Mark D. Ryan
John-Jules Ch. Meyer
Hans-Dieter Ehrich (Eds.)

LNCS 2975

Objects, Agents, and Features

International Seminar
Dagstuhl Castle, Germany, February 2003
Revised and Invited Papers

Springer

Lecture Notes in Computer Science 2975
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Mark D. Ryan John-Jules Ch. Meyer
Hans-Dieter Ehrich (Eds.)

Objects, Agents, and Features

International Seminar
Dagstuhl Castle, Germany, February 16-21, 2003
Revised and Invited Papers

Springer

Volume Editors

Mark D. Ryan
University of Birmingham, School of Computer Science
Birmingham B15 2TT, UK
E-mail: M.D.Ryan@cs.bham.ac.uk

John-Jules Ch. Meyer
Utrecht University, Institute of Information and Computing Sciences
Intelligent Systems Group
Padualaan 14, De Uithof, P.O. Box 80.089
3508 TB Utrecht, The Netherlands
E-mail: jj@cs.uu.nl

Hans-Dieter Ehrich
Technische Universität Braunschweig, Abteilung Informationssysteme
Postfach 3329, 38023 Braunschweig, Germany
E-mail: HD.Ehrich@tu-bs.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, D.1.5, D.1.3, C.2.4, I.2.11, F.3

ISSN 0302-9743
ISBN 3-540-21989-7 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11006299 06/3142 5 4 3 2 1 0

Preface

In recent years, concepts in object-oriented modeling and programming have been extended in several directions, giving rise to new paradigms such as agent-orientation and feature-orientation.

This volume came out of a Dagstuhl seminar exploring the relationship between the original paradigm and the two new ones. Following the success of the seminar, the idea emerged to edit a volume with contributions from participants – including those who were invited but could not come. The participants' reaction was very positive, and so we, the organizers of the seminar, felt encouraged to edit this volume. All submissions were properly refereed, resulting in the present selection of high-quality papers in between the topics of objects, agents and features. The editors got help from a number of additional reviewers, viz. Peter Ahlbrecht, Daniel Amyot, Lynne Blair, Jan Broersen, Mehdi Dastani, Virginia Dignum, Dimitar Guelev, Benjamin Hirsch, Maik Kollmann, Alice Miller, Stephan Reiff-Marganiec, Javier Vazquez-Salceda, and Gerard Vreeswijk.

Finally, we would like to take this opportunity to thank all the persons involved in the realization of the seminar and this book: attendees, authors, reviewers, and, last but not least, the staff from Schloss Dagstuhl and Springer-Verlag.

February 2004 The Editors

Table of Contents

Objects, Agents, and Features: An Introduction

John-Jules Ch. Meyer[1], Mark D. Ryan[2], and Hans-Dieter Ehrich[3]

[1] Institute of Information and Computing Sciences
Utrecht University
The Netherlands
jj@cs.uu.nl
[2] School of Computer Science
University of Birmingham
UK
M.D.Ryan@cs.bham.ac.uk
[3] Dept. of Information Systems
TU Braunschweig
Germany
HD.Ehrich@tu-bs.de

1 Introduction

There are many ways of structuring software, and the seminar focussed on an established one (object-orientation) and two emerging ones (agent-orientation and feature-orientation).

The *object* paradigm is now widely used in software technology (with programming languages like C++ and Java, and OO modelling frameworks such as UML). However, the theoretical foundations of the object paradigm are not settled yet, although clean concepts and reliable foundations are more and more demanded not only by academia but also by practitioners. In particular, the precise meaning of UML concepts is subject to wide debate.

Agents are more special kinds of objects, having more autonomy, and taking more initiative. For this reason, agent-oriented programming is sometimes referred to as 'subject-oriented' rather than 'object-oriented', indicating that an agent is much more in control of itself than an object which is manipulated by other entities (objects). There is some work on investigating typical object notions like inheritance in the context of agents. An interesting question is whether this is a fruitful way to go. Typically, agents are thought of being endowed with 'mental states' involving concepts like knowledge, belief, desires and goals, in order to display autonomous and in particular pro-active behaviour.

Features are optional extensions of functionality which may be added to a software product, in order to reflect changes in requirements. They also cut across the class structure, because implementing a feature typically involves updating several classes or objects. The more complex the system is, the harder it is to add features without breaking something; this phenomenon has been dubbed the 'feature interaction problem'. Because users like to think of a system as comprising a base system together with a number of features on top, features could potentially be seen as a structuring mechanism rivalling objects and agents.

M.D. Ryan et al. (Eds.): Objects, Agents, and Features, LNCS 2975, pp. 1–7, 2004.

The following two subsections give brief introductions into the ideas behind these paradigms.

1.1 The Agent Structuring Mechanism

Intelligent Agents. In the last decade or so the subject of 'intelligent agents' has been getting increasingly important in both the fields of software engineering and artificial intelligence [24]. (Intelligent) agents are software (or hardware) entities that display a certain degree of *autonomy* while operating in an environment (possibly inhabited by other agents) that is not completely known by the agent and typically is changing constantly. Agents possess properties like *reactiveness*, *proactiveness* and *social behaviour*, often thought of as being brought about by mental or cognitive attitudes involving knowledge, beliefs, desires, goals, intentions,..., in the literature often referred to as '*BDI attitudes*' (for beliefs, desires, intentions).

The area of agent technology covers the foundations as well as the design, implementation and application of intelligent agents, both stand alone and within (the context of) multi-agent systems. The foundations concern agent theories and in particular agent logics as a basis for agent architectures and the specification and verification of agent programs written in agent programming) languages. Design and implementation of agents involve the study of agent architectures and agent programming languages by means of which agents can be realised. Applications of agent technology are numerous (at least potentially), and range from intelligent personal assistants in various contexts to cognitive robots and e-commerce, for example [15].

Especially important is the study of *multi-agent systems*, which incorporates such issues as communication, coordination, negotiation and distributed reasoning/problem solving and task execution (e.g. distributed / cooperative planning and resource allocation). As such the area is part of the field of distributed AI. An important subfield is that of the investigation of agent *communication languages* that enable agents to communicate with other agents in a standardized and high-level manner.

Logics for Intelligent Agents. In order to describe and specify the behaviour (and the BDI attitudes, more in particular) of intelligent agents a number of logics have been proposed. The most well-known are those of Cohen and Levesque [3] and Rao and Georgeff [22]. The former is based on a linear-time temporal logic and is augmented with modal operators for belief and goals, and a possibility to express the performance of actions. On this basis Cohen and Levesque define intentions as certain (persistent) goals. In fact, the formalism is 'tiered' in the sense that it contains an 'atomic layer' describing beliefs, goals and actions of an agent, and a 'molecular layer' in which concepts like intention are defined in terms of the primitives of the atomic layer. The (title) slogan of Cohen and Levesque [3] is:

$$intention = choice + commitment$$

meaning that an intention is a particularly chosen desire or goal to which the agent is committed so that it will not drop it without a good reason (e.g. the goal is reached or the agent realises that it cannot ever be achieved any more).

The framework of Rao & Georgeff [22] is known as 'BDI logic', and is based on the branching-time temporal logic CTL*, to which modal operators for belief, desire/goal and intention are added. *A priori* these three modalities are put in the framework as independent modal operators (with their own semantic accessibility relations) but Rao & Georgeff give a systematic account of how these modalities may interact (yielding constraints on the general models for the logic). BDI logic has inspired its inventors and many others to think about the architectures of BDI agents (such as PRS - Procedural Reasoning System, [7]) and also about languages for programming agents (such as AgentSpeak(L) [21]).

Since in both of the above approaches the dynamics of BDI agents cannot be properly specified mainly due to the lack of an adequate account of the results of actions, KARO logic for describing intelligent agents was proposed, based on *dynamic logic* [9], a well-known action logic, rather than temporal logic [18, 14, 19].

Agent-Oriented Programming Languages. Of course, if our aim is to realize intelligent agents logics with which one may describe or specify them is not enough. We need means to construct these agents. A number of possible options is now available. One may directly think of architectures of agent-based systems where components of agents are given a place and more or less abstractly it is indicated how these components interact with each other. An example is the BDI architecture [22] in which beliefs, desires and intentions are realized by means of belief, goal and plan bases, and where a generic program indicates how a sense-reason-act cycle is implemented using these bases. A disadvantage of this approach is that it is both very complex and it is hard to link an agent to a logical specification by some agent logic (even a BDI logic).

Therefore other researchers have moved into the direction of building agents by means of special so-called agent-oriented programming languages, in which agent concepts are used explicitly.

The first researcher who proposed such a language was Yoav Shoham [23], the language AGENT0. Besides basic actions having to do with the particular application one has in mind (called 'private actions') such as a move action in the case of a robot, in AGENT0 one can use 'mental actions' such as updating a belief base with incoming messages, the addition of commitments (actions to be performed by the agent) and the removal of non-executable commitments, and one may also use communication actions of informing other agents and (un)requesting the performance of actions to other agents. The basic loop of the AGENT0 interpreter performs an update of the agent's mental state, that is, it reads the current messages, updates the belief and commitment bases of the agent, and consequently executes the appropriate commitments, possibly resulting in further belief updates.

Although the link with agent concepts as also used in agent logics is much more direct (via the explicit use of beliefs and commitments), in a language such

as AGENT0 there still remains a problem of directly linking agent programs with a specification in an agent logic. Some researchers have therefore proposed to use an executable agent logic, where the ideal is to directly execute a specification written in the logic that contains elements of temporal logic as well as some agent notions such as beliefs and goals. An example of this approach is the work on (Concurrent) METATEM [6]. Related is the work on the language (CON)GOLOG [17, 8] and its many variants: here programs written in that language are employed to guide a theorem prover to find a program (plan) to reach the goals of an agent. Other approaches that were proposed include the agent languages AgentSpeak(L) [21] and, inspired by this, 3APL [11, 12, 10], where it is attempted to match equipping the language with mental BDI-like concepts with giving a rigid formal semantics in which BDI-models are incorporated and which renders programs written in the language amenable to the formal analysis of correctness issues [2, 13].

1.2 The Feature Structuring Mechanism

When users evaluate competing systems and try to figure out which is the best one, they frequently do so on the basis of the *features* that the systems offer. For example, when buying a printer, one might discriminate between features on the basis of whether they have features such as:

- double-sided printing;
- ethernet compatibility;
- accepts Postscript; etc.

A given printer might or might not have each of these features.

It is useful to think of a system as comprising a 'base' system together with a set of features. The idea of the feature structuring mechanism is to describe computer systems explicitly in those terms. In its pure form, we decompose a system not in terms of objects, but in terms of a minimal *core* (which we call the base system), together with some extensions (called features).

Structuring by features is often orthogonal to structuring by objects. If we consider adding a new feature to an existing system, we find that the new feature cuts across the object structuring. Introducing a new feature does not affect just one object, but a set of them.

Features have become popular in the domain of telephony, because telephone companies have sought to gain market advantage by offering new features, and charging for them separately. Features of a telephone system include

- Call forward on busy: a call to a busy terminal is forwarded to another (pre-selected) terminal;
- Voicemail on busy: a caller to a busy terminal is offered voicemail;
- Ring back when free: a caller to a busy terminal is offered the possibility of being notified when the destination terminal becomes free.

There are hundreds of such features defined in the literature.

Features should be defined as independently as possible of the systems to which they will be added, in order to make them more abstract and to enable them to be integrated into a larger class of systems. As a consequence, when we define a feature we normally do not know what other features may be deployed with it. This means that when we do put features together, they may interact in unexpected ways. The problem of predicting and coordinating interactions between features is called the *feature interaction problem*, and receives considerable attention in the literature [5, 4, 25, 20].

Examples of feature interactions in telecommunications systems are:

- "Call Forward on Busy" and "Voice Mail on Busy": both these features try to take control of a second (incoming) call to the subscriber. This is inconsistent, so one cannot allow both features to be active on the same phone.
- The "Ring Back When Free" (RBWF) attempts to set up a call for the subscriber to a callee whose line is engaged as soon as the line becomes free. The interaction of RBWF and "Call Forward Unconditional" (CFU) leads to consistent, but potentially undesirable behaviour:
  x requests RBWF from **y**, then CFU to **z**.
  z will be notified when **y** becomes available.
  However, it was **x** who requested the notification.

Some systems have features already built-in, ready to be enabled by users, while other systems are built using an architecture which supports adding features. Examples of the latter kind include systems allowing plug-ins, like web browsers and media players, and systems allowing user-written packages like GNU Emacs and LaTeX. All of these systems are prone to feature interaction. For example, LaTeX styles may not work as intended when loaded in the wrong order, or in some cases not be compatible at all. These problems can usually be traced down to the fact that two features manipulate the same entities in the base system, and in doing so violate some underlying assumptions about these entities that the other 'features' rely on.

Feature-oriented programming is related to aspect-oriented programming, and to superimpositions. We consider each of these in turn. Aspect-oriented programming is a technique for managing program changes which cut across the existing structure. For example, AspectJ [1] provides constructs allowing the programmer to stipulate that after every call to a certain method, some other piece of code should be run. An aspect is a collection of such changes, introduced for a particular purpose. In this way, aspects could be used to implement features.

Superimpositions [16] are another technique for making cross-cutting changes to a program, though with slightly different motivation. The focus in superimposition is on adding extra code to a given program, usually to make it better behaved with respect to other concurrently running programs. In the classic example of superimposition, extra code is added to enable processes to respond to interrogations from a supervisory process about whether they are awaiting further input, and this enables smooth termination of the system.

The purpose of this volume is to explore the connections between system structuring mechanisms.

References

1. The AspectJ project. `www.aspectj.org`. Accessed January 2004.
2. R. H. Bordini, M. Fisher, C. Pardavila and M.Wooldridge. Model checking AgentSpeak. In Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03), pages 409-416, Melbourne, 2003.
3. P.R. Cohen and H.J. Levesque, Intention is Choice with Commitment, *Artificial Intelligence* 42, 1990, pp. 213–261.
4. D. Amyot and L. Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press, 2003.
5. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000.
6. M. Fisher, A Survey of Concurrent METATEM – The language and Its Applications, in: *Temporal Logic – Proc. of the First Int. Conf.* (D.M. Gabbay and H.J. Ohlbach, eds.), LNAI 827, Springer, Berlin, 1994, pp. 480–505.
7. M.P. Georgeff and A.L. Lansky, Reactive Reasoning and Planning, in: Proc. 3rd Nat. Conf. on Artif. Intell. (AAAI-87), Seattle. WA, 1987, pp. 677-682.
8. G. de Giacomo, Y. Lespérance and H.J. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence* 121(1-2), 2000, pp. 109–169.
9. D. Harel, Dynamic Logic, in: D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic, Vol. II*, Reidel, Dordrecht/Boston, 1984, pp. 497–604.
10. K.V. Hindriks, Agent Programming Languages: Programming with Mental Models, PhD. Thesis, Utrecht University, 2001.
11. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, Formal Semantics for an Abstract Agent Programming Language, in: *Intelligent Agents IV* (M.P. Singh, A. Rao and M.J. Wooldridge, eds.), LNAI 1365, Springer, 1998, pp. 215–229.
12. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, Agent Programming in 3APL, *Int. J. of Autonomous Agents and Multi-Agent Systems* 2(4), 1999, pp.357–401.
13. K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, A Programming Logic for Part of the Agent Language 3APL, in: (Proc. First Goddard Workshop on) Formal Approaches to Agent-Based Systems (FAABS 2000) (Rash, J.L., Rouff, C.A., Truszkowski, W., Gordon, D. and Hinchey, M.G. eds.),, LNAI 1871, Springer, Berlin/Heidelberg, 2001, pp. 78-89.
14. W. van der Hoek, B. van Linder and J.-J. Ch. Meyer, An Integrated Modal Approach to Rational Agents, in: *Foundations of Rational Agency* (M. Wooldridge and A. Rao, eds.), Applied Logic Series 14, Kluwer, Dordrecht, 1998, pp. 133–168.
15. N.R. Jennings and M.J. Wooldridge, *Agent technology: Foundations, Applications, and Markets*, Springer, Berlin, 1997.
16. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
17. H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin and R.B. Scherl, GOLOG: A Logic Programming Language for Dynamic Domains, J. of Logic Programming 31, 1997, pp. 59–84.

18. B. van Linder, Modal Logics for Rational agents, PhD. Thesis, Utrecht University, 1996.
19. J.-J. Ch. Meyer, W. van der Hoek and B. van Linder, A Logical Approach to the Dynamics of Commitments, *Artificial Intelligence* 113, 1999, 1–40.
20. M. C. Plath and M. D. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 2001.
21. A.S. Rao, AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: *Agents Breaking Away* (W. van der Velde and J. Perram, eds.), LNAI 1038, Springer, Berlin, 1996, pp. 42–55.
22. A.S. Rao and M.P. Georgeff, Modeling Rational Agents within a BDI-Architecture, in *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)* (J. Allen, R. Fikes and E. Sandewall, eds.), Morgan Kaufmann, 1991, pp. 473–484.
23. Y. Shoham, Agent-Oriented Programming, *Artificial Intelligence* 60(1), 1993, pp. 51–92.
24. M.J. Wooldridge and N.R. Jennings (eds.), *Intelligent Agents*, Springer, Berlin, 1995.
25. P. Zave, FAQ Sheet on Feature Interaction. At `www.research.att.com/~pamela/faq.html`, accessed January 2004.