Manuel Kolp
Paolo Bresciani
Brian Henderson-Sellers
Michael Winikoff (Eds.)

LNAI 3529

Agent-Oriented Information Systems III

7th International Bi-Conference Workshop, AOIS 2005
Utrecht, Netherlands, July 2005
and Klagenfurt, Austria, October 2005
Revised Selected Papers

Springer

TP18-53
A638
2005

Manuel Kolp Paolo Bresciani
Brian Henderson-Sellers Michael Winikoff (Eds.)

Agent-Oriented Information Systems III

7th International Bi-Conference Workshop, AOIS 2005
Utrecht, Netherlands, July 26, 2005
and Klagenfurt, Austria, October 27, 2005
Revised Selected Papers

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Manuel Kolp
Catholic University of Louvain (UCL)
School of Management (IAG), Information Systems Research Unit (ISYS)
1, Place des Doyens, 1348 Louvain-La-Neuve, Belgium
E-mail: kolp@isys.ucl.ac.be

Paolo Bresciani
European Commission
DG Information Society and Media, Unit D3: Software Technologies
Avenue de Beaulieu 29, level 4, office 49, 1049 Brussels, Belgium
E-mail: paolo.bresciani@ec.europa.eu

Brian Henderson-Sellers
University of Technology, Sydney
Faculty of Information Technology
P.O. Box 123, Broadway, NSW 2007, Australia
E-mail: brian@it.uts.edu.au

Michael Winikoff
RMIT University
School of Computer Science and Information Technology
Melbourne, VIC 3001, Australia
E-mail: winikoff@cs.rmit.edu.au

Library of Congress Control Number: 2006936083

CR Subject Classification (1998): I.2.11, H.4, H.3, H.5.2-3, C.2.4, I.2

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN	0302-9743
ISBN-10	3-540-48291-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-48291-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11916291 06/3142 5 4 3 2 1 0

Lecture Notes in Artificial Intelligence 3529

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Lecture Notes in Artificial Intelligence (LNAI)

¥410.00元)

Preface

Information systems underpin today's business and entertainment. The means by which these information systems have been developed has changed over the years. Although the current paradigm is to use object-oriented concepts, a new set of concepts, focussed on agent technology, is starting to be evaluated. Agents offer higher level abstractions (than objects) for the conceptualization, design and implementation of information systems. Agents have autonomy, can reason and can coordinate within societies of agents.

The AOIS series of workshops explores the potential for facilitating the increased usage of agent technology in the creation of information systems in the widest sense. In 2005, two AOIS workshops were held internationally. The first was affiliated with the AAMAS 2005 meeting in July in Utrecht in The Netherlands and chaired by Henderson-Sellers and Winikoff and the second with ER 2005 in November in Klagenfurt in Austria and chaired by Kolp and Bresciani. The best papers from these meetings were identified and authors invited to revise and possibly extend their papers in the light of reviewers' comments and feedback at the workshop.

We have grouped these papers loosely under four headings: Agent behavior, communications and reasoning; Methodologies and ontologies; Agent-oriented software engineering; and Applications. These categories fairly represent the breadth of current AOIS research as well as encompassing the papers presented at the two AOIS workshops. We trust you will find the content of these selected and revised papers to be of interest and utility.

Since the papers presented at the Utrecht workshop were not formally published, some of the authors chose not to make any significant extension to their papers. On the other hand, the Klagenfurt workshop papers were published by Springer as part of the ER proceedings and thus have been significantly extended before acceptance for this volume. All invited papers for this volume were re-reviewed (in their extended forms) by three members of the Program Committee prior to acceptance. We wish to thank all authors for undertaking the necessary revisions and meeting the editorial deadlines.

September 2006

Manuel Kolp
Paolo Bresciani
Brian Henderson-Sellers
Michael Winikoff

Organization

Workshop Co-chairs

Manuel Kolp (Catholic University of Louvain, Belgium)
Paolo Bresciani (IRST-ITC, Italy)
Brian Henderson-Sellers (University of Technology, Sydney, Australia)
Michael Winikoff (RMIT, Australia)

Steering Committee

Yves Lesperance (York University, Canada)
Gerd Wagner (Eindhoven University of Technology, Netherlands)
Eric Yu (University of Toronto, Canada)
Paolo Giorgini (University of Trento, Italy)

Program Committee

Carole Bernon (University Paul Sabatier, Toulouse, France)
Brian Blake (Georgetown University, Washington DC, USA)
Paolo Bresciani (ITC-IRST, Italy)
Jaelson Castro (Federal University of Pernambuco, Brazil)
Luca Cernuzzi (Universidad Católica Nuestra Señora de la Asunción, Paraguay)
Massimo Cossentino (ICAR-CNR, Palermo, Italy)
Luiz Cysneiros (York University, Toronto)
John Debenham (University of Technology, Sydney)
Scott DeLoach (Kansas State University, USA)
Frank Dignum (University of Utrecht, Netherlands)
Paolo Donzelli (University of Maryland, College Park, USA)
Bernard Espinasse (Domaine Universitaire de Saint-Jérôme, France)
Stéphane Faulkner (University of Namur, Belgium)
Behrouz Homayoun Far (University of Calgary, Canada)
Innes Ferguson (B2B Machines, USA)
Alessandro Garcia (PUC Rio)
Chiara Ghidini (ITC-IRST, Italy)
Aditya Ghose (University of Wollongong, Australia)
Marie-Paule Gleizes (University Paul Sabatier, Toulouse, France)
Cesar Gonzalez-Perez (University of Technology, Sydney, Australia)
Giancarlo Guizzardi (University of Twente, Netherlands)
Igor Hawryszkiewycz (University of Technology, Sydney, Australia)
Brian Henderson-Sellers (University of Technology, Sydney, Australia)
Carlos Iglesias (Technical University of Madrid, Spain)

Table of Contents

Automated Interpretation of Agent Behaviour

D.N. Lam and K.S. Barber

The University of Texas at Austin
The Laboratory for Intelligent Processes and Systems
dnlam@lips.utexas.edu, barber@lips.utexas.edu

Abstract. Software comprehension, which is essential for debugging and maintaining software systems, has lacked attention in the agent community. Comprehension has been a manual process, involving the analysis and interpretation of log files that record agent behaviour in the implemented system. This paper describes an approach and tool to automate creating interpretations of agent behaviour from observations of the implementation execution, thus helping users (i.e. designers, developers, and end-users) to understand the motivations of agent actions. By explicitly modelling the user's comprehension of the implemented system as background knowledge for the tool, feedback can be provided as to whether the user's comprehension accurately represents the implementation's behaviour and, if not, how it can be corrected. Additionally, with the aid of the Tracer Tool, many of the manual tasks are automated, such as verifying that agents are behaving as expected, identifying unexpected behaviour and generating explanations for any particular observation.

1 Introduction

Agents are distributed software entities that are capable of autonomous decision-making. Besides being motivated by its own goals, an agent's behaviour is influenced by interactions with other agents (i.e. their goals, beliefs and intentions), by events that have occurred in the past and by the current situation. With so many factors that can influence an agent's decision, end-users may not trust the agent's decision, and developers may have difficulty debugging the implementation. Software designers, developers and end-users often need to comprehend why an agent acted in a particular way when situated in its operating environment, which itself can be unpredictable and uncertain. Currently, the process of comprehending agent behaviour is done manually by *interpreting* the observations from the implementation executions to create a connected, comprehensive view of what the software is doing. The interpretation process links (usually with a causal link) actual observations together using the user's comprehension (or background knowledge of expected behaviour). In essence, an interpretation compares the actual implementation behaviour with expected behaviour, which may have been created by the user from the software design, previous experience, intuition etc.

Considering the complexities of agent software (e.g. autonomous decision-making and a high degree of interaction) and the usual disparity between software design and implementation, software comprehension is a difficult, time-consuming and tedious process. To alleviate these issues, this research aims to automate the comprehension process as much as possible. This paper describes (1) how the Tracer Tool can be used to help build and verify a model of the user's comprehension of the implemented agent system's behaviour (i.e. background knowledge) and (2) how an interpretation of agent behaviour can be automatically generated from the background knowledge and recorded observations.

Sophisticated software such as an agent-based system 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 low abstraction level, their dependence on analyzing source code, their lack of automation to help decipher tremendous amounts of collected data and their lack of a model for how much the user understands. Taking the formal approach to modelling systems, model-checking facilitates comprehension by verifying properties of systems but 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 the actual system.

To remedy limitations of current comprehension techniques, this research offers a novel approach to computer-aided software comprehension that involves: (1) modelling the user's comprehension of the system as background knowledge usable by tools, (2) ensuring that the user's comprehension accurately reflects the actual system and (3) generating interpretations and explanations as evidence of comprehension.

This paper describes an approach and tool that builds on the ideas from reverse engineering and model-checking to better assist the human user (of various skill levels) in comprehending agent-based software. Section 2 reviews limitations of existing work and highlights advantages that are used in this research. Section 3 presents the formulation of the problem and the approach employed to automate building the interpretation of agent behaviour. Section 4 describes how the Tracer Tool implements the approach. Section 5 demonstrates how the interpretation can be used to generate explanations. Finally, Section 6 summarizes the contributions of this research.

2 Background

Agent concepts (i.e. beliefs, goals, intentions, actions, events and messages) are abstractions of low-level implementation constructs (e.g. data structures, classes and variables) that make designing and communicating the design easier. Though agent concepts help in designing software for sophisticated and distributed domains, there has been little research in leveraging them for the expensive maintenance phase of software engineering. Since software designs use agent concepts to describe agent structure (e.g. an agent encapsulates localized beliefs, goals, and intentions) and behaviour (e.g. an agent performs an action when it believes an 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 [1]. In this research, agent concepts are used to take advantage of the user's intuitive knowledge of agent-based systems to comprehend agent behaviour in the implementation. The set of concepts can be extended or replaced by practically any set of concepts that are relevant in understanding the behaviour of a software system and influential factors that affect those behaviours.

Software comprehension, which historically has been associated with program comprehension and reverse engineering, involves extracting and representing the structural and behavioural aspects of the implementation in an attempt to recreate the intended design of the software. Software comprehension is motivated by the fact that 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 reverse engineering (RE) tools and techniques.

RE tools (e.g. Rigi [2] and PBS [3]) analyse 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). Besides being limited to supported programming languages, 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" [4]. To attain an understanding of agent behaviour, the models resulting from the comprehension process must be at the abstraction level at which agent concepts are the elemental or base concepts.

In addition to static analysis of the source code, dynamic analysis tools (e.g. SCED [5] and Hindsight [6]) can create flowcharts, control-flow and state diagrams. However, these tools 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 behaviour (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.

To deal with the large amount of data resulting from source code or execution analysis, some RE tools (e.g. SoftSpec [7]) allow users to query a relational database of gathered data. However, most RE tools leave it up to the user to parse, interpret and digest the data. The research described in this paper deals with the large amount of data by automating data interpretation for the user.

Instead of a list of unconnected, detailed data that the user must relate manually, the presented solution automatically relates run-time observations together in a causal graph. This is similar to the GUPRO toolset [8], where source code is transformed into graphs, except that the graphs nodes are in terms of agent concepts.

As described, 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, model-checking techniques in general do not verify the accuracy of the "model" with respect to the actual system (often referred to as the translation gap problem). Hence, any checked properties may not apply to the actual implementation. Additionally, the model must be made simple enough such that the model-checker can search the entire state-space. 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 the actual system.

3 Building the Interpretation

When a user tries to comprehend agent behaviour in the implemented system, the user is essentially building an interpretation by observing and examining agent actions, communicated messages, environmental events and any other run-time data that can be acquired from the implementation. As shown in Fig. 1, background knowledge about the expected behaviour of the implemented system is required to relate the otherwise unconnected observations together. 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.

In this research, K is modelled using a semantic network (i.e. directed graph) of agent concepts that are interconnected by causal relations. The current set of agent concepts includes *goal*, *belief*, *intention*, *action*, *event* and *message* – the set can be extended to include other concepts that may be of interest to the user. For example, in Fig. 1, the background knowledge for an agent's behaviour denotes an intention that is influenced by two different beliefs (denoted by a circle and square). The intention causes an action to occur, which in turn affects one of the beliefs.

This research takes advantage of agent concepts to create interpretations of agent behaviour in the implemented system. Note that K represents a

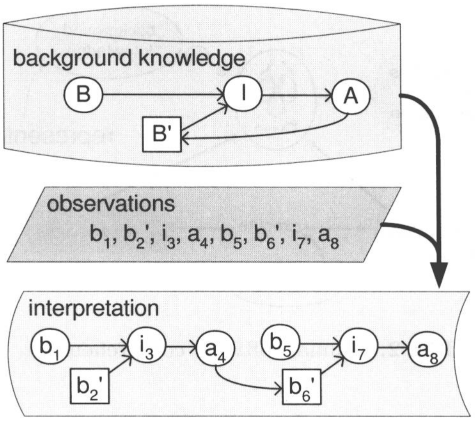

Fig. 1. An interpretation for an agent, given the background knowledge K and observations O_sSS

behavioural pattern and, thus, can have cycles in the graph. However, the interpretation, which consists of actual observations and their relationships, does not have cycles.

To build an interpretation, observations are mapped to agent concepts in K and are linked together using relations defined in K. For example, observations b_1 and b_5 are mapped to agent concept B because the observations are beliefs about a target's state; b'_2 and b'_6 are mapped to B' because the observations are beliefs about the target's location; i_3 and i_7 are mapped to I; etc. Since I is causally related to B and B', directed edges are added between the appropriate nodes (e.g. from b_1 and b_2 to i_3) to relate the observations together. In other words, since the user expects beliefs about a target's state B to influence the agent's intention I, the user will create an interpretation where the corresponding observations for that agent are causally linked.

Background knowledge K is constructed by the user and describes how the agents are expected to behave in terms of the agent concepts. As shown in Fig. 2, the manual procedure for building comprehension can be expressed as

$$K' = update_{manual}(K, D, I, O_s) \tag{1}$$

where K is the previous background knowledge, D denotes 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:

$$O_s = observe(execute(I, s)) \tag{2}$$

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.

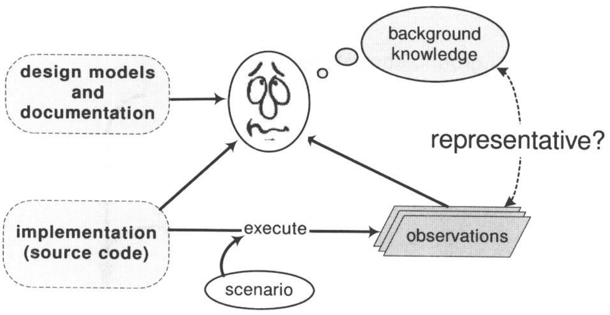

Fig. 2. Manual software comprehension

Due to human error or outdated design specifications, system behaviour described by K may be erroneous or inaccurate with respect to the actual behaviour of the system, particularly as the implementation is updated and maintained over time. To generate accurate interpretations, K must accurately reflect the implementation's actual behaviour. Using empirical techniques, the user must manually verify that the expected behaviour expressed as K is representative of the actual behaviour from the implementation. Due to complexities and uncertainties of some systems, agent behaviours cannot always be predicted from only the design specification in general [9]. Thus, the construction of K must incorporate empirical studies of the implementation.

The overall approach of this research is to build up the background knowledge K using observations from the actual implementation's executions, rather than relying on design specifications as it is in model-checking. 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 also 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 has seen. Consequently, the user does not have to digest all interpretations. Any new or inconsistent behaviours are automatically detected and brought to the attention of the user. Additionally, automatically generated suggestions and explanations can help the user deal with the anomalous behaviour.

The following describes the overall approach taken by this research to ensure the representativeness of the background knowledge. Functions begin with a lowercase letter (e.g. $interpret(K, O_s)$), while predicates begin with an uppercase letter (e.g. $Consistent(K, N_s)$).

As seen in Fig. 3, the reverse engineering approach helps the user by analyzing O_s to produce interpretations N_s, which consists of models derived from observations O_s resulting from actual system behaviour: