

Automated Analysis and Verification of Agent Behavior

Tibor Bosse

Vrije Universiteit Amsterdam
Department of Artificial Intelligence
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
tbosse@cs.vu.nl

Dung N. Lam, K. Suzanne Barber

The University of Texas at Austin
Laboratory for Intelligent
Processes and Systems
1 University Station C5000
Austin, TX 78712-0240

{dnlam, barber}@lips.utexas.edu

ABSTRACT

Comprehending and analyzing agent behavior is an arduous task due to complexities in agent systems and sophistication of agent behaviors, in addition to the common difficulties with any complex software system. This paper presents an integrated approach for the analysis and verification of behaviors of agent-based systems. The approach is a result of collaboration between the Tracer Tool and the TTL Checker, which together automate the analysis and verification of agents in an implemented agent system with the aim of aiding the user in redesigning, debugging, and maintaining the software system. The Tracer Tool ensures that the user's comprehension of the system behavior is accurate and provides explanations of anomalous behavior, which can be detected as a failed behavioral property by the TTL Checker. The integrated approach has been applied successfully in a case study in the domain of Unmanned Aerial Vehicles.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: *formal methods*; D.2.5 [Testing and Debugging]: *debugging aids, diagnostics, monitors, testing tools, tracing*.

General Terms

Documentation, Experimentation, Human Factors, Verification.

Keywords

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

1. INTRODUCTION

Software comprehension is critical in debugging and maintaining complex software systems, though it has lacked appropriate attention in agent software systems. Most existing tools aimed at comprehending agent-based software can be classified either as reverse engineering tools or as model checking tools. According

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'06, May 8-12, 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005...\$5.00.

to [2], *reverse engineering* is “the process of analyzing a subject system to create representations of the system at a higher level of abstraction”. Following this definition, a number of reverse engineering tools have been developed to assist the designer in analyzing and perusing the source code of a software system, e.g., [5]. In addition, various tools have been developed to perform *model checking* (i.e., to prove that the formal specification of a system satisfies certain behavioral properties) [3]. Nevertheless, despite the above tools to aid the user in understanding software behaviors, software comprehension has remained a time-consuming and mostly manual process. The obstacles faced by designers, developers, and end-users (who will be referred to simply as ‘users’) during software comprehension include (1) software complexity and sophistication, (2) the translation gap between low-level implementation structures and high-level design concepts, and (3) the large amount of time and effort demanded to gather a comprehensive picture of the software.

This paper demonstrates the Tracing Framework, which is supported by the Tracer Tool by [7] and the TTL Checker by [1], to reduce the effort in comprehending agent-based software systems. The approach to contend with the difficulties of software comprehension is to automate as many of the tasks involved in software comprehension as possible by presenting a visualization of what the agents are doing in terms of high-level agent concepts (e.g., beliefs, intentions, actions, and events) based on logs of system execution and comparing the visualization to the user's comprehension of the implemented agent system. The user's comprehension is explicitly modeled in order for the Tracer Tool to provide feedback as to the accuracy of the user's comprehension.

Additionally, the TTL Checker provides automated checking of agent or system behavioral properties against logs of system executions. This approach is similar to model-checking, except the input is a set of observations from the systems execution, rather than a formal model of the system. This is in accordance with Edmonds' emphasis on the need for empirical analysis of agent behavior, such as scenario-based analysis and field testing [4]. A user can discover behavioral anomalies from Tracer's graphical interface or be alerted by the TTL Checker. To aid the user in tracking down a problem, Tracer can generate explanations that reveal, for example, why an agent performed a certain action.

2. TRACER TOOL AND TTL CHECKER

Motivated by the difficulty of comprehending complex agent systems, the objective of the Tracing Framework [7] is to aid the

user in understanding what is happening in the agent system and why such things are happening in terms of high-level agent concepts that are familiar to software designers, developers, and end-users. The Tracing Framework is established by the **Tracer Tool**, which provides automation to some of the arduous, manual tasks involved for comprehension (see [7] for details):

Building Background Knowledge. In order to help the user build and maintain a comprehension of the system, the Tracer Tool requires a model of what the user understands. This model is called the background knowledge and is represented as a relational graph, where each node represents an agent concept and an edge represents a relation (e.g., causal or temporal relation) between two agent concepts. Given an empty or incomplete background knowledge, the Tracer Tool can offer suggestions about possible relations between agent concepts based on observations from the system's execution, thus building up the background knowledge.

Logging. Agent concepts (i.e., beliefs, goals, intentions, action, events, and communication messages) are captured as run-time observations using a logging mechanism.

Interpretation. The observations are a manifestation of agent behavior; actual agent behavior must be interpreted with respect to the user's knowledge of expected behavior. The Tracer Tool can create an interpretation of the observations using the modeled background knowledge. The interpretation is visualized as a relational graph, similar to the background knowledge.

Explanation. When the behavior of the implementation changes due to maintenance or redesign activities, the Tracer Tool provides suggestions to update the background knowledge. Since the user must review and accept all modifications to the background knowledge, the user's comprehension is also kept updated with respect to the current implementation. To help the user decide whether a suggestion is valid or to further clarify causal factors of a particular observation, explanations can be generated by the Tracer Tool.

Verification. Given the interpretation of a system execution, patterns of agent behavior can be readily discovered. As a result, anomalous behavior can also be detected as any observation that does not conform to a behavior pattern. Currently, anomaly detection is a manual task that involves discovering behavioral patterns in the interpretation, a task at which humans are extremely good. The Tracer Tool provides a capability to verify the expected behavior (represented as causal graphs) against the actual behavior of the implemented system.

For more complex behavioral properties, the **TTL Checker** by [1] provides automated verification of such properties using the observations collected by the Tracer Tool. The TTL Checker is a tool that enables the automated verification of complex dynamic properties against traces. It is based on the predicate logical Temporal Trace Language (TTL) [1]. In this language, to describe dynamic properties of complex agent systems, explicit reference is made to *states*, *time points* and *traces*. Dynamic properties can be formulated that relate a state at one point in time to a state at another point in time. A simple example is the following informally stated dynamic property for belief creation:

if the agent observes at t_1 that it is raining,
then the agent will believe that it is raining

In formalized (TTL) form this property looks as follows:

$$\forall t_1 [\text{state}(\gamma, t_1) \models \text{observes}(A, \text{itsraining}) \Rightarrow \\ \exists t_2 \geq t_1 \text{state}(\gamma, t_2) \models \text{belief}(A, \text{itsraining})]$$

To enable the automated verification of dynamic properties specified in TTL against formal traces (or logs), the TTL Checker takes a dynamic property and one or more formal traces as input, and checks whether the dynamic property holds for the traces.

3. INTEGRATED ANALYSIS APPROACH

Combining the advantages of the Tracer Tool and TTL Checker, an integrated approach is presented for the analysis and comprehension of agent-based software. This section describes the constituent steps in this approach.

1. Create a number of traces (or logs) of executions of a real agent system, in terms of agent concepts such as beliefs, desires, and actions. This step is done using the Tracer Tool (in particular, using the logging mechanism mentioned in Section 2). Normally, the logs are represented in XML format. To facilitate user understanding of the logs, a translation program has been written that translates the logs to traces in TTL format. As a result, the traces can also be loaded into a specific TTL visualization tool.

2. Formalize a number of relevant dynamic properties in TTL. This step is done using a graphical editor for TTL (see [1] for details). Two types of dynamic properties can be distinguished: *generic* dynamic properties and *domain-specific* dynamic properties. Generic dynamic properties are those that are independent of the domain. A simple example is 'Communication Correctness', i.e., "all messages sent from agent a_1 to agent a_2 are received by agent a_2 ". In TTL, this property looks as follows:

$$\forall \gamma_1: \text{TRACES } \forall t: \text{TIME } \forall a_1, a_2: \text{AGENT } \forall m: \text{MESSAGE} \\ [[\text{state}(\gamma_1, t) \models \text{sending_message_to}(a_1, a_2, m) \wedge a_1 \neq a_2] \\ \Rightarrow \exists \gamma_2: \text{TRACES } [\text{state}(\gamma_2, t) \models \text{receiving_message_from}(a_2, a_1, m)]]$$

The TTL checker contains a library of such generic dynamic properties, which can be re-used in any domain. Domain-specific dynamic properties have to be specified by hand for every new domain. Due to the expressiveness of the TTL language, complex behavioral properties can be specified, e.g., relating events at many different time points and different traces to each other.

3. Automatically check the dynamic properties created in step 2 against the logs created in step 1. This step is done using the TTL Checker (see [1]). As a result of this step, the user knows exactly which properties hold and which do not hold for the traces. Moreover, in case a property does not hold, the checker can provide detailed information about the position in the trace where the property fails. Usually this information has the form of a counter example. For example, if the aforementioned property 'Communication Correctness' fails, then the TTL Checker shows an example of a message that is sent from agent a_1 to a_2 , but it is not received by a_2 .

4. Explain failed properties. This step is done using the Tracer Tool (in particular, the explanation mechanism described in Section 2). Given a counter example from the TTL Checker due to a failed property, the user can examine the exact observations and its relation to other observations in the Tracer Tool's visualization. Observation details provide further evidence to track down the source of the problem, such as the type of communication message, who sent it, and why it was sent.

4. SUMMARY

Software comprehension is an essential issue during the development, maintenance, and debugging of complex agent systems. Nevertheless, existing techniques (such as reverse engineering [2] and model-checking [3]) face a number of difficulties, in particular (1) the high complexity and sophistication of the software, (2) the translation gap between low-level implementation structures and high-level design concepts, and (3) the large amount of time and effort demanded to gather a comprehensive picture of the software.

To deal with these problems, this paper presents a novel approach to analyze the behavior of agent systems. The approach is the result of the collaboration between the Tracer Tool for comprehension of agent software [7] and the TTL Checker for automated checking of dynamic properties of agent systems [1]. The integrated approach comprises four important elements, i.e. (1) logging of agent system executions in terms of high-level design concepts, (2) formalization of desired dynamic properties, (3) automated verification of dynamic properties against logs, and (4) computer-aided explanation of unexpected properties.

To test the proposed approach, it has been applied in the domain of Unmanned Aerial Vehicles (UAV), see [8]. First, using the Tracer Tool, a number of logs of the UAV system were generated in terms of agent concepts. Next, generic and domain-specific dynamic properties were specified in TTL, and checked against the generated logs. Finally, several of the unexpected properties were explained using the Tracer Tool. Using this approach, the software developers succeeded in locating some undesired behavior, which subsequently allowed them to investigate the causes for such behavior. Thus, the approach turned out useful in the comprehension and analysis of agent software.

Concerning related work, existing research addresses the different elements of the presented approach separately. For example, in [6] several tools are suggested for the collection of trace files (consisting of events and corresponding time parameters) in a distributed system. To examine the collected traces, techniques for querying distributed databases are proposed. This approach is similar to our approach in the sense that both make use of logging and analyzing real executions of an implemented system. A difference is that our approach automates reasoning further by leveraging agent concepts of agent-based systems. Furthermore, various papers, such as [10], describe tools for the visualization of agent-based system behavior based on the presumption that visualization is essential to facilitate efficient debugging of agent-based systems. This approach is also adopted in this paper to aid rapid comprehension of agent behavior in the implemented system. In addition, a variety of analysis tools can process constructs in the visualization to further help automate agent software comprehension, as shown in this paper.

Obviously, in an ideal situation the verification of dynamic properties is done exhaustively (i.e., against all theoretically possible traces instead of a practically given set of traces). However, due to the expressiveness of the TTL language, this process is highly complex. Currently, the possibilities are investigated to combine recent model-checking techniques for real-time systems (such as [9]) with the presented approach.

5. ACKNOWLEDGMENTS

The authors are grateful to Jan Treur for his constructive comments, and to David Han for the development and analysis of the agents in the UAV system.

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

6. REFERENCES

- [1] Bosse, T., Jonker, C.M., Meij, L. van der, and Treur, J. A Temporal Trace Language for the Formal Analysis of Dynamic Properties. Technical Report, Vrije Universiteit Amsterdam, 2005.
- [2] Chikofsky, E.J. and Cross, J.H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, vol. 7(1), 1990, pp. 13-17.
- [3] Clarke, E.M., Grumberg, O., and Peled, D.A. *Model checking*. MIT Press, 2000.
- [4] Edmonds, B. and Bryson, J. The Insufficiency of Formal Design Methods. In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS'04*, 2004, pp. 938-946.
- [5] Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H.A., Mylopoulos, J., Perelgut, S.G., Stanley, M., and Wong, K. The Software Bookshelf. *IBM Systems Journal*, vol. 36(4), 1997, pp. 564-593.
- [6] Garcia-Molina, H., Germano, F., and Kohler, W. Debugging a Distributed Computer System. *IEEE Transactions on Software-Engineering*, SE-10 (2), 1984, pp. 210-219.
- [7] Lam, D.N. and Barber, K.S. Comprehending Agent Software. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS'05*, 2005, pp. 586-593.
- [8] Lam, D. N. and Barber, K. S. Debugging Agent Behavior in an Implemented Agent System. In: Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (Eds.), *Programming Multi-Agent Systems* (Selected Revised and Invited Papers), Springer LNCS, vol. 3346, 2005, pp. 103-125.
- [9] Larsen, K.G., Petterson, P., and Yi, W. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2), 1997, pp. 134-152.
- [10] Ndumu, D., Nwana, H., Lee, L., and Collis, J. Visualising and Debugging Distributed Multi-Agent Systems. In: *Proceedings of the Third Annual Conference on Autonomous Agents*, 1999, pp. 326-333.