Multiagent Based Problem-solving in a Mobile Environment

Odd Erik Gundersen
odderik@idi.ntnu.no

Anders Kofod-Petersen
anderpe@idi.ntnu.no

Department of Computer and Information Science
Norwegian University of Science and Technology

Abstract

As users are becoming more mobile and information services are increasingly being personalised, the need for computer systems to adapt to the user is also increasing. One important aspect of this adaptation is the ability to solve problems in a dynamic environment. We propose a multiagent based approach to dynamically planning and problem solving. This approach allows for construction of plans based on available resources in the environment. We demonstrate the capabilities of this system by showing an implemented prototype.

1 Introduction

The average computer user is becoming more and more mobile. Users are bringing an increasing amount of data and computational power along, and are likely to have net access everywhere. With this movement of the computer from the desktop toward the ubiquitous paradigm described by Weiser [1], the computer system now has to adapt to the user's situation, instead of the user adapting to the computer. One particular interesting aspect of this adaptation is the ability to dynamically solve problems in a mobile environment.

Special considerations need to be taken into account when dynamically solving problems in a mobile environment. The same problem may need to be solved in different ways depending on where the mobile user is located when the problem arises. Hence, different plans may be generated to solve the same problem. A plan should take into consideration services available to the user of the mobile device at the time and place a problem is encountered.

The prototype developed and presented in this paper demonstrates a multi-agent based solution for co-ordinating the problem-solving process within a context-aware framework. The approach is evaluated in the on-going research in here.

The paper is organised as follows. In Section 2 related work in the literature is described. Section 3 introduces our approach for planning. In Section 4, the way plans are executed is described. Section 5 describes the implemented prototype. In section 6, an example is presented. Finally, Section 7 summarises the work done and points to future works.

2 Related work

Our work is based on the premise that agents take part in a multi-agent society containing benevolent agents that shares common goals. In such systems, agents co-operate to solve problems. Several systems based on this premise have been implemented, but due to space limitations we will only present three of the most relevant in this paper. Common to all of the systems presented here is that they represent plans as task hierarchies at the knowledge level [2]. A task describes what to do and how tasks may be decomposed into smaller, less complex tasks called subtasks. The decomposition continues until the granularity is satisfying. For instance, the task decomposition may stop when one or more agents are capable of executing the subtasks. Then, tasks may be assigned to one or more agents for execution.

RETSINA MAS is one of the most influential software agent systems and has also been based on the assumption of co-operating agents. The system has been thoroughly documented in several papers [3, 4]. RETSINA MAS distinguish between three different agent types, which are the *Interface* agents, *Task* agents and *Information* agents. *Interface* agents are the interface between the user and the system. The user specify a problem to be solved and the *Interface* agents translate the problem into a task which is handed to a *Task* agent. *Task* agents may decompose tasks into subtasks. The decomposition specifies a plan for how to solve a problem. One or more *Task* agents may cooperate to design the solution to the problem. *Task* agents query *Information* agents that are in control of different information sources. The received answers are assembeled by the *Task* agents into an answer to the initial problem. The answer of the initial problem is sent back to the requesting *Interface* agent, which present the solution to the user.

eHermes is a system that has been developed with ubiquitous computing in mind [5, 6]. Plans in form of task-hierarchies are referred to as *Missions*. *Missions* are generated on basis of a user request and the profile of the user making that request. Thus, the same problem may be solved in different ways according to which user makes a request. *Missions* are not generated by agents but a system component called the *Mission Generator*. Agents are not part of the planning process, they only participate in the problem-solving process. One agent, the *Mission Control* agent, is responsible for co-ordinating the agents performing the problem-solving. These problem-solving agents are assembled on demand from the mission specification, an agent component library, and ontologies. The agent assembly is done by a system component called the *Agent Generator*, which assemble agents on request from the *Mission Control* agent. The system is reported to be capable of solving problems in domains where the overall purpose of agents may be explicitly identified and expressed. Financial services are proposed as a suitable domain.

The Web Information Mediation (WIM) application [7] was developed as a part of the *ibrow* project [8]. The objective of the ibrow project was to create an intelligent brokering service for the World Wide Web. WIM was implemented to show how to build Cooperative Information Agents by using the Unified Problem-solving Method Language framework (UPML) [9]. UPML provides both a framework and a language to describes libraries of knowledge components and their relationships to form knowledge systems. Tasks in UPML describe *what* to do, problem-solving methods describe *how* to do it. UPML defines two types of problem-solving methods: *Problem Decomposers* and *Reasoning Resources*. Problem Decomposers decompose

a task into several subtasks and specifies the operational control structure over the subtasks. A Reasoning Resource solves a subtask that is specified by a Problem Decomposer. It specifies the assumptions on the domain knowledge to perform a primitive reasoning step. In WIM, tasks and problem-solving methods are used to make plans for how to solve problems. The plans are generated by *Librarian agents* on request from a *Personal Assistant* agent. The Personal Assistant agents form teams of co-operating Problem-solving agents based on the tasks that needs to be performed. Information needed to solve problems are provided by *Wrapper agents* that wrap information sources found on the World Wide Web.

3 Planning in a mobile environment

In this work plans are detailed descriptions of how to solve a problem – what tasks must be achieved in order to solve the problem, and whom are to achieve these tasks. UPML is used as a basis for describing the components of the plan. The tasks to be achieved are described in task structures. A task structure is a directed acyclic graph (DAG) where nodes corresponds to tasks. Tasks that cannot immediately be accomplished are divided into subtasks. Subtasks, again, are decomposed into smaller task until all leaf-node tasks of the task structure may be accomplished without further decomposition.

For a task to be considered a subtask of another task (the supertask), it must meet at least one of three requirements: *i*) the task must take the same input as its supertask, *ii*) the task must produce the same output as its supertask, *iii*) the task must connect two other tasks that are part of the supertask's set of subtasks. For a task to connect two tasks, it must take as input the output of one of the tasks it connect, and produce as output the input of the other task it connects. If two task are equivalent in input and output values, they are considered as the same task; thus two equivalent task cannot be in a subtask/supertask relationship.

A *set of subtasks* must meet all three requirements to be a valid decomposition. One, and only one, task must fulfil requirement *i*; only one other task must fulfil requirement *ii*; all other tasks must fulfil requirement *iii*. In the special case where only two subtasks exists, the two subtasks must be connected. By following these requirements and constructing the task structure in a goal-driven depth-first manner, unambiguous trees will be constructed.

However, a task structure alone does not make a plan. It is important to locate agents capable of achieving the tasks specified in the task structure. Therefore, tasks need to be mapped to capabilities of agents located in the system. An agent may advertise more than one capability as it may perform more than one action. Agents are capable of accomplishing a task if they have the capability of producing the same output from the same input and the same ontology as is specified by the task. Agent capabilities map to the UPML concept Reasoning Resources. Thus, both a task structure and agents with the capability to perform the leaf-node tasks are needed to specify a complete plan.

A plan is generated when a problem is encountered. The problem is mapped to a task, which becomes the root node of the task structure. The root node is then decomposed until all leaf-node tasks may be accomplished. Finally, agents capable of solving the leaf-node tasks are searched for. If capable agents are found, a plan for solving the problem is completely specified. Tasks are considered to be accomplished when all of its subtasks are accomplished. Thus, only leaf-nodes of a task structure

need to be accomplished for the root node of a task structure to be completed.

The above planning scheme enables planning in environments where a solution is based on services present in the system when encountering a problem. Such a scheme is capable of producing results that take into consideration the context of the situation. This context generally encompasses all available information in the environment. However, to limit the potential infinite amount of information available a model is imposed. Since the exact nature of this model is unimportant for this work, interested parties are directed to [10].

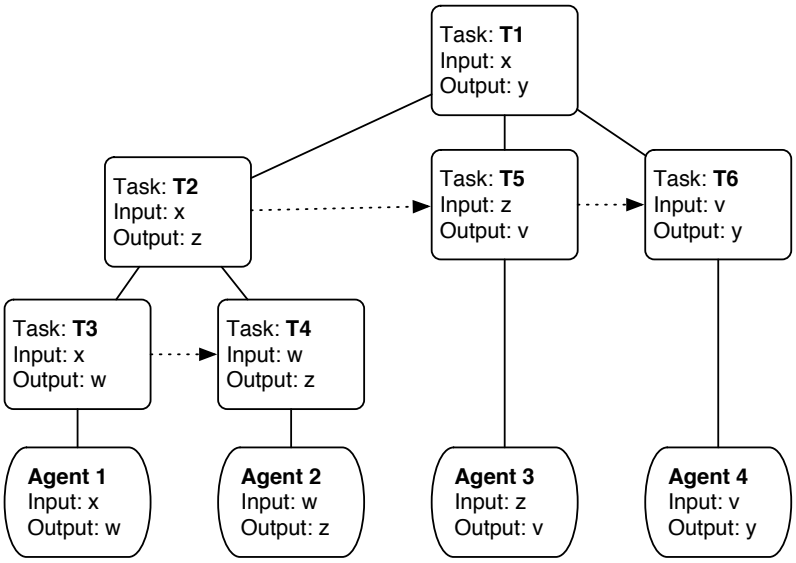

Figure 1: Example of a complete plan

An example of a plan having the qualities described in this section is illustrated in Figure 1. Task T1 is the root node of the task structure. This task is decomposed into a task structure consisting of six tasks including T1 itself. Task T2 takes as input the input of T1, and T6 produces the same output as T1. Thus, when T6 has been performed, the root node T1 is also considered performed, and so the problem is solved. The task T5 connects (illustrated by the dashed arrow) the two tasks T2 and T6 by taking the output of T2 as input and producing as output the input of T6. T2 is further decomposed into T3 and T4 as the precondition of T2 must be true before T2 may be achieved. The four agents Agent1 to Agent4 are mapped to tasks they are capable of solving.

In the initial implementation task structures are predefined in a task structure library and decomposition of a task is merely recursively extracting predefined subtasks. As the focus of this work has not been problem-solving methods, we have not implemented a fully UPML-compatible library. Hence, problem-solving methods specifying how a task structure is generated, have not been implemented.

4 Carrying out the plan

After a plan has been generated, it has to be carried out. Carrying out the plan is referred to as the problem-solving process. More than one agent may assist in the *problem-solving process*. Thus, the problems of task result sharing and task dependency extraction need to be solved. The current implementation is centralised, where one agent is responsible for the problem-solving process. This agent acts as a

project co-ordinator that resolves dependencies and distribute the tasks to the other agents. Agents that perform tasks are referred to as *Application agents*, while the agent responsible for the problem-solving process is the *Responsible agent*.

The dependencies between different tasks must be made explicit. This is accomplished by examining the input and output of the tasks being part of the task structure. As only leaf-nodes need to be achieved for the root node of the task structure to be achieved, only the dependencies between leaf-node tasks need to be examined and resolved. Tasks that require input from another task are defined as being dependent of another task. Dependent tasks must be executed after the tasks they depend on.

The execution of a plan with dependencies is done by sending batches of tasks for execution. A batch is a set of tasks that can be executed in parallel. Tasks located in the same batch cannot have dependencies among them, neither can they be dependent of tasks in a batch set to be executed after them. This arrangement resembles bucket sort, where a batch correspond to a bucket, and the batches are sorted by dependency relations. Hence, tasks with no dependencies are placed in the first batch. In the second batch only tasks that are dependent on the tasks in the first batch are placed. Then, the third batch contains tasks that are dependent of tasks in the second (and maybe also the first batch), and so on until there are no more leaf-node tasks left.

After the batches of tasks possible to perform in parallel are generated, all the tasks in the first batch are distributed simultaneously to the proper Application agents. It is the responsibility of the Responsible agent to distribute tasks to the correct Application agents. For an Application agent to be capable of solving its assigned task, it requires the input specified by the task. When a task description along with the input is received by an Application agent, the Application agent may initiate the assigned task. Application agents only receive task descriptions and input that are necessary to perform the tasks assigned to them.

The results achieved by Application agents are returned to the Responsible agent, which keeps track of what tasks produced what results. When all tasks in the first batch have been performed and results have been produced, the next batch is sent to the proper Application agents. The proper input is sent along with the request to perform the task. Batches of tasks are processed in this manner until all batches have been completed and an output equal to the output that the root node requires. When the output of the root node is produced, the root node is considered achieved, thus the problem has been solved.

5 The prototype

In this section the implemented prototype will be described. Below, an overview of the system architecture is given, followed by a description of the different agents types, finally the problem-solving process is explained.

System architecture

The architecture, as described in this work, was originally develop as part on the AmbieSense[1] research project. The main purpose of this architecture is to supply a generic architecture for implementing context-aware systems in an ambient

[1] www.ambiesense.net

environment. The three main pillars are: *Context Tags* (small, Bluetooth enabled computers), *Mobile Devices* and *Net-Based Information Services*.

A Context Tag supply information (contextual and other) to a mobile user from information providers. Contextual information can, as an example, be location. It can also distribute localised content, such as a menu from the restaurant where it is mounted, and other available devices in the surroundings.

The Mobile Device is divided into two main parts: the Context Middleware and the agency. The Context Middleware offers a general platform for aggregating and storing contextual information from diverse sources. It can furthermore distribute contextual information to other components wishing to utilise context in their service adaptation. The primary recipient of contextual information is the agency. The Mobile Device typically stores contextual information regarding the user's profile, e.g. goals as part of the task context.

The agency contains all the agents necessary to reason about user situations and handle the advanced services offered through goal identification and problem solving. For a more thorough description of the AmbieSense architecture see [11, 12, 13, 13].

Agency

The agency is implemented using the Jade [14] platform. A main parts of this system is the agency which contains the agents that facilitates the complex system adaptation. Beside the Jade specific agents, the agency contains two primary types of agents: the Core agents and the Application agents.

The Core agents are responsible for the basis services associated with the reasoning process. These agents are: the Context agent, the Creek agent and the Decomposer agent.

The Context agent is responsible for communicating with the Context Middleware. It responds to the contextual events triggered by the Context Middleware, and using the ontology translates them to the representation used inside the agency. It then notifies the agents subscribing to these changes; currently only the creek Agent.

The Creek agent utilises Case-Based Reasoning to asses the user's situation. Once a situation has been identified the goals associated with the particular situation is transferred to the Decomposer agent to be solved. Case-Based Reasoning [15] adapts to new situations by remembering similar past experiences (cases). Case-Based Reasoning is a particular promising method for assessing situations in context-aware systems [16, 17, 18]. The version of Case-Based Reasoning used and extended in this work is the Creek system [19, 20].

The Decomposer agent is responsible for satisfying the goals identified by the Creek agent, and is the main focus in this paper, as it is responsible for coordinating the problem-solving process. Hence, it corresponds to the Responsible agent discussed in Section 4.

Each of the Application agents are responsible for offering their own specific service. These services can range from the mundane, such as offering todays menu, to a complex service of suggesting suitable shops where the user would find interesting and good bargains. Every Application agent constructed must implement the correct FIPA[1] compliant communication protocol, and supply a specification of its capabilities in UPML.

[1] www.fipa.org

The problem-solving process

A sequence diagram describing the agent communication is shown in Figure 2 (the star indicating the possibility of more than one Application agent).

The Creek agent generates a problem specification when a situation that need to be resolved is encountered. A request is sent to the Decomposer agent to solve the specified problem, and along with it, the user context.

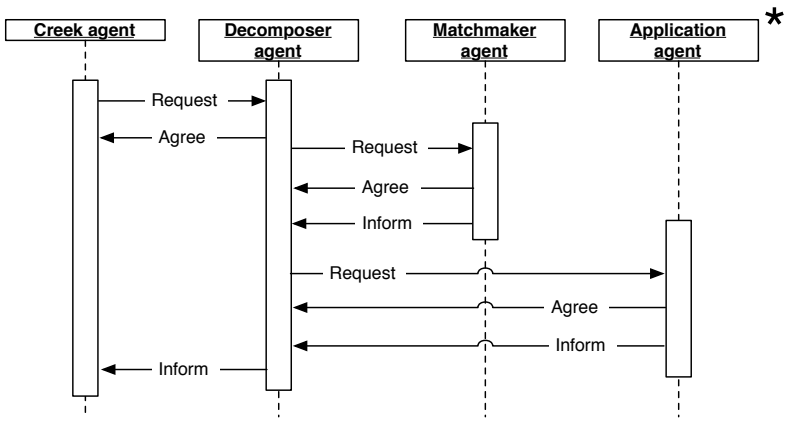

Figure 2: Agent communication

When receiving a request to solve a problem, the Decomposer agent search for a task that fulfils the goal specified in the problem specification. If such a task is found, the task is decomposed into a task structure. Application agents that are capable of accomplishing the leaf-node tasks of the task structure have to be found and mapped to the task structure in order to generate a complete plan. This is done by querying a matchmaker agent (the Directory Facilitator agent in Jade) whether any such agents are present in the current environment. A set of queries is sent to the matchmaker, one query for each leaf-node task that needs to be accomplished. The content of the query is a description of the leaf-node tasks. Application agents must register their capabilities with the matchmaker, and their capabilities must be described in the same manner as the tasks they are capable of performing. Agents that do not register their capabilities, or register capabilities that do not comply with the UPML description of a reasoning resource, will not get any requests to perform tasks.

For each query, a list of capable agents is returned by the matchmaker, and for each leaf-node task, one application agent is chosen by the Decomposer agent. In the case where all leaf-node tasks have been mapped with capable agents, a plan for how to solve the problem has been generated. Before the actual problem-solving can begin, the batches of tasks which is performed in parallel is generated. After generating the batches, the proper agents will be requested to perform the tasks. If an Application agent fails to accomplish a task, another agent is selected to solve the problem. The problem cannot be solved if no other capable agents are present. A new batch of tasks is not scheduled for execution until all tasks in the previous batch is performed and the results are received by the Decomposer agent. The Decomposer agent shares results with the proper Application agents. When an solution to the root node in the task structure is achieved or a failure has been encountered, a reply is sent to the Creek agent. The reply may be an *inform* or a *failure* message depending of the result the Decomposer agent has achieved.

6 An example

The following section describes a execution of our implemented prototype. The test run illustrates what happens when the Creek agent identify a situation of a *hungry user*. Being hungry is not a preferred situation over being satisfied. Thus, a problem is encountered and a problem specification that specify the preferred state is defined. A request for finding a solution of how to get to the state where the user no longer is hungry is sent to the Decomposer agent. The message is quoted in Figure 3.

```
(DecomposeProblem
  :ID "1098699115531"
  :Goal "Feed Hungry User"
  :UserContext (UserContext
    :SocialContext (SocialContext :Role Tourist)
    :TaskContext (TaskContext)
    :PersonalContext (PersonalContext
      :MentalContext (MentalContext
        :ShoppingPreferance Gifts
        :EatingPreferance Fast Food
        :LanguagePreferance English)
      :PhysiologicalContext (PhysiologicalContext))
    :EnvironmentalContext (EnvironmentalContext
      :Service Tickets
      :AirlineService SAS
      :TaxfreeService Wine)
    :SpatioTemporalContext (SpatioTemporalContext
      :Location Lounge
      :Destination London))
```

Figure 3: Content of the message sent to the Decomposer agent

The Decomposer agent identifies this request as valid and answers with an *agree* message before a solution is pursued. The `Feed Hungry User` goal is matched against the tasks in the library and a matching task is found. The matching task called `Get Food` is decomposed and a task structure is generated. However, the plan is not complete until agents capable of solving the leaf-node tasks of the task structure are found. Two messages, one for each task, which specify what tasks need to performed, are sent to the matchmaker. The matchmaker answers each message with a message containing all capable agents. In this case, the `Find Restaurants` task is answered with a message specifying the Find agent, and the answer to which agents capable of performing the task `Match to Context` is the Match agent.

The two agents Find and Match are mapped onto the correct tasks in the task structure, and as all the leaf-nodes have a corresponding agent, a complete plan has been generated. The complete plan is illustrated in Figure 4.

Next, any dependencies that exist between the tasks in the task structure is found. The dependencies are resolved by sorting tasks in batches of tasks that may be accomplished in parallel. As the task structure described in this example is rather simple, the result is that the two tasks are put into two batches – one for each task. The two tasks may not be performed in parallel as `Match to Context` requires the output of `Find Restaurants` as input. The decomposition is valid as the set of subtasks fulfil the three requirements mentioned in Section 3. `Find Restaurants` takes as input the input of the super task `Get Food`, and `Match To`

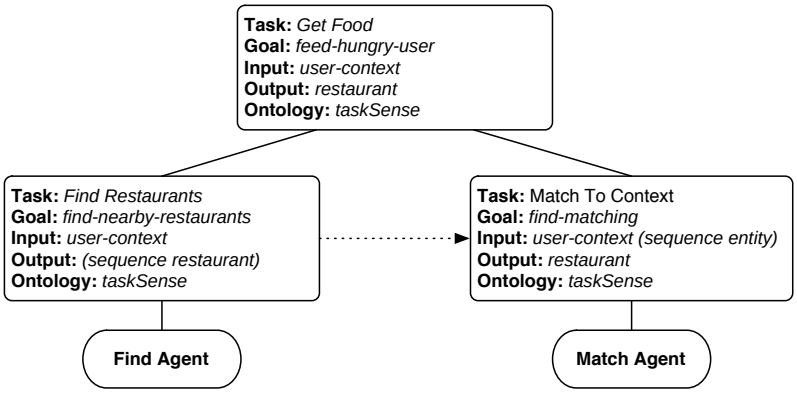

Figure 4: Plan for solving the task Get Food

`Context` produces the same output as the super-task. Also, `Find Restaurants` and `Match To Context` are connected as `Find Restaurants` produce the input of `Match To Context` as its output.

Requests to perform the `Find Restaurant` task is sent to the Find agent. The content slot of the request sent to the Find agent is reproduced in Figure 5.

```
(Find Resturant
  :inputrole (sequence (Location
  :InternalLocation Lounge))
  :outputrole (sequence Restaurant))))
```

Figure 5: Content of the message send to the Find agent

The request sent to the Find agent state that the Decomposer desire the Find agent to perform an action called `Find Restaurant`. The Find agent requires a location as input to perform this action, which result in a list of restaurants. The Decomposer specifies that the Find agent should narrow its search for restaurants located near the lounge.

After receiving the request to perform the `Find Restaurant` task, the Find agent answers with an *agree* message. Then, a search is performed, and an answer containing all restaurants that fulfil the location requirement is generated. The two restaurants *"Food 'R' Us"* and *"Eating 'R' U"* fulfil the location requirement. This is specified in the answer sent by the Find agent to the Decomposer agent.

A conversation similar to the one described above is carried out between the Decomposer agent and the Match agent after the result of the `Find Restaurants` task is received from the Find agent. The Match agent matches the restaurant list with the context and finds out that the restaurant that match the user context best is the one called *"Food 'R' Us"*. When the Decomposer receives the result, it establish the fact that the root node of the task structure has been accomplished, and that a solution is found. The proposed solution is sent to the Creek agent as an *inform* message. The final message send to the Creek agent is shown in Figure 6.

7 Conclusion and future work

The work presented here demonstrates by implementation, an architecture that allows for dynamic decomposition of problems based on available resources. As this

```
(DecomposeProblem
  :ID "1098699115531"
  :Goal "Feed Hungry User"
  :UserContext (UserContext
    :SocialContext (SocialContext :Role Tourist)
    :TaskContext (TaskContext)
    :PersonalContext (PersonalContext
      :MentalContext (MentalContext
        :ShoppingPreferance Gifts
        :EatingPreferance Fast Food
        :LanguagePreferance English)
      :PhysiologicalContext (PhysiologicalContext))
    :EnvironmentalContext (EnvironmentalContext
      :Service Tickets
      :AirlineService SAS
      :TaxfreeService Wine)
    :SpatioTemporalContext (SpatioTemporalContext
      :Location Lounge
      :Destination London))
  :Solution (Sequence
   (Restaurant :name "Food 'R' Us" :type Fast Food :location Lounge)))
```

Figure 6: Result message

is ongoing research, the work has some shortcomings.

No empirical tests has been conducted yet. However, work has been initiated to implement this architecture to health care situations, such as ward rounds, where the reasoning capabilities of the system is to be thoroughly tested. This work will also use Activity Theory to model activities in health care, thus aiding us in constructing plans and Application Agents [21].

Parts of the UPML standard have not yet been implemented. As the goal of the work reported in this paper was to get a fully functional prototype up and running, conforming fully with the UPML standard was not considered important. However, it is our goal to have, within a foreseeable future a fully UPML compatible problem-solving method based system implemented.

The system described in this paper will not solve a problem if no agents are capable of performing one or more of the leaf-node tasks in the task structure. No alternative solution will be sought after, and the result will be failure. In the version of this system that we are currently working on, task allocation will be done in a combined top-down and bottom-up approach. This approach enables plans to change dynamically in run-time. Thus, the system will try to work out alternative solutions to the specified problem. The work will be based on [22].

Acknowledgements

Part of this work was carried out in the AmbieSense project, which was supported by the EU commission (IST-2001-34244).

References

[1] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[2] Allan Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.

[3] Katia Sycara and Dajun Zeng. Coordination of multiple intelligent software agents. *International Journal of Cooperative Information Systems*, 1996.

[4] Katia Sycara. In-context information management through adaptive collaboration of intelligent agents. In Matthias Klush, editor, *In-Context Information Management Through Adaptive Collaboration of Intelligent Agents*. 1999.

[5] Glenn Jayaputera, Oshadi Alahakoon, Lito Perez Cruz, Seng Wai Loke, and Arkady B. Zaslavsky. Assembling agents on-demand for pervasive wireless services. In Qusay H. Mahmoud, editor, *Wireless Information Systems*. ICEIS Press, 2003.

[6] Glenn Jayaputera, Seng Loke, and Arkady Zaslavsky. Mission impossible? automatically assembling agents from high-level task descriptions. In *IEEE/WIC International Conference on Intelligent Agent Technology*, october 2003.

[7] Mario Gomez, Enric Plaza, and Chema Abasolo. Problem-solving methods and cooperative information agents. *International Journal of Cooperative Information Systems*, 11(3 and 4):329–354, 2002.

[8] B. J. Weilinga, C. van Aart, A. A. Anjewierden, and W. N. H. Jansweijer. IBROW Final Report. Deliverable 19, IBROW IST-1999-19005, 2003.

[9] Dieter Fensel, Enrico Motta, V. Richard Benjamins, Monica Crubezy, Stefan Decker, Mauro Gaspari, Rix Groenboom, William Grosso, Frank van Harmelen, Mark Musen, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The unified problem-solving method development language upml. *Knowledge and Information Systems*, 5(1), 2003.

[10] Anders Kofod-Petersen and Marius Mikalsen. Context: Representation and Reasoning – Representing and Reasoning about Context in a Mobile Environment. *Revue d'Intelligence Artificielle*, 19(3):479–498, 2005.

[11] Hans Inge Myrhaug, Nik Whitehead, Ayse Göker, Tor Erlend Fægri, and Till Christopher Lech. AmbieSense – A System and Reference Architecture for Personalised Context-Sensitive Information Services for Mobile Users. In Panos Markopoulos, Berry Eggen, Emile Aarts, and James L. Crowley, editors, *Ambient Intelligence: Second European Symposium on Ambient Intelligence, EUSAI 2004*, volume 3295 of *Lecture Notes in Computer Science*, pages 327–338. Springer Verlag, 2004.

[12] Anders Kofod-Petersen and Marius Mikalsen. An Architecture Supporting implementation of Context-Aware Services. In Patrik Floréen, Greger Lindén, Tiina Niklander, and Kimmo Raatikainen, editors, *Workshop on Context Awareness for Proactive Systems (CAPS 2005), Helsinki, Finland*, pages 31–42. HIIT Publications, June 2005.

[13] Till C. Lech and Leendert W. M. Wienhofen. AmbieAgents: A Scalable Infrastructure for Mobile and Context-Aware Information Services. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 625–631, New York, NY, USA, 2005. ACM Press.

[14] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade - a fipa-compliant agent framework. Technical report, Centro Studi e Laboratori Telecomunicazioni, 1999. Part of this report has been also published in Proceedings of PAAM'99, London, April 1999, pp. 97-108.

[15] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[16] A. Zimmermann. Context-awareness in user modelling: Requirements analysis for a case-based reasoning application. In Kevin D. Ashley and Derek G. Bridge, editors, *ICCBR 2003, Case-Based Reasoning Research and Development*, number 2689 in Lecture Notes in Artificial Intelligence, pages 718–732. Springer-Verlag, 2003.

[17] Oh Byung Kwon and Norman Sadeh. Applying case-based reasoning and multi-agent intelligent system to context-aware comparative shopping. *Decision Support Systems*, 37(2):199–213, May 2004.

[18] Enric Plaza and Josep-Lluís Arcos. Context-aware personal information agents. In *Cooperative Information Agents V*, number 2128 in Lecture Notes in Artificial Intelligence, pages 44–55. Springer Verlag, 2001.

[19] Agnar Aamodt. *A knowledge-intensive, integrated approach to problem solving and sustained learning.* PhD thesis, University of Trondheim, Norwegian Institute of Technology, Department of Computer Science, May 1991. University Microfilms PUB 92-08460.

[20] Agnar Aamodt. Knowledge-intensive case-based reasoning in creek. In Peter Funk and Pedro A. Gonzalez Calero, editors, *Advances in case-based reasoning, 7th European Conference, ECCBR 2004, Proceedings*, pages 1–15, 2004.

[21] Anders Kofod-Petersen and Jörg Cassens. Activity Theory and Context-Awareness. In Stefan Schulza, David B. Leake, and Thomas R. Roth-Berghofer, editors, *Proceedings of the IJCAI-05 Workshop on Modeling and Retrieval of Context (MRC 2005)*, volume 146, pages 1–12. CEUR Workshop Proceedings, July 31 - August 1 2005.

[22] Pinar Öztürk and Odd Erik Gundersen. A Combined Top-Down and Bottom-up Approach to Integrated Task-Decomposition and Allocation. In *Proceedings to The Third International Conference on Machine Learning and Cybernetics*, 2004.