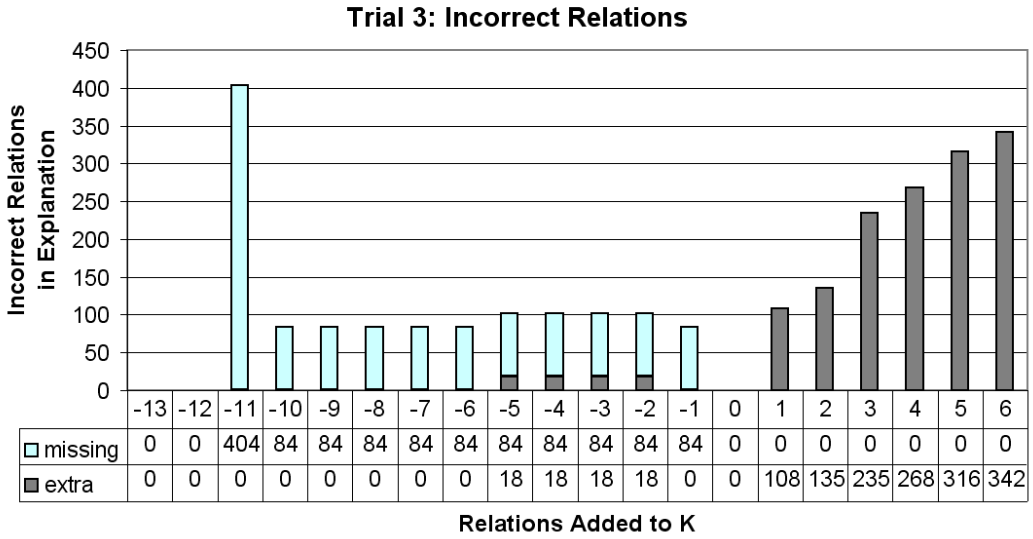
Figure 5.5 also shows that as relations in K are removed, more relations in the generated explanation become missing, and thus the explanation becomes progressively incomplete until all the correct relations have disappeared. It can be generalized from the charts that as the background knowledge K becomes less representative of the actual behavior, the explanations become less complete and less accurate. This is not surprising, but there are some interesting discoveries from the experiment worth noting.

The first discover is that, in some cases, the number of incorrect relations remains constant even after a relation is removed. For example in Trial 1, there is no difference between the 8^{th} and 9^{th} relation that was removed (i.e., both -8



	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
missing	0	0	0	0	0	0	0	0	404	376	376	376	371	0	0	0	0	0	0	0
extra	0	0	0	0	0	0	0	0	0	1	1	1	1	0	27	96	102	150	179	211

Figure 5.6: Trial 2 results



	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
missing	0	0	404	84	84	84	84	84	84	84	84	84	84	0	0	0	0	0	0	0
extra	0	0	0	0	0	0	0	0	18	18	18	18	0	0	108	135	235	268	316	342

Figure 5.7: Trial 3 results

and -9 resulted in explanations with 1 extra and 106 missing relations, as seen in Figure 5.5). The explanation remained the same despite removing Relation 14 (9^{th} relation removed), which states that a 'servicedTarget' message is caused by the 'servicedTarget' belief concerning the same target. The reason there is no change in the explanation is because the 'servicedTarget' message has already been disconnected from the explanation by removing Relation 7 (the 6^{th} to be removed).

In other words, looking at the explanation as a tree representation, all branches of the explanation that involve 'servicedTarget' messages has already been disconnected from the explanation. Hence, removing subbranches of the disconnected branch does not affect the explanation. This occurs due to the removal sequence of the relations and can also be observed in Trial 2 and Trial 3. This suggests that background knowledge K can be very fragile in that a single missing relation

can make an explanation very incomplete. For example, in Trial 2, 371 of the 404 relations were missing in the explanation after the first relation was removed.

The second interesting facet from the experiments is the increase in extra relations in the explanation, as seen in Trial 1 and Trial 3 between -1 and -2 (the 1^{st} and 2^{nd} to be removed), despite relations being removed. When a relation is removed from K, the observation that would otherwise have been connected is available for another relation to be formed by other relation-creating rules representing K. For example in Trial 1, when the second relation (Relation 1, which says that the 'addCommitment' intention is influenced by the last 'addCommitment' intention) is removed from K, all 'addCommitment' intentions will no longer have a relation coming from the last 'addCommitment' intention. Extra relations are then formed by Relation 2 (the 5^{th} relation to be removed), which states that any intention without an incoming relation is influenced by the last 'removeCommitment' intention. Once Relation 2 has been removed from K, the number of extra relations goes down to 0, as seen in Trial 1 in Table 5.3.

Recall that the depth of an explanation continues until an initial or exogenous observation is found. With relations removed from K, the explanation tree is incorrectly stunted at certain branches. There are two situations at those stunted branches: (1) an observation that should have an incoming relation but does not and (2) an observation that has an incorrect relation to another observation. In the first situation, if the observation is not an initial or exogenous observation, then a relation is needed that connects from the appropriate causal observation to the unconnected observation. Such a relation is suggested by the Tracer Tool. In the second situation, the user must notice that the relation is incorrect (that one observation does not directly influence the other observation) and create a correct relation in K.

The results from this experiment show that if the background knowledge is not representative of the implementation, then the explanations will not be accurate. Thus, as the hypothesis states, to generate accurate explanations of observations, the background knowledge must accurately represent the implementation. The experimental results also supports the premise that the *relations* among the observations are central to comprehension, which is why the focus of this research is to automate the creation and verification of these relations.

5.4 Contributions from RQ2

Contributions from RQ2 include (1) an application of ideas from abductive reasoning to the software comprehension of agents, (2) an explanation generator that uses background knowledge to explain why an agent performed some unexpected behavior, and (3) an investigation of how much and what type of background knowledge errors can affect explanation accuracy. Abductive reasoning has been applied to numerous application domains, including diagnosis [Console et al., 1991], classification [Mooney, 1997], and natural-language processing [Stickel, 1989], in order to automate human reasoning. Software comprehension is another application domain where automated reasoning can be useful, especially as agent software becomes more sophisticated and unmanageable. The conceptual framework developed by abductive reasoning researchers has been leveraged in this research to separate the background knowledge from the automated reasoning. In addition to describing what is happening in the system, the Tracer Tool generates explanations as evidence of software comprehension and allows the user to analyze reasons for agent behavior, thereby facilitating software maintenance tasks and the adoption of agent technology.

Chapter 6

SUMMARY

The intent of this research is to study, observe, and explain agent behavior within the system as a whole and provide feedback to the agent system designer and developer. In order to improve the agent system, to debug the implementation, or to determine whether to adopt an agent-based software solution, a comprehension of the agent software needs to be created. From this objective, software comprehension for agent systems was motivated. Comprehending the behavior of potentially sophisticated agents within a dynamic, uncertain environment is critical to the software development and maintenance process, as well as to the advancement of agent research. The sophistication of agent capabilities and the complexities of agent interaction demand an automated solution to comprehending agent software. In response to a need for comprehending software agent systems, this dissertation presents *agent software comprehension* (ASC) as a new challenging subfield of agent research. This research investigates an ASC framework that enables the automation of currently manual comprehension tasks, such as creating interpretations from observations of agent behavior and explaining anomalous observations in the implemented system. The result of automating these tasks is an integrated tool called the Tracer Tool, which is aimed at a wide range of users, from designers to end-users. By employing familiar abstract terminology (i.e., agent concepts) and generating explanations that use high and low abstraction levels (i.e., observations

that reference specific run-time data), the Tracer Tool bridges the language gap among the wide range of users.

6.1 Challenges

The following enumerates the challenges and difficulties that must be faced when a user (designer, developer, or end-user) attempts to comprehend an agent system:

ambiguous terminology The terminology used to describe agent-based system still remains inconsistent and ambiguous across different systems and across different users. This research prescribes a method to concretize agent concepts to actual data structures, thus agent concepts are linked to specific implemented concepts in the particular agent system.

agent sophistication Individual agents, when observed and studied, often reveal complex, proactive, adaptive, and social behaviors. The complexity of planning algorithms, goal interactions, and resource constraints can hinder the process of forming an understanding of the system. This research seeks to create an abstracted and integrated view of the agents, thereby facilitating the comprehension of the overall agent system.

agent interaction Due to the multitude of exogenous environmental events, agent interaction (contracts and negotiations), and other agent actions that an agent must consider when making a decision, it can be difficult to envision all of the connections within the system. This research provides a visualization of the interactions (e.g., causal and temporal relations) within and among the agents.

nondeterminism Agent-based systems are suited to nondeterministic environments, where environmental events can occur unexpectedly. Agents may or may not consider such events in their decision-making process. This research captures nondeterministic events and relates them to agent decision-making so that they are recognized as factors that influence an agent's decisions.

large source code size Users must deal with the usually large source code size of the agent system implementation. This research abstracts from the source code details to build a comprehensive picture of the system while maintaining a mapping to the source code. Since each observation contains run-time data (e.g., stacktrace) about where and in what context the observation occurred, sections of code can be identified for each observation.

data overload Human users must organize, make connections, and reason about a copious amount of observation data. This research reduces the amount of manual effort involved by automating observation data processing.

capturing human comprehension Representing what the human user comprehends (i.e., the background knowledge) is difficult, but it is necessary in order to provide informative feedback to the user. This research captures agent behaviors according to the user comprehends, as well as suggesting other behaviors that the user may not have recognized, thus, helping the user to build up his/her background knowledge.

verifying the implementation The user (particularly a designer or developer) expects certain agent behaviors as specified in a design and must manually verify those behaviors in the actual implemented system. This

research automatically verifies the user's expected behavior against the implementation's actual behavior.

These challenges pervade through all fields within the agent research community. For example, a recent call for papers from SASEMAS (an international workshop on Safety and Security in Multiagent Systems http://sasemas.org) reads:

> Moreover, agents often integrate such activities as deliberately planning to achieve their goals, dynamically reacting to obstacles and opportunities, communicating with other agents to share information and coordinate actions, and learning from and/or adapting to their environments. Because agents are often situated in dynamic environments, these activities are often time-sensitive. These aspects of agents make the process of developing, verifying, and validating safe and secure multiagent systems more difficult than for conventional software systems. Hence, new and different techniques and perspectives are required to assist with the development and deployment of such systems.
>
> [SASEMAS 2005 call for papers]

6.2 Research Problem Statement

The research problem is *comprehending agent behavior in a situated agent system*, or more specifically, *finding explanations for agent actions that occur during the execution of the implemented agent system*. The hypothesis for this problem considers the importance of background knowledge accuracy in generating accurate explanations – *accurate explanations of actual agent behavior can be generated if models of expected agent behavior being used as background knowledge are representative of the implementation's behavior*. The overall approach for the hypothesis is (1) to aid the user as much as possible in building an accurate and up-to-date background knowledge and (2) to use the background knowledge in generating the

explanation. In order to produce accurate explanations, the explanation generator requires background knowledge that is representative of what it is explaining – specifically, consistency between the background knowledge and the implementation's behavior must be checked. The following research questions decompose the approach into manageable topics concerning the representation, consistency-checking, and use of the background knowledge for accurate explanation generation.

Research Question 1 : *How can background knowledge be created such that it accurately represents the implementation's behavior?*

Research Question 2 : *How can background knowledge be used to explain observed agent behavior?*

The approach taken by this research is to leverage background knowledge to (1) make anomalous behavior easily detectable as shown by the visualization of the Tracer Tool and (2) investigate observations by generating explanations in terms of familiar agent concepts.

6.3 Research Question 1

In answering the Research Question (RQ1) "*How can background knowledge be created such that it accurately represents the implementation's behavior?*", the background knowledge must be formally represented and the representation must be flexible enough to embody the user's comprehension without being overly complex. Due to several existing tools that can capture the structural aspects of software (e.g., Rigi [Agrawal et al., 1998], Gen++ [Devanbu, 1992], and DESIRE [Biggerstaff et al., 1994]), this dissertation focuses on analyzing behavioral aspects of agent systems and leverages the abstraction of agent concepts to automate com-

prehension activities. Hence, the background knowledge contains behavioral aspects of the agents where the primitive elements of the representation are agent concepts, such as beliefs, goals, intentions, actions, events, and messages. Unlike detailed data structures, these agent concepts are familiar to designers, developers, and end-users. Also important are the relations among these agent concepts, which will be defined by the user and/or suggested by the Tracer Tool. By employing agent concepts in the background knowledge, the generated explanations will be in terms of agent concepts, which is desired to facilitate comprehension. As a natural result of these requirements, a semantic network (specifically, a causal network) was chosen as the representation for the background knowledge.

Given that the initial background knowledge is the user's comprehension of the agents' behavior represented using semantic networks, the tasks involved in answering RQ1 are (1) to compare the background knowledge with the implementation's behavior and (2) to help update the background knowledge for accuracy. The approach taken in this dissertation for RQ1 is to extend the ideas of empirical analysis from reverse engineering and abstracted modeling from model-checking.

First, instead of analyzing the source code of the implementation, which would be programming language-specific and produce too much data for the user to grasp, the research approach is to record observations (at the agent concept level) of the executing agent system and relate those observations using the background knowledge. The result of processing and relating observations is an *interpretation* of actual agent behavior, visualized as a semantic network [Lam and Barber, 2005*a*]. Second, if (1) an observation is not connected to other observations, (2) a defined relation in the background knowledge is not used, or (3) the resulting relations do not make sense to the user, then the background knowledge does not accurately represent the implementation. In such cases, the Tracer Tool allows the

user to quickly assess the problem using the GUI to the background knowledge and the interpretation, or the Tracer Tool will suggest possible relations that can be added to the background knowledge.

Experiment 1 (EX1) showed that the algorithm used to suggest possible relations can also be used to build up an initially empty or sparse background knowledge [Lam and Barber, 2005*b*]. Experiment 2 (EX2) then demonstrated the types of errors that can be detected using this research approach. In summary, the detected errors can be traced to the implementation or to the user's comprehension of the implementation. In either case, the user is aware of the inaccuracy between the background knowledge and implementation and can make updates appropriately, which is the objective of RQ1. With this approach, the background knowledge and implementation is kept in synchrony even as the implementation evolves, and the background knowledge can grow as new scenarios are realized.

6.4 Research Question 2

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, Research Question 2 (RQ2) asks, "*How can background knowledge be used to explain observed agent behavior?*" Given the accurate background knowledge from RQ1 and a set of recorded observations from the system's execution in some scenario, how can the Tracer Tool generate an explanation for any particular observation (i.e., reasons for why an observation occurred). Since the background knowledge is in terms of agent concepts and their relations, the resulting explanation will consist of observations at the agent concept level and relations between those observations. The explanations can be used by designers, developers, and end-users to elucidate, debug, and build trust in the agent system's behavior. The explanations can be used

to assist the designer in (1) verifying that the agents behave as predicted by agent models (and not by coincidence) and (2) understanding unexpected agent behavior.

The approach taken in this research borrows much from abductive reasoning, which has often been used to generate explanations from observations and background knowledge. A focus of abductive reasoning is to make assumptions (e.g., the occurrence of an observation) in order to produce a causal sequence of observations, linking one observation to another for a more complete picture of why certain observations occur. Example applications of abductive reasoning include medical diagnosis [Console et al., 1991], natural-language processing [Stickel, 1989], and plan recognition [Appelt and Pollack, 1992]. In this dissertation, all relevant observations are recorded so no assumptions need to be made. This fact facilitates explanation generation and avoids the NP-hard complexity of abductive reasoning. In others applications of abductive reasoning, the background knowledge may need to be updated to maintain representative accuracy about the domain [Pagnucco, 1996]. Since the answer to RQ1 offers the Tracing Method to build and maintain accurate background knowledge, RQ2 does not directly address the issue of background knowledge inaccuracies. Often, abductive reasoning assumes a complete background knowledge from which to hypothesize about assumptions. RQ1 has also addressed this issue by suggesting updates to the background knowledge as new behaviors appear. Note that the completeness of the background knowledge depends on the set of scenarios selected in which to test the agent system (see Assumptions and Limitations section below).

An explanation is generated by mapping an observation to the corresponding agent concept in the background knowledge, then following the defined relations in the background knowledge to other observations that influenced that observation. This process can recurse into any depth until the user is satisfied with

the explanation and stops when an initial or exogenous observation is found (e.g., the initialize agent beliefs or nondeterministic environmental events, respectively). The resulting explanation is essentially a subgraph (specifically, a tree) of the interpretation produced in RQ1, where the root of the tree is the observation being explained.

Experiment 3 (EX3) addresses the hypothesis and demonstrates that as the background knowledge becomes less representative of the actual behavior, the explanations become more inaccurate. The experimental results show that the generated explanation is very sensitive to changes in the background knowledge. When incorrectly removing or adding a relation to the background knowledge, the number of inaccuracies in the explanation can drastically increases, depending on the frequency that the relation occurs. Hence, Experiment 3 affirms that the effort spent in building an accurate background knowledge (Research Question 1) is a crucial and worthwhile investment.

6.5 Contributions/Conclusion

Software comprehension is essential for developing, maintaining, and redesigning complex software, such as agent-based systems. This research strives to remedy the drawbacks and limitations of existing techniques, specifically 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" [Lange et al., 2001]. Though model-checking offers conciseness and abstraction in its representation and exhaustive search in its property verification, the representativeness of its model with re-

spect to the actual implementation is difficult to prove and maintain as the implementation evolves. The verified properties may not apply to the implementation if the model does not accurately represent the implementation or vice versa.

This research contributes to the fields of software engineering and artificial intelligence and, in particular, advances the state-of-the-art in Agent-Oriented Software Engineering (AOSE). In an effort to address the limitations of traditional software comprehension approaches (i.e., reverse engineering), this research offers an approach for comprehending agent systems by modeling the user's understanding of the system and by generating explanations of agent behavior. Traditional software comprehension approaches analyze the detailed structure and program flow of the source code. For agent software, comprehension should involve analysis of the implementation in terms of high-level agent concepts – the same concepts used in agent-oriented design and analysis; the same concepts used to present the agent system to the end-user. This research contributes a method to analyze and comprehend agent software via knowledge modeling, automated interpretation, and explanation generation. The explanations operate over agent concepts and provide relational information among agent concepts (e.g., causal and temporal information) to support software maintenance and testing activities so that the desired behaviors can be attained.

This dissertation describes three contributions that extend ideas from existing work to assist the user in comprehending agent software. First, a high-level representation (called the background knowledge) was defined that explicitly describes the user's growing knowledge of the software's behavior in terms of agent concepts. This 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 the background knowledge (or 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 background knowledge, 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 leverages the semantics of agent concepts as abstractions and provides automated reasoning about the resulting abstracted representations.

This research facilitates the study of software agent systems by automating as much of the comprehension process as possible, such as organizing the logged data, creating interpretations of logged data, verifying the background knowledge, and generating explanations. By combining techniques from model-checking and reverse engineering, a model of the user's comprehension is maintained and verified such that the model accurately represents the actual system. Given an accurate representation of the user's comprehension and of the implementation, an accurate explanation of why the agent behaved in a certain manner can be automatically generated.

The result of this research is a high-level, more scalable, practical, semi-automated solution for agent software comprehension. Since comprehension is performed at the system-level using high-level agent concepts that are familiar to many stakeholders along the software engineering cycle, all activities in this dissertation approach 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 because (1) it can be applied to any programming language and

(2) the learning curve is low, unlike many reverse engineering and model-checking tools.

6.6 Future work

The contributions of this dissertation offers a foundation and framework for future work in agent software comprehension. Future work to extend and/or build on this research includes the following:

1. Create a translator to generate background knowledge from the design specification. As a result, the design and the background knowledge are equivalent (for the most part). By using this background knowledge in the Tracer Tool, the implementation is being checked against the design, and thus one can check whether the implementation was faithful to the design.
2. Develop a behavior-pattern recognition algorithm to automatically detect anomalous behavior. Using machine learning or data-mining techniques, patterns of software behavior can be discovered and collected into a behavior library. If a behavior does not match any behavioral pattern, then an anomaly probably exists.
3. Explore other types of interpretations that can be more easily understood by the human user. Graphical representations are essential for effectively communicating what the agents are doing to a wide audience. Different visualization can be tested to determine how well human subjects comprehend agent activities. Furthermore, an interpretation can readily become the main visualization for the agent system. In particular, a domain-independent interpretation can remedy the need to create a GUI for every application domain.

4. Integrate the interpretation with agent decision-making for human-in-the-loop activities. For example, if an agent has several equally rewarding choices based on its belief about the world, then the agent can present those choices for the human user (who may have a better understanding of the current situation or future states) to decide the best choice or to provide more information so the agent can reassess its choices.

5. Apply the Tracing Method to systems where the implemented agent's behavior dramatically changes. In particular, agents that mutate or learn different ways to achieve their goals (using neural networks) are challenging because the expected agent behavior is continually changing. It is unclear how to address the issue of a background knowledge that is continually being revised and does not stabilize.

Appendices

Appendix A

Agent Design Methods

AOSE methodologies guide the generation of agent design models representing and consisting of agent concepts. With the current state of agent technology, designing agents is a time-consuming process, requiring much research and deliberation [Wooldridge and Jennings, 1998]. The main purpose of AOSE is to create methodologies and tools that facilitate the development and maintenance of agent-based software, similar to the purpose of object-oriented software engineering (OOSE) [Tveit, 2001]. AOSE must deal with the same issues as OOSE, such as ways to build flexible, scalable, high quality software. As a result, AOSE methodologies and techniques adopt or extend OO methodologies and techniques to be applicable to agents. Due to a push for the design and deployment of agent systems, current AOSE research has not focused on techniques for the testing and maintenance phases of the software engineering life-cycle. The following describes design methods that create agent design models, which are possible sources for deriving the background knowledge for the abductive reasoning used in this research to explain agent behavior.

There are a number of available agent-building tools (e.g., Concordia [Wong et al., 1997]) and agent-based development environments (e.g., JADE [JADE, 2000]). Eiter and Mascardi compares five agent development environments to help agent developers choose the best one that suits the features and requirements of their application [Eiter and Mascardi, 2001]. A brief description of two

popular analysis and design methodologies (Gaia and MaSE) are reviewed to give an idea of the types of agent models that exist, followed by other agent research (AUML, design patterns, component reuse) that extends techniques and tools from object-oriented software engineering.

The Gaia methodology offers a top-down approach for agent-oriented analysis and design [Wooldridge et al., 2000]. Gaia supports derivation of agent structure and agent organizational structure. The analysis phase identifies roles (and their associated domain-specific responsibilities, permissions, activities, and protocols) and interactions. The design phase maps roles to agent types, determines services model to fulfill roles, and creates the acquaintance model for agent communication. The Multiagent Systems Engineering (MaSE) Methodology by DeLoach et al. aims to lead the designer from the initial system specification to the implemented MAS, including support for automatic code generation [DeLoach et al., 2001]. This general methodology has seven phases (capturing goals, applying use cases, refining roles, creating agent classes, constructing conversations, assembling agent classes, and system design) and generates models that utilize agent concepts, such as intentions, negotiation, and roles.

Odell and Parunak propose an extension to Universal Modeling Language (UML) called Agent UML (AUML) [Odell and Parunak, 2000], which includes richer role specifications and the capability to model mobile agents. For example, a new protocol diagram combining UML interaction diagrams and state diagrams to model agent roles during agent interaction was developed. Aridor and Lange suggest a classification scheme for design patterns (reoccurring patterns of programming code or software architecture) in order to reuse agent designs and reduce development effort of mobile agents [Aridor and Lange, 1998]. Erol et al. suggest a

three-tiered (interactions, local information and expertise, and information-content) architecture to design agents from reusable components [Erol et al., 2000].

Bibliography

Agrawal, A., Du, M., McCollum, C., Syst, T., Wong, K., Yu, P. and Mller, H. [1998], Rigi - An End-User Programmable Tool for Identifying Reusable Components, *in* 5th International Conference on Software Reuse, Victoria, British Columbia. 3, 30, 147

Aliseda-Llera, A. [1997], Seeking Explanations: Abduction in Logic, Philosophy of Science, and Artificial Intelligence, PhD thesis, Stanford University. 41, 42

Appelt, D. E. and Pollack, M. E. [1992], Weighted Abduction for Plan Ascription, *in* User Modeling and User-Adapted Interaction. 17, 25, 40, 150

Aridor, Y. and Lange, D. B. [1998], Agents Design Patterns: Elements of Agents Application Design, *in* K. P. Sycara and M. J. Wooldridge, eds, 2nd International Conference on Autonomous Agents, ACM Press, Minneapolis/St. Paul, MN, pp. 108–115. 158

Barber, K. S. and Lam, D. [2003*a*], Enabling Abductive Reasoning for Agent Software Comprehension, *in* 18th International Joint Conference on Artificial Intelligence Workshop on Agents and Automated Reasoning, Acapulco, Mexico, pp. 7–13. 65, 75, 83

Barber, K. S. and Lam, D. [2003*b*], Motivating Abductive Explanation for Multi-Agent System Comprehension, *in* 7th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, FL, pp. 484–489. 70

Biggerstaff, T. J., Mitbander, B. G. and Webster, D. [1994], Program Understanding and the Concept Assignment Problem, *Communications of the ACM* **37**(5), 72–83. 31, 74, 147

Boehm, B. W. [1981], *Software Engineering Economics*, Prentice Hall. 37

Bosse, T., Lam, D. N. and Barber, K. S. [2006], Automated Analysis and Verification of Agent Behavior, Technical Report TR2006-UT-LIPS-001, The University of Texas at Austin. 121

Bruening, D., Devabhaktuni, S. and Amarasinghe, S. [2000], Softspec: Software-based Speculative Parallelism, *in* 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, ACM Press, Montery, California. 5, 31

Bylander, T., Allemang, D., Tanner, M. C. and Josephson, J. R. [1991], The Computational Complexity of Abduction, *Artificial Intelligence* **49**, 25–60. 19, 25, 44, 46

Charniak, E. and McDermott, D. [1985], *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Inc., Reading, MA. 45

Chikofsky, E. J. and James H. Cross, I. [1990], Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software* **7**(1), 13–17. 3, 30

CoABS [2003], Control of Agent-Based Systems.
URL: *http://coabs.globalinfotek.com/* 81

Console, L., Portinale, L. and Dupre, D. T. [1991], Focusing Abductive Diagnosis, *AI Communications* **4**, 88–97. 17, 25, 40, 142, 150

CORBA [1999], Common Object Request Broker Architecture.
URL: *http://www.corba.org/* 81

De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. and Yang, J., eds [2002], *Software Visualization, State-of-the-Art Survey*, Lecture Notes in Computer Science 2269, Springer Verlag. 33

DeLoach, S. A., Wood, M. F. and Sparkman, C. H. [2001], Multiagent Systems Engineering, *International Journal of Software Engineering and Knowledge Engineering* **11**(3), 231–258. 24, 25, 158

Devanbu, P. T. [1992], GENOA- A Customizable, Language- and Front-End Independent Code Analyzer, *in* Fourteenth International Conference on Software Engineering, Melbourne, Australia, pp. 307–319. 31, 74, 147

Doyle, D., Tsymbal, A. and Cunningham, P. [2003], A Review of Explanation and Explanation in Case-based Reasoning, Technical Report TCD-CS-2003-41, Trinity College. 50

Dupre, D. T. and Rossotto, M. [1995], The Different Roles of Abstraction in Abductive Reasoning, *in* Lecture Notes in Artificial Intelligence, Vol. 992, pp. 211–216. 47

Edmonds, B. and Bryson, J. [2004], The Insufficiency of Formal Design Methods, *3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems* pp. 938–946. 34, 47, 121

Eiter, T. and Mascardi, V. [2001], Comparing Environments for Developing Software Agents, Technical Report Technical report INFSYS RR-1843-01-02, Vienna University of Technology. 157

Erol, K., Lang, J. and Levy, R. [2000], Designing Agents from Reusable Components, *in* 4th International Conference on Autonomous Agents, pp. 76–77. 159

Findler, N. [1979], *Associative Networks: The Representation and Use of Knowledge by Computers*, Academic Press, New York. 45

Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Meller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M. and Wong, K. [1997], The Software Bookshelf, *IBM Systems Journal* **36**(4), 564–593. 3, 30

Fjeldstad, R. K. and Hamlen, W. T. [1983], Application Program Maintenance Study: Report to Our Respondents, *in* GUIDE 48, Philadelphia, PA. 37

Garcia-Molina, H., Germano, F. and Kohler, W. [1984], Debugging a Distributed Computer System, *IEEE Transactions on Software-Engineering* **10**(2), 210–219. 48

Gasser, L., Braganza, C. and Herman, N. [1987], MACE: A Flexible Testbed for Distributed AI Research, *in* M. N. Huhns, ed., Distributed Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, pp. 119–152. 1, 3

Hempel, C. G. [1965], *Aspects of Scientific Explanation*, The Free Press, New York, NY. 45

Hindsight [2004].
URL: *http://www.testersedge.com/hindsight.htm* 4, 33

Holzmann, G. J. [1997], The Model Checker Spin, *IEEE Transactions on Software Engineering* **23**(5), 279–295. 33

JADE [2000], Java Agent Development Framework.
URL: *http://sharon.cselt.it/projects/jade/* 24, 25, 157

Java Logging APIs [2002].
URL: *http://java.sun.com/j2se/1.4/docs/guide/util/logging* 81

Jennings, N. R. [1999], 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), Vol. 1647, pp. 1–7. 69

Jennings, N. R. [2000], On Agent-based Software Engineering, *Artificial Intelligence* **117**, 277–296. 15, 36

Jennings, N. R. and Wooldrige, M. [1998], Applications of Intelligent Agents, *in* N. R. Jennings and M. J. Wooldrige, eds, Agent Technology: Foundations, Applications, and Markets, Springer-Verlag, Berlin, pp. 3–28. 22

Jerding, D. and Rugaber, S. [1998], Extraction of Architectural Connections from Event Traces, *in* ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM Press, Montreal, Canada. 4, 32, 33

Jonker, C. M. and Treur, J. [2002], Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness, *Intl. Journal of Cooperative Information Systems* **11**, 51–92. 121

Josephson, J. R. and Josephson, S. G., eds [1994], *Abductive Inference: Computation, Philosophy, Technology*, Cambridge University Press, New York. 40

Kalech, M. and Kaminka, G. A. [2003], On the Design of Social Diagnosis Algorithms for Multi-Agent Teams, *in* 18th Intl. Joint Conference on Artificial Intelligence, pp. 370–375. 47

Konolige, K. [1991], Abduction versus Closure in Causal Theories, *Artificial Intelligence* **52**, 255–272. 44

Koskimies, K., Mnnist, T., Syst, T. and Tuomi, J. [1998], Automated Support for Modeling OO Software, *IEEE Software* **15**(1), 87–94. 4, 32

Kullbach, B. and Winter, A. [1999], Querying as an Enabling Technology in Software Reengineering, *in* P. Nesi and C. Verhoef, eds, 3rd European Conference on Softward Maintenance and Reengineering, IEEE Computer Society, Los Alamitos, pp. 42–50. 5, 31

Lam, D. N. and Barber, K. S. [2004], Debugging Agent Behavior in an Implemented Agent System, *in* 2nd International Workshop on Programming Multi-Agent Systems at the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY, pp. 45–56. 128

Lam, D. N. and Barber, K. S. [2005*a*], Automated Interpretation of Agent Behavior, *in* Workshop for Agent-Oriented Information Systems, Utrecht, Netherlands, pp. 74–81. 148

Lam, D. N. and Barber, K. S. [2005*b*], Comprehending Agent Software, *in* 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems, Utrecht, Netherlands. 69, 149

Lange, C., Winter, A. and Sneed, H. M. [2001], Comparing Graph-Based Program Comprehension Tools to Relational Database-Based Tools, *in* Ninth International Workshop on Program Comprehension, IEEE Computer Society, Toronto, Canada, p. 209. 32, 51, 151

Leake, D. B. [1993], Focusing Construction and Selection of Abductive Hypotheses, *in* International Joint Conference on Artificial Intelligence, Chambery, France, pp. 24–29. 17, 25, 40, 47

Liedekerke, M. and Avouris, N. [1995], Debugging Multi-Agent Systems, *Information and Software Technology* **37**(2), 103–112. 15, 48

Lucca, G. D., Fasolino, A. and Carlini, U. [2000], Recovering Use Case Models from Object-Oriented Code: A Thread-based Approach, *in* 7th Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 108–117. 33

Mayrhauser, A. v. and Lang, S. [1999], On the Role of Static Analysis during Software Maintenance, *in* 7th International Conference on Program Comprehension, IEEE Computer Society, Pittsburgh, PA, pp. 170–177. 4, 33

McCarthy, J. [1980], Circumscription- A Form of Non-monotonic Reasoning, *Artificial Intelligence* **13**(1-2), 27–39. 46

Memmel, M. and Roth-Berghofer, T. [2004], Explanation and e-Learning: A 1st Pass, *in* K.-P. Fahnrich, K. P. Jantke and W. S. Wittig, eds, Von e-Learning bis e-Payment. Das Internet als sicherer Marktplatz, Akademische Verlagsgesellschaft Aka, Berlin, pp. 255–263. 48, 49

Minker, J. [1993], An Overview of Nonmonotonic Reasoning and Logic Programming, *Journal of Logic Programming* **17**(2-4), 95–126. 24

Mooney, R. J. [1997], Integrating Abduction and Induction in Machine Learning, *in* Working Notes of the IJCAI-97 Workshop on Abduction and Induction in AI, Nagoya, Japan, pp. 37–42. 17, 25, 40, 142

Ndumu, D., Nwana, H., Lee, L. and Collis, J. [1999], Visualising and Debugging Distributed Multi-Agent Systems, *in* 3rd Annual Conference on Autonomous Agents, pp. 326–333. 16, 48

Odell, J. and Parunak, H. V. D. [2000], Extending UML for Agents, *in* Agent-Oriented Information Systems Workshop at the 17th National conference on Aritificial Intelligence. 158

Pagnucco, M. [1996], The Role of Abductive Reasoning within the Process of Belief Revision, Ph.d., University of Sydney. 42, 150

Peirce, C. S., ed. [1931-1958], *Collected Papers*, Harvard University Press, Cambridge, MA. 41, 42

Peng, Y. and Reggia, J. A. [1990], *Abductive Inference Models for Diagnostic Problem-Solving*, Symbolic Computation, Springer-Verlag. 46

Poole, D. [1993], Probabilistic Horn Abduction and Bayesian Networks, *Artificial Intelligence* **64**(1), 81–129. 45

Quillian, M. [1968], Semantic Memory, *in* M. Minsky, ed., Semantic Information Processing, MIT Press, Cambridge, Mass., pp. 216–270. 45

Robby, Dwyer, M. B. and Hatcliff, J. [2003], Bogor: An Extensible and Highly-Modular Model Checking Framework, *in* 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, Helsinki, Finland, pp. 267–276. 33

Ross, R., Collier, R. and O'Hare, G. M. [2004], AF-APL – Bridging Principles& Practice in Agent Oriented Languages, *in* 2nd International Workshop on Programming Multi-Agent Systems at the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY, pp. 21–33. 69

Rugaber, S. [1995], Program Comprehension, *in* Encyclopedia of Computer Science and Technology, Vol. 35(20), Marcel Dekker, Inc., New York, pp. 341–368. 37

Schauer, R. and Keller, R. K. [1998], Pattern Visualization for Software Comprehension, *in* 6th International Workshop on Program Comprehension, Ischia, Italy, pp. 4–12. 32

Sim, S. E. and Storey, M.-A. [1999], A Structured Demonstration of Program Comprehension Tools, *in* Seventh Working Conference on Reverse Engineering, IEEE Computer Society, Toronto, Ontario, Canada, p. 184. 4, 32

Sowa, J. F. [1984], *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA. 61

Stasko, J. T. [1990], TANGO: A Framework and System for Algorithm Animation, *IEEE Computer* **23**(9), 27–39. 4, 32

Stickel, M. E. [1989], Rationale and Methods for Abductive Reasoning in Natural Language Interpretation, *in* International Scientific Symposium: Natural Language and Logic. 17, 24, 40, 142, 150

Stroulia, E. and Syst, T. [2002], Dynamic Analysis for Reverse Engineering and Program Understanding, *ACM SIGAPP Applied Computing Review* **10**(1), 8–17. 26

Suna, A. and Fallah-Seghrouchni, A. E. [2004], A Mobile Agents Platform: Architecture, Mobility and Security Elements, *in* 2nd International Workshop on Programming Multi-Agent Systems at the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY, pp. 57–66. 69

Swartout, W. R. and Moore, J. D. [1993], Explanation in 2nd Generation Expert Systems, *in* J. David, J. Krivine and R. Simmons, eds, 2nd Generation Expert Systems, Springer Verlag, Berlin, pp. 543–585. 48

Swartout, W. R., Paris, C. L. and Moore, J. D. [1991], Design for Explainable Expert Systems, *IEEE Expert* **6**(3), 58–64. 49

Systa, T. [2000], Understanding the Behavior of Java Programs, *in* Working Conference on Reverse Engineering, pp. 214–223. 29

TASK [2001], DARPA Taskable Agent Software Kit.
URL: *http://www.darpa.mil/ipto/programs/task/* 93

Thagard, P. and Shelley, C. [1997], Abductive Reasoning: logic, visual thinking, and coherence, *in* M. D. Chiara, K. Doets, D. Mundici and J. v. Benthem, eds, Logic and Scientific Method, Kluwer Academic Publisher, p. 413427. 44, 46

Tveit, A. [2001], A Survey of Agent-Oriented Software Engineering, *in* NTNU Computer Science Graduate Student Conference, Norwegian University of Science and Technology. 157

Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F. [2003], Model Checking Programs, *Automated Software Engineering Journal* **10**(2). 34

Wills, L. M. [1993], Flexible Control for Program Recognition, *in* Working Conference on Reverse Engineering, IEEE Computer Society Press, Baltimore, MD, pp. 134–143. 31

Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M. and Peet, B. [1997], Concordia: An Infrastructure for Collaborating Mobile Agents, *in* K. Rothermel

and R. Popescu-Zeletin, eds, Mobile Agents: Proceedings of the 1st International Workshop, MA '97, LNCS 1219, Springer, Berlin, Germany, pp. 86–97. 25, 157

Wooldridge, M. [2002], *An Introduction to MultiAgent Systems*, John Wiley and Sons, Chichester, England. 3

Wooldridge, M. J. and Jennings, N. R. [1998], Pitfalls of Agent-Oriented Development, *in* K. P. Sycara and M. J. Wooldridge, eds, 2nd International Conference on Autonomous Agents, ACM Press, Minneapolis/St. Paul, MN, pp. 385–391. 157

Wooldridge, M., Jennings, N. R. and Kinny, D. [2000], The Gaia Methodology for Agent-Oriented Analysis and Design, *Journal of Autonomous Agents and Multi-Agent Systems* **3**(3), 285–312. 24, 25, 158

Vita

Dũng Ngọc Lâm was born in Saigon, Vietnam on March 20, 1977, the son of Trung N. and Frances T. Ngũyên. At age 6, he arrived in the United States of America with his mother and began his education at several public elementary schools. After completing his work at Alfred Bonnabel High School, Metairie, Louisiana, in 1995, he entered Loyola University in New Orleans, Louisiana. He received the degree of Bachelor of Science in Physics from Loyola University in May 1999. The Phi Beta Kappa Honors Organization awarded his undergraduate thesis as the Outstanding Honors Thesis in Mathematics and Computer Science. In September 1999, he entered The Graduate School at The University of Texas at Austin. He received his Master's Degree in December 2001 and his Doctorate's Degree in December 2005 in Computer Engineering under the guidance and support of his graduate supervisor, Dr. K. Suzanne Barber.

Permanent address: 8603 Danville Dr.
Austin, Texas 78753

This dissertation was typeset by the author.