NTNU
Innovation and Creativity

A Shared Memory Structure for Cooperative Problem Solving

Kari Røssland

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Pinar Öztürk, IDI

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

Problem Description

CoPS is a multiagent framework for cooperative problem solving, and the purpose of this framework is to ease the implementation of multiagent systems. The aim of this master thesis is to model and implement the part of the CoPS framework facilitating the teamwork. Teamwork will be performed through a shared communication medium, or a blackboard like structure, where the agents add information and partial results that might be interesting for other agents. Rules might be used to decide when, what and for whom certain pieces of information should be communicated. Before this work can start, the existing prototype must be slightly modified due to some weaknesses and errors. Tools that should be used are:
*Jess – a java based rule engine
*JADE – a java implemented software framework simplifying the implementation
of multiagent systems
*jCreek - a java implmeneted, flexible, frame-based
knowledge representation language

Assignment given: 20. January 2006
Supervisor: Pinar Öztürk, IDI

Abstract

CoPS is a FIPA compliant multiagent framework for cooperative distributed problem solving. The purpose of this framework is to ease the implementation of cooperative problem solving agents. Agents are autonomous software modules acting in an environment. A multiagent system is an environment inhabited by several agents. Agents in a multiagent system are designed to perform certain tasks that will fulfill the overall goal of the system. When such a task is too complicated for an agent to solve by itself, it has to be decomposed and divided on a team of agents solving the overall task cooperatively.

Work with CoPS was initiated by Gundersen at NTNU in 2003 [34]. The focus of his work was the decomposition of tasks and formation of a problem solving team. An architecture was designed and a prototype of the CoPS framework was implemented. Autumn 2005, we (read Kari Røssland and Pinar Öztürk) did a project where the main objective was to use the CoPS framework for a medical domain application. During this work, some problems about the CoPS framework were uncovered.

The main motivation behind this master thesis is to extend the pre-existing CoPS framework with a shared memory structure that assists the team of agents who are cooperating to solve a task. The task is decomposed into a three, showing the hierarchy and precedence of subtasks; representing a plan for solving the problem. The shared memory structure is supposed to execute this plan on an abstract level, by coordinating a team of agents in executing the plan on the concrete level.

In our approach to integrate a shared memory structure with the pre-existing CoPS framework, we first corrected the problems that were discovered in our 2005-project. Second, we modeled the CoPS framework with a black box - shared memory structure, to specify the requirements for the shared memory structure and to uncover which modifications had to be done to the pre-existing CoPS components. Third, an architecture for the shared memory structure, named the TEAM SPACE, was modeled. Fourth, the modifications to the pre-existing CoPS components and the TEAM SPACE were implemented in the CoPS framework prototype. Finally, part of the medical domain application model from the 2005-project was implemented using the CoPS framework prototype.

The contribution of this thesis is a framework architecture for cooperative distributed problem solving in multiagent systems using a shared memory structure. Our shared memory structure, the TEAM SPACE, coordinates the problem solving process that is based on a plan in form of a hierarchy of decomposed tasks.

Contents

1 Introduction **1**
1.1 Motivation and Objectives 1
1.2 Approach 4
1.3 Structure 4
1.4 Summary 5

2 Multiagent Systems and Shared Memory Structures **6**
2.1 Intelligent Agents 6
2.2 Multiagent Systems 7
2.2.1 Motivations for Multiagent Systems 7
2.2.2 Issues in Multiagent Systems 8
2.2.3 Cooperative Distributed Problem Solving 12
2.3 Shared Memory Structures 14
2.3.1 Blackboard Systems 15
2.3.2 The Publish/Subscribe Model 16
2.4 Summary 17

3 An Architecture for Cooperative Problem Solving: CoPS **19**
3.1 The overall CoPS System 20
3.2 A three-layered CoPS Architecture 22
3.2.1 Problem Solving Knowledge 22
3.2.2 Agents 25
3.2.3 Problem Solving Process 26
3.3 An Example 29
3.4 Current work with CoPS 33
3.5 Corrections and Extensions to CoPS 33
3.5.1 Corrections 33
3.5.2 Extensions 34
3.6 Summary 36

4 The TEAM SPACE Architecture **37**
4.1 The Core TEAM SPACE Architecture 37
4.1.1 TEAM SPACE Interactions and Interfaces 38
4.1.2 TEAM SPACE Agent 41
4.1.3 TEAM SPACE Structure Components 44
4.1.4 TEAM SPACE (TS) Structure Functions 47
4.2 An Example 55
4.3 TEAM SPACE Architecture Additions 60

4.3.1 Parallel TEAM SPACEs 60
4.3.2 Failure Handling 64
4.3.3 Dynamics and Re-Planning 65
4.4 Other Extensions to the CoPS Architecture 66
4.5 How the TEAM SPACE Architecture Meets the Functional Requirements 72
4.6 Summary 75

5 Comparison of CoPS to Relevant Work 76
5.1 Criteria for the Comparison 76
5.2 Relevant work - Systems Using Shared Repositories 77
5.2.1 Application of a Blackboard Framework to a Cooperative Fixture Design System 78
5.2.2 A Blackboard-Based Multiagent System for Supporting Concurrent Engineering Projects 82
5.2.3 A Blackboard Used for Collaborative Development of Interactive Robot 85
5.2.4 MAPSEC: Mobile-Agent Based Publish/Subscribe Platform for Electronic Commerce 88
5.3 The Comparison to CoPS 91
5.3.1 Representation 92
5.3.2 Awareness 94
5.3.3 Investigation 95
5.3.4 Interaction 96
5.3.5 Integration 97
5.3.6 Coordination 99
5.4 Summary 99

6 Implementation Details 101
6.1 Implementation Overview 101
6.2 Implementation Tools and The Pre-Existing CoPS Prototype . . 103
6.2.1 JADE and CoPS Agents 103
6.2.2 jCreek and TMST 106
6.2.3 Jess 107
6.3 Corrections 109
6.4 CoPS Problem Solving Process Extensions 110
6.4.1 The Modified CoPSTaskResponsible 111
6.4.2 The Modified CoPSProblemSolver 115
6.4.3 The Modified TMST 117
6.5 The TEAM SPACE 119
6.5.1 The TEAM SPACE agent 119
6.5.2 The TEAM SPACE structure 124
6.6 Summary 128

7 Experimentation and Results 129
7.1 How to Implement an Application Using the CoPS Framework . 129
7.2 The Checkup Example Application 130
7.2.1 Implementation of the Checkup Ontology 130
7.2.2 Implementation of the Checkup TMST 134
7.2.3 Implementation of the Checkup Agents 136

7.3 Test-Run of the Checkup Example Application 139
7.3.1 Solving a single problem 139
7.3.2 Solving of two problems in parallel 148
7.4 Test-Run Results 152
7.4.1 Corrections 153
7.4.2 Extensions Pre-Existing CoPS components 154
7.4.3 The Main Extension to CoPS: TEAM SPACE 154
7.5 Summary 155

8 Conclusions and Future Work 156
8.1 Discussion and Conclusions 156
8.1.1 The Current CoPS Framework 156
8.1.2 Accomplished Objectives 157
8.1.3 Arguments for Our TEAM SPACE Approach 158
8.1.4 A Critical View of CoPS 159
8.2 Future Work 160
8.3 Summary 161

A Content of the Enclosed Zip-File 162
A.1 The *javadoc* folder 162
A.2 The *source_code* folder 162
A.3 The *checkup_example* folder 162

B How to Run the Checkup Example Application 164
B.1 Running the Checkup Example Application in UNIX 164
B.2 Running the Checkup Example Application in Windows 166

List of Figures

1.1 Using CoPS in a system for performing checkups at a health center. 2
1.2 A decomposition of the *Take bloodtest* task, into subtasks. The arrows between the subtasks indicate I/O dependencies. 2

2.1 An agent in its environment taking as input its perceptions and producing as output its actions, modified from [26]. 7
2.2 The school-example ontology - a hierarchy of classes where all artifacts of the world is a subclass of *Entity*. 9
2.3 A taxonomy of some of the different ways in which agents can coordinate their behaviour and activities, modified from [36]. . . 11
2.4 The architecture of a basic blackboard system, showing the blackboard, knowledge sources or agents, and control components, taken from [36]. 16
2.5 A simple object-based publish/subscribe system, consisting of publishers, subscribers and a central event service, taken from [11]. 17

3.1 The overall CoPS system view. The CoPS system has four types of agents: Task Responsible (TR), Decomposer (DEC), Problem Solver (PS), and Personal Assistant (PA). Tasks handled by CoPS are decomposed into a TMST (Task Method Sub-Task tree). Problem solving and integration of partial results are supported by a shared memory - the TEAM SPACE. 20
3.2 A three - layered CoPS Architecture. The layers are: PROBLEM SOLVING PROCESS, AGENTS and PROBLEM SOLVING KNOWLEDGE. The agents abbreviations are: Task Responsible (TR), Decomposer (DEC), Personal Assistant (PA), and Problem Solver (PS). The shaded shapes reflect the focus of the current work with CoPS. 22
3.3 The nodes part of the TMST (Problem Solving Method - PSM, task and action) and their attributes. Short explanations to the attributes are provided. Modified from [34]. 23
3.4 The TMST - the complete solution space for a task, and the (chosen) solution of the TMST (marked with grey nodes) - a specific solution for a task. Modified from [34]. 24
3.5 The FIPA Request Interaction Protocol, describing a conversation starting with a request - message being sent from an Initiator to a Participant. Taken from [13]. 27

3.6 The FIPA Contract Net Interaction Protocol, describing a conversation starting with a call-for-proposal (cfp) message being sent from an Initiator to a Participant. Taken from [13]. 28
3.7 The TMST returned by the Decomposer (DEC) when the task, *(Task :name takeLaboratoryTest :input (Patient :name Paul))*, is received. 30
3.8 The TMST showing the solution, marked with shaded nodes. The lowest cost suggested for actions or PSMs propagates upwards in the tree-structure. Actions are connected to executors (Problem Solvers, PSs)- that has offered to execute them for the price attached to the corresponding action. 32
3.9 Parts of the overall CoPS system view depicted in figure 3.1, interacting with the TEAM SPACE. Agents interaction with the TEAM SPACE are Task Responsible (TR) and a group of Problem Solvers (PSs). 35

4.1 The Team Space Interface shows how Task Responsible (TR) and Problem Solvers (PSs) interact with the TEAM SPACE (TS) Agent, and how the TEAM SPACE (TS) Agent interacts with the TEAM SPACE (TS) Structure. 38
4.2 A protocol for collaborative work in CoPS. The protocol defines the messages exchanged between the TS Agent, PSs (Problem Solvers) and TR (Task Responsible) during problem solving in the TEAM SPACE. 41
4.3 The general agent architecture used by all of the CoPS agents, and also by the TEAM SPACE (TS) Agent. The components of the architecture are represented by boxes, and the interactions between them with arrows. 42
4.4 The TEAM SPACE (TS) Agent process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of the TS Agent. Here we suppose that the TS Agent only handles one TS Structure at a time. 43
4.5 The architecture of the TEAM SPACE (TS) Structure. Rule Base Container keeps and manipulates a set of rules. Result Library and Goal Stack keeps and manipulates facts. Plan Library provides information stored in the TMST. TEAM SPACE (TS) Problem Solving State, represents a view of the facts stored in the Result Library and the Goal Stack. 45
4.6 The TMST showing the solution, marked with shaded nodes. The lowest cost suggested for actions or PSMs propagates upwards in the tree-structure. Actions are connected to executors (Problem Solvers, PSs)- that has offered to execute them for the price attached to the corresponding action. 56
4.7 Parallel problem solving in the TEAM SPACE with a single TEAM SPACE (TS) Agent. The TS Agent has references to a set of TEAM SPACE (TS) Structures, and it interacts with several problem solving teams at once. The problem solving teams are comprised of Problem Solvers (PSs) and a Task Responsible (TR). 61

4.8 Parallel problem solving in the TEAM SPACE with multiple TEAM SPACE (TS) Agents. Each TS Agent only has reference to one TEAM SPACE (TS) Structure, and it interacts with one problem solving team at a time. The problem solving team is comprised of Problem Solvers (PSs) and a Task Responsible (TR). 62
4.9 The TEAM SPACE (TS) Agent process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of the TS Agent. Here we suppose that the TS Agent may handle several TS Structures at a time. 63
4.10 The **former version of the Problem Solver (PS)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of PS. Taken from [34]. 67
4.11 The **new version of Problem solver (PS)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR. 68
4.12 The **former version of Task Responsible (TR)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR. Taken from [34]. 70
4.13 The **new version of Task Responsible (TR)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR. 71

5.1 The architecture of the cooperative fixture design system, modified from [28]. 78
5.2 Hierarchical structure of the blackboard, from [28]. 79
5.3 Implementation of the system architecture in the GBB environment, from [28]. 81
5.4 The blackboard-based I2QFD, from [19]. 83
5.5 Agency system of I2QFD, from [19]. 83
5.6 Basic composition and basic design process, from [24]. 86
5.7 Combined blackboard structure with publish/subscribe model, taken from [24]. 87
5.8 Transactions between the servers, from [24]. 88
5.9 The architecture of MAPSEC, taken from [31]. 89
5.10 The MAPSEC buyer subsystem architecture, taken from [31]. . . 89
5.11 The MAPSEC supplier subsystem architecture, taken from [31]. . 90
5.12 The inner structure of the MAPSEC Broker, taken from [31]. . . 90

6.1 The three layered CoPS architecture, showing which parts are involved in our implementation. The agent abbreviations are: PA - Personal Assistant, TR - Task Responsible, DEC - Decomposer, PS - Problem Solver. 102
6.2 The class hierarchy of the CoPS agent implementation [34]. The shaded classes represent which agents that are actually implemented in the CoPS framework prototype. 104
6.3 The FileWriter class. Used by most of the main classes in the CoPS framework prototype, to logs different happenings. 111
6.4 A modified UML diagram of the CoPSTaskResponsible agent and its behaviours. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes. 112
6.5 A modified UML diagram of the CoPSProblemSolver agent and its behaviours. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes. 116
6.6 A modified UML diagram of the TMST main class. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes. 118
6.7 Our package structure for the TEAM SPACE implementation. *agent* contains TS Agent classes, and *structure* contains TS Structure classes. 119
6.8 An UML class diagram showing the classes that implements the TS Agent. These classes are contained in the package: *teamSpace.agent* 120
6.9 An UML class diagram showing the classes that implements the TS Structure. These classes are contained in the package: *teamSpace.structure* 125
6.10 Classes representing some of the TMST-units. These are used by the other classes implementing the TS Structure and the TS Agent. These classes are contained in the package: *teamSpace.structure* 128

7.1 The implemented part of the Checkup TMST. The initial task *Do checkup* is decomposed into a hierarchy of PSMs, actions, and tasks. The Checkup TMST has a total of 20 different actions. 131
7.2 The class diagram of the checkup ontology. The ontology describes concepts used by agents when solving a checkup problem. 132
7.3 Class diagram for the agents implemented to solve the *Do checkup* task. 136
7.4 A proposal message encoded in FIPA-SL, using the Checkup Ontology. 138
7.5 Action (action - type) - agent tuples that are a returned by the *Matchmaker* (DF). 141

7.6 Result from the *TR's* handling of proposals. *Executors* (*TMST* node) are named after PSs, and related to a *TMST Action* and a *Cost*. 142
7.7 Result from the solution generation. The figure lists the different nodes part of the solution and their final costs. The cost of the initial *Task* is approximately *31.6*. 142
7.8 A list of the agents receiving accept-proposal and refuse-proposal messages, after the solution of the *TMST* is generated. 143
7.9 Agent interactions in the problem solving process steps: *solving of subtasks* and *integration of partial results*. 144
7.10 The first problem solving state, generated after executing the rules of *RuleBase* for the first time. 145
7.11 The second problem solving state, generated after executing the rules of *RuleBase* for the second time. 146
7.12 A list of the agents receiving accept-proposal messages from *TR1*, after the solution of the *TMST* is generated. 150
7.13 A list of the agents receiving accept-proposal messages from *TR2*, after the solution of the *TMST* is generated. 150
7.14 A segment from TR1.txt showing that *TR1* requests *TSAgent* to initialize a TS structure, and specialize it for a task with a certain input and teamID. 150
7.15 A segment from TR2.txt showing that *TR2* requests *TSAgent* to initialize a TS structure, and specialize it for a task with a certain input and teamID. 151
7.16 A list of the agents receiving accept-proposal messages from *TR1*, after the solution of the *TMST* is generated. 152
7.17 A list of the agents receiving accept-proposal messages from *TR2*, after the solution of the *TMST* is generated. 152
7.18 A segment from TR1.txt showing that *TR1* requests *TSAgent1* to initialize a TS structure, and specialize it for a task with a certain input and teamID. 152
7.19 A segment from TR2.txt showing that *TR2* requests *TSAgent2* to initialize a TS structure, and specialize it for a task with a certain input and teamID. 153

B.1 The starting window of the Jade GUI. 165
B.2 This illustrations shows how to fill out the parameters, when starting a new agent in the Jade GUI. 165

List of Tables

3.1 The attributes related to the TMST-nodes in the TMST returned by the DEC when the task *(Task :name takeLaboratoryTest :input (Patient :name Paul)), is received.* 31

4.1 How the interactions in the figure 3.1 relate to the interactions in figure 4.1 39

4.2 Information needed from each of the nodes (action, PSM, task) in the TMST to make the rules in the Rule Base Container. . . . 50

Chapter 1

Introduction

CoPS is a FIPA compliant multiagent framework for cooperative distributed problem solving, and the purpose of this framework is to ease the implementation of cooperative problem solving agents. Agents are autonomous software modules acting in an environment. A multiagent system is an environment inhabited by several agents. Agents in a multiagent system are designed to perform certain tasks that will fulfill the overall goal of the system. When such a task is too complicated for an agent to solve by itself, it has to be decomposed and divided on a team of agents solving the overall task cooperatively.

This chapter gives an introduction to the master thesis, which main objective is to develop a shared memory structure for the CoPS framework. In section 1.1 the motivations and objectives for our work is stated. Section 1.2 outlines our approach to reach the objectives. And finally, the structure of this document is described in section 1.3.

1.1 Motivation and Objectives

Work with CoPS was initiated by Gundersen at NTNU in 2003 [34]. The focus of his work was the decomposition of tasks and formation of a problem solving team. An architecture was designed and a prototype of the CoPS framework was implemented. Autumn 2005, we (read Kari Røssland and Pinar Öztürk) did a project where the main objective was to use the CoPS framework for a medical domain application. The *Medical Checkup System* that was modeled is illustrated in figure 1.1. Next, we use this figure to give a simplified example description of how CoPS works.

The *Medical Checkup System* is used by all of the persons working at the health center. Each person has its own specialized user interface, represented by a *Personal Assistant* agent. *Personal Assitant* agents also connect the users to problem solver agents in the system. The problem solver agents are experts on the users' fields of competence and assist their users in performing their daily activities. Some of the problem solver agents are connected to external resources like databases or software systems. The problem solver agents exchange information between them and engage in cooperative problem solving.

Imagine that a patient shows up at the reception of the health center, requesting a bloodtest. Then, the *Receptionist* checks with the system if there are

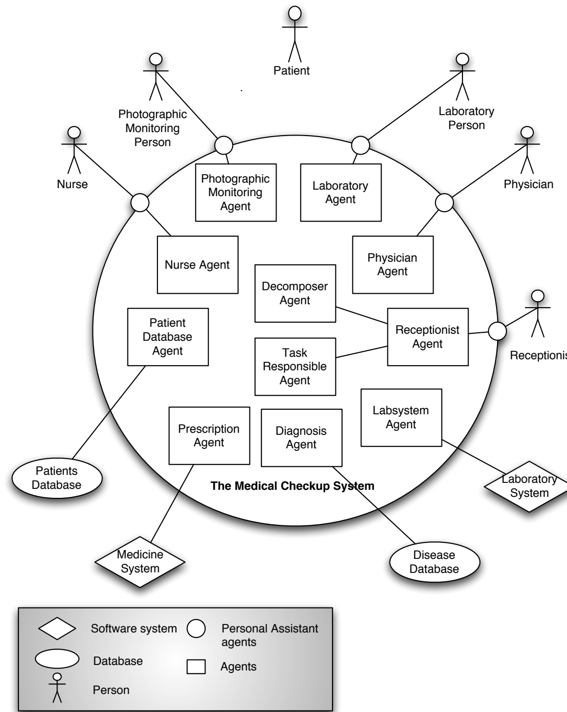

Figure 1.1: Using CoPS in a system for performing checkups at a health center.

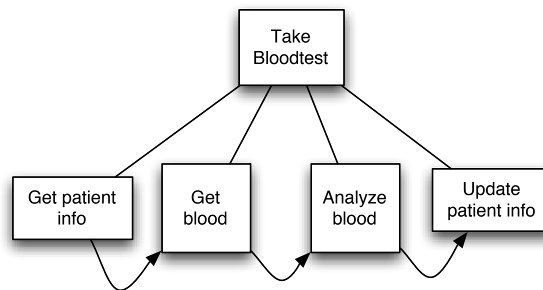

Figure 1.2: A decomposition of the *Take bloodtest* task, into subtasks. The arrows between the subtasks indicate I/O dependencies.

enough available resources. The *Receptionist Agent* asks the *Task Responsible Agent* if it is able to gather a team of problem solver agents to solve the task *Take bloodtest*. *Task Responsible Agent* asks the *Decomposer Agent* to decompose the task. The decomposition is illustrated in figure 1.2. When the *Task Responsible Agent* gets the decomposition from the *Decomposer Agent*, it forms a team of agents (some of them related to medical personnel) that are capable of solving the *Take bloodtest* - task together.

The team consists of *Patient Database Agent* performing *Get patient info* and *Update patient info*, *Laboratory Agent* performing *Get blood* and *Labsystem Agent* performing *Analyze Blood*. Problem solver agents decide for themselves if they will or can join the team dependent on their users' available capacity. When the team is formed, *Task Responsible Agent* informs the *Receptionist Agent* that there are available capacity to take the patient's bloodtest.

As one can see in figure 1.2, there are I/O dependencies between the different subtasks. Thus, there are dependencies between agents. This means, for example, that *Laboratory Agent* performing the task *Get blood*, cannot start solving the task before it knows the solution of *Get patient info*, produced by *Patient Database Agent*. The problem solver agents do not know about each other and for that reason they need to be coordinated.

In this master thesis we propose a shared memory structure for coordinating a team of agents engaged in solving a shared task. The shared memory structure uses the information from the task decomposition to make a set of rules that decide when the different tasks are ready to be performed. The structure also keeps a knowledge base where partial results are stored. When *Patient Database Agent* has performed *Get patient info* it stores the task's output (result) in the shared memory structure knowledge base. The rules then use the information in the knowledge base to infer that *Get blood* is ready to be performed. The input (result) needed to perform *Get blood* is sent to *Laboratory Agent* which then performs its task.

The main objectives of this master thesis are:

- Correct problems in the pre-existing CoPS framework.
- Modify the pre-existing CoPS framework, in such a way that a shared memory structure could be integrated.
- Model an architecture for a shared memory structure and integrate it with the pre-existing CoPS framework.

And the outputs from reaching those objectives are:

- CoPS framework architecture with a shared memory structure.
- CoPS framework prototype (API), supporting shared memory structures.
- An example application (part of the *Medical Checkup System*), implemented using the CoPS framework API.

1.2 Approach

In our approach to integrate a shared memory structure with the pre-existing CoPS framework, we first corrected the problems that were discovered in our 2005-project. Second, we modeled the CoPS framework with a black box - shared memory structure, to specify the requirements for the shared memory structure and to uncover which modifications had to be done to the pre-existing CoPS components. Third, an architecture for the shared memory structure, named the TEAM SPACE, was modeled. Fourth, the modifications to the pre-existing CoPS components and the TEAM SPACE was implemented in the CoPS framework prototype. Finally, part of the medical domain application model from the 2005-project was implemented using the CoPS framework prototype.

Different Java tools were used for the implementation. These tools also put some implications from modeling of the architecture. The tools are:

- Jess - a java based rule engine
- JADE - a java implemented software framework simplifying the implementation of multiagent systems
- jCreek - a java implmeneted, flexible, frame-based knowledge representation language

1.3 Structure

The approach for attaining the objectives is reflected in how this report is structured. The process has been iterative though. The further structure of the report is as follows:

- Chapter 2 introduces important concepts of intelligent agents, multiagent systems and shared memory structures.
- Chapter 3 gives an overall view of the CoPS architecture integrated with a black-box shared memory structure, called the TEAM SPACE. The pre-existing work with CoPS is described, and corrections and extensions to this work are proposed. Finally, functional requirements for the TEAM SPACE architecture are listed.
- Chapter 4 presents our proposition for the TEAM SPACE architecture. The TEAM SPACE is our main extension to CoPS.
- Chapter 5 gives a comparison of CoPS, with the integrated TEAM SPACE, to other relevant work. Relevant work are other systems using shared memory structures.
- Chapter 6 describes the implementation details. It includes a presentation of the different implementation tools and a description of how corrections and extensions are implemented in the CoPS framework prototype.
- Chapter 7 explains how the CoPS framework prototype was used to implement part of the *Medical Checkup System*. Results from the test-run of the application are also provided.

- Finally, chapter 8 concludes the results from our work, and outline future work.

1.4 Summary

In this chapter we have given an introduction to the work part of this master thesis. The introduction included a description of our motivation and objectives, our approach to achieve these objectives, and an overview of how this document is outlined.

Chapter 2

Multiagent Systems and Shared Memory Structures

A multiagent system (MAS) is inhabited by intelligent agents solving problems alone or by cooperating with other agents in their environment. A shared memory structure could be part of a multiagent system, and be used by the intelligent agents to cooperate and share information.

This chapter provide an introduction to important issues considering intelligent agents, multiagent systems and shared memory structures. Intelligent agents are shortly covered in section 2.1, multiagent systems and cooperative distributed problem solving (CDPS) are described in section 2.2, and finally some research areas focusing on shared memory structures are touched upon in section 2.3.

Content of this chapter serves as the theoretical background for the work with this master thesis, and will hopefully make it easier to read and understand the remaining chapters of this document.

2.1 Intelligent Agents

There are many definitions of agents. Both simple systems like thermostats and more complex systems like space probes can be seen as agents. *Intelligent agents* have more specific definitions and restrictions though. The definition to be used here is found in [36]:

> "An intelligent agent is one that is capable of *flexible autonomous* actions in some environment in order to meet its design objectives. Flexibility means three things: *Reactivity*, *pro-activeness* and *social ability*."

As pictured in figure 2.1, an agent takes as input its perceptions; information about the current environmental state, and produces as output; actions that affect its surroundings, and causes new environmental states. Agents use perceptions in different ways to decide which action to perform next through a decision making process.

According to [23], an agent capable of *autonomous behavior*, make judgments and perform actions independently and without control by others or by outside

Figure 2.1: An agent in its environment taking as input its perceptions and producing as output its actions, modified from [26].

forces. We can also say that the agent is self-directed. To explain this further we can compare it to objects in object-oriented programming. An object is told what to do, through method invocation made by other objects. An agent is asked to do something by accepting requests from other agents. Whether the request is fulfilled or not, is decided by the agent itself. The agent can also perform actions without being asked, which is not possible with objects [36].

Another difference between objects and agents is that agents are capable of *flexible behavior*; the standard object model do not mention this kind of behavior. Flexible behavior is composed of *reactivity*, *pro-activeness* and *social ability*. A reactive agent perceives its environment, and responds in a timely fashion to changes that occur in it in order to satisfy its design objectives. An agent's pro-active behavior describe the abilities of attending to goal-directed behavior by taking the initiative in order to satisfy their design objectives. Finally, social behavior is important for an agent to interact with other agents. Communication through exchange of information in cooperative or competitive settings are used as a mean by agents to achieve their goals and to satisfy their design objectives [36, 37, 12].

2.2 Multiagent Systems

Distributed Artificial Intelligence (DAI) is a subfield of AI, concerned with problem solving where agents solve tasks and subtasks [37]. The main areas of DAI are *multiagent systems* (MAS) and *cooperative distributed problem solving* (CDPS). MAS is concerned with the behavior of a collection of autonomous agents aiming to solve a given problem. How this problem can be divided among the agents and how the solution are cooperatively found through communicating and sharing information is considered by DPS.

2.2.1 Motivations for Multiagent Systems

There are several applications of multiagent systems in domains like medicine, biology, air traffic, urban traffic, telecommunications, economics, news, electronic commerce, computer games, enterprise management and design. Some of these applications were described in [33]. Problems solved by the different applications have one or more of these characteristics: *modular*, *decentralized*, *changeable*, *ill-structured* and *complex*. Motivations for using MAS are tightly connected to these problem characteristics [27].

First of all, agents are suited to applications that fall into natural modules,

since agents themselves are modular, like objects in object-oriented programming. Second, since an agent is autonomic and pro-active, it does not need to be invoked externally or be told to take action or which action to perform. When several such agents are distributed they can solve problems that are too large for a centralized agent and provide solutions which draw from distributed information sources, or where expertise is distributed. Third, the fact that agents are modular and pro-active make them especially valuable when a problem is likely to change frequently. Modularity permits the system to be modified one piece at a time, and decentralization minimizes the impact that changing one module has on the behavior of other modules. Fourth, agents offer a realistic approach to manage ill-structured, under-specified applications. Because when you develop agents, you do not have to consider a specific domain structure, but only the classes of entities that exist in that domain. Finally, when implementing agents one gives each of them a set of behaviors. At run-time, when different agents are working together by combining their behaviors, the total set of behaviors increase. In addition agents can combine simple behaviors to make more advanced ones. Rather complex systems may be implemented with a reduced amount of software and thus more cost-effective.

2.2.2 Issues in Multiagent Systems

To explain some important issues in multiagent systems we will here introduce a multiagent environment inhabited by intelligent agents. The domain of the environment is a school and the agents play roles as teachers and students. Especially important is an agent's social ability, as the agents cannot go around attempting to achieve their goals without taking others into account. And some goals can only be achieved through cooperation.

Knowledge Representation and Ontologies

Agents may keep some symbolic model of the environment. This model is not an exact copy of the environment. Since there are large amounts of entities and information, agents only partly models the world. *Environmental models* introduce two key problems [36]. First, the real world should be translated into an accurate adequate symbolic description, in time for that description to be useful. Then, it is the representation or reasoning problem; the problem of how to symbolically represent information about complex real-world entities and processes, and how to get agents to reason with this information in time for the results to be useful.

Knowledge is represented using a *knowledge representation language*. The fact that Paul and Karen are students and are located in a classroom, and that Paul is their teacher and is located in the teachers room can be represented as follows (this is just one of many ways to represent this information):

student (Paul)	teaching (Tom, Karen)
student (Karen)	teaching (Tom, Paul)
teacher (Tom)	inroom (Room1, Karen)
classroom (Room1)	inroom (Room1, Paul)
teachersroom (Room2)	inroom (Room2, Tom)

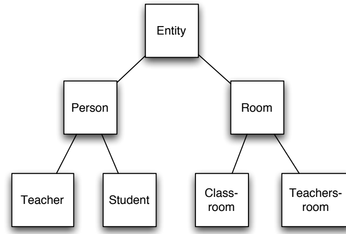

Figure 2.2: The school-example ontology - a hierarchy of classes where all artifacts of the world is a subclass of *Entity*.

A specification of all the *objects*, *concepts*, and the *relationships* in the school environment are represented in an *ontology*. The ontology is a vocabulary for the knowledge representation. A part of the school ontology, in a simplified version, can be as in figure 2.2. It shows a hierarchy of subclasses, where all artifacts in the world is a subclass of entity. Further on, both teachers and students are types of person. Each artifact can also have attributes, and these attributes will be inherited by its subclasses.

Communication and Communication Languages

For teachers and students to be able to work together, in either a cooperative or competitive way, they have to communicate. *Communication* between agents is a special *type of action*, like the actions they use to change the state of the environment [36]. A communicative action influence other agents by changing their state (beliefs and knowledge), rather then influencing the environment.

Teachers communicate knowledge to their students so that their knowledge bases grow, or faulty information are corrected or removed. The communication can appear through *message passing*, when a teacher lectures for his students in class, or through a *shared memory*, when the teacher writes something on the blackboard. A message can be *broadcasted*, or *sent directly* to an agent when the address is known. When an agent do not know who should receive a message, it may send it to a *facilitator* which forwards the message to the right receiver(s). The facilitator helps coordinating agents' activities and can satisfy requests on behalf of their subordinated agents. Other similar existing methods include mediators, brokers, matchmakers, yellow pages and blackboards [12].

Rules for the set-up of a message being sent are defined by a *communication protocol* [36]. The semantics of a communication protocol is domain independent, meaning that agents from a school domain may as well communicate with agents from a hospital domain, using similar communication protocols. The structure of a protocol may contain fields like *performative*, *sender*, *receiver*, *language*, *ontology* and *content*. To be able to interpret a message an agent has to speak the language submitted in the language field, and it has to know about the ontology. For example, when a student communicates with a doctor about some medical problem, the doctor knows about medical ontologies, but he would have to use a more common ontology for the student to understand what he is saying. The content of a message is a symbolic representation using

the structure of the language and the vocabulary of the ontology. From this we can draw the conclusion that a message is domain independent, while it's content is domain dependent.

Interaction and Interaction Protocols

Communication is a tool for *agent interaction*. There is no way to avoid interactions in multiagent systems. Just look at the school environment: Students and teachers are walking around talking to each other. By moving into a classroom, finding a chair, and sitting down, a student changes the environment. By changing the environment, the student interacts *indirectly* with the persons in that environment (classroom in this case) perceiving the changes. If the chair belongs to another person and this person comes up to the student starting arguing to get his chair back, they interact *directly*.

An interaction is basically a series of events or a dialogue between agents, with some final outcome. Therefore while a communication protocol specifies a single message, an *interaction protocol* specifies series of messages (the message flow) [36]. For multiagent systems to be efficient, the interaction among agents should serve some purpose. Like agents interacting to share resources or coordinate their actions.

Imagine a group of students working together on a project. First, they would have to decide upon a common goal for the project. Next, they must plan and organize the work. When a plan is worked out, and the total project is divided into smaller parts, the singular tasks should be divided among the students. Typically, a student will participate in those parts of the project that he/she is best at. After working on the project for a while, the students find out that they must make some changes, so they re-plan and divide the work once again. Finally, the project is finished by putting together and adapting the different parts. In this example, the interactions among students are crucial to complete the project.

Coordination: Cooperation and competition

A reason for agents to interact, is to *coordinate agent actions*. Coordination has been defined in [36] like...:

> "...The process by which an agent reasons about its local actions and the (anticipated) actions of others to try and ensure that the community acts in a coherent manner."

For agents to coordinate their actions, they have to know about how they depend on other agents. Then they can arrange their actions in a way that improves the performance or reduces conflict in a group of agents. As described in the figure 2.3, coordination can be either *cooperative* or *competitive*.

Whether to use cooperative or competitive coordination in a MAS mostly depends on the MAS being open or closed [14]. An *open MAS* can contain agents that are not designed to cooperate and coordinate at all. But, most open MASs are designed to assist the agents in working together. The most common kinds of these mechanisms are negotiations and auctions. An example of negotiation is when an agent barters services in exchange for assistance on a particular task or subtask. An example of an auction would be where you have

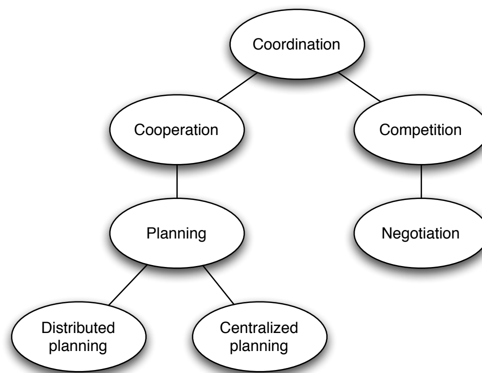

Figure 2.3: A taxonomy of some of the different ways in which agents can coordinate their behaviour and activities, modified from [36].

a group of tasks and agents that you would like to distribute as efficiently as possible. You would let the agents bid on the tasks they want to do. Assuming that the agents are configured correctly they would only bid on the tasks they can complete for less than the other agents.

Closed MASs contain well-behaved agents designed to cooperate easily towards a global goal. An agent is driven by goals, which they strive to fulfill. These goals may be connected to its own interests, or to the design objectives of this particular agent. But, in a MAS agents also attempt to fulfill the system's overall goal. A goal can be achieved by performing several tasks. When agents' goals or tasks are connected in some way, like information dependencies, sharing of resources, task redundancy or need of another agent's capabilities to accomplish a certain task, the agents will have to cooperate, or at least they will be better off doing it. The time to perform a set of tasks is reduced and the use of resources is improved. Again we go back to the student-project-example. The overall goal of the project is to produce good results and deliver in time. Participating students have goals connected to the tasks that they are going to perform. But while working, a student might recognize that he needs some information from what the others have been doing since he is going to write the summary chapter. Sandra comes up to Paul and tells him that they have to read some of the same articles, and therefore she could tell him what is important so that he does not have to read them also. This indicates that the results of the project are better and more consistent, and that the project-process is more efficient, when the students cooperate.

Coordination Mechanisms

There are several kinds of techniques or mechanisms that are used for agent coordination [26, 35, 18]. Some of them are described here.

Organizational structuring. Agents are organized into an organization. The organization is a pattern of information and control relationships between individuals and is responsible for shaping the types of interactions among the

agents. The organization may be based on how a task is decomposed. Agents use knowledge of the organization to determine with whom to communicate and how to prioritize tasks - it specifies which actions an agent will undertake. Organizational structures may be functional (based on skills), spatial (based on physical location) or temporal (based on time relationship).

Social norms and laws. A norm is an established, expected pattern of behaviour (to queue when buying cinema tickets). A social law is similar to norms, but carry some authority (traffic rules). Social laws in an agent system can be defined as a set of constraints, for example saying that when the environment is in some state, then a certain set of actions is forbidden.

Metal-level information exchange. Agents exchange control level information about current priorities and focus. The control level information influences agent's decisions. But, it does not specify which goals an agent will or will not consider, and might therefore be imprecise.

Partial global planning. The main principle is that cooperating agents exchange information in order to reach common conclusions about the problem solving process. The planning is partial because the system does not generate a plan for the entire problem. And it is global because agents form non-local plans by exchanging local plans and cooperating to achieve a non-local view of problem solving. Partial global planning involves 3 iterated stages: (1) each agent decides what its own goals are and generates short-term plans in order to achieve them, (2) agents exchange information to determine where plans and goals interact, (3) agents alter local plans in order to better coordinate their activities.

Multi-agent Planning. There are two basic approaches to multi-agent planning. The first one, centralized planning involving central coordination to identify interactions. The second one, distributed planning involving a group of agents cooperating to form a centralized or a distributed plan.

Mutual modelling. Agents build a model of the other agents - their beliefs and intentions, and coordinate own activities based on this model.

2.2.3 Cooperative Distributed Problem Solving

As mentioned before *cooperative distributed problem solving* (CDPS) is an area within DAI. A definition made by Durfee(1989)[37, 26] is:

> "CDPS studies how a loosely-coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities. Each problem-solving node in the network is capable of sophisticated problem-solving and can work independently, but the problems faced by the nodes cannot be completed without cooperation. Cooperation is necessary because no single node has sufficient expertise, resources, and information to solve a problem, and different nodes might have expertise for solving different parts of the problem"

Most work on CDPS has made the *benevolence* assumption: that the agents in a system implicitly share a common goal, and thus that there is no potential for conflict between them. This assumption greatly simplifies the designer's task. CDPS is described in [37, 26].

In CDPS tasks are known at design time, and the process of CDPS has four phases which work out from the given task or problem:

1. Problem decomposition
2. Subproblem distribution; task allocation
3. Subproblem solving/solution
4. Answer/solution synthesis

A *teamwork model based* on CDPS have a slightly different process from this one. One of the differences is that the tasks or goals used for cooperation are not known at design time:

1. Recognize the potential for cooperative action
2. Team formation: Find a group of agents that have a commitment to joint action
3. Plan formation: Agree upon course of action
4. Team action: Execute agreed plan of joint action

The different stages of these two problem solving processes can be solved by using a variety of methods. Different parts of the problem solving and methods used are explained next.

Problem decomposition and task allocation

First of all, the potential for *cooperative action* would have to be recognized. The agents find out that the overall system performance will be better with cooperative action. Then the overall problem to be solved is decomposed into smaller subproblems. The *problem decomposition* will typically be hierarchical, so that subproblems are further decomposed into smaller subproblems, and so on, until the subproblems are of an appropriate granularity to be solved by individual agents. The different levels of decomposition will often represent different levels of problem abstraction.

Both *centralized* and *distributed* decomposition is possible depending on the MAS-design. If the problem is decomposed by one individual agent, this assumes that this agent must have the appropriate expertise to do this. It must have knowledge of the task structure, that is, how the task is put together, and of agents that will eventually solve problems and their capabilities. In distributed decomposition, several agents have the necessary knowledge to make a problem decomposition. And some agents might have additional information to other agents, and therefore the decomposition itself may be better treated as a cooperative activity.

After the problem is decomposed one has to find agents capable of performing the leaf-node-actions. This part of the process can be called *subproblem distribution*, *task allocation* or *team formation*. Here we will use team formation as the process of finding and deploying capable agents for performing the different tasks. The chosen agents will work together and cooperate to achieve the common goal of solving the initial problem. How to gain the required information about agents, and on the basis of this recruit new team members, is the

core problem of the team formation process. The solution may depend on the cooperation techniques or cooperation protocols used, among other attributes.

The *contract net protocol* is a widely used standard for allocating tasks in multiagent systems. In the contract net protocol there are a network of agents or nodes. The collection of nodes is the "contract net". Each node in the network can play "contractor" or "manager". Everything starts with a contractor recognizing that it cannot accomplish a task alone. It breaks it into several subtasks and announces these subtasks for potential contractors. In the next step, the manager receives and evaluates bids from different contractors and finally awards a contract to a suitable contractor. A contractor may also divide a subtask into new subtasks. So the team formation process is managed by the contracting. And the final team working on the initial problems is a collection of contractor and manager nodes (some nodes act as both) that spring out from the initial problem.

Subproblem solution

In this stage, the subproblems identified during problem decomposition are individually solved. This stage typically involves *sharing of information* between agents: one agent can help another out if it has information that may be useful to the other. This process may be interleaved with the processes of problem decomposition and task allocation.

Solution synthesis

In this stage, solutions to individual subproblems are integrated into an overall solution. As in problem decomposition, this stage may be hierarchical, with partial solutions assembled at different levels of abstraction.

A typical mode of cooperative coordination that is involved in CDPS is *task sharing*, where components of a task are distributed to component agents. Task sharing relates to task allocation, and considers how tasks are to be allocated to individual agents. In a homogenous system a task could be allocated to any agent, but in a system where agents are different and have different capabilities and the benevolence assumption is not present, task sharing is a more complex problem.

Another typical mode of cooperative coordination involved in CDPS, in addition to task sharing, is *result sharing*. Result sharing is distribution of information, for example partial results that is relevant to subtasks. Information can be shared proactively (one agent sends another agent some information because it believes the other will be interested in it), or reactively (an agent sends another information in response to a request that was previously sent). Result sharing may be realized through message passing or through a shared memory.

2.3 Shared Memory Structures

A shared memory structure may be used by agents in a MAS for indirect communication of information. During the cooperative distributed problem solving process, described in section 2.2.3, agents working on the same problem share tasks and results. By using a shared memory structure, agents may place results

from their actions in this common repository, so that they are available to other interested agents. When a shared memory is used like this, it can also be seen as a common workspace or working memory. Next, we describe some research areas focusing on the idea of shared memory structures.

2.3.1 Blackboard Systems

For the past quarter century, AI researchers have used the paradigm of *collaborating software systems* to tackle large and difficult problems. Blackboard systems were the first attempt at integrating "cooperating" software modules. The goal was to achieve the flexible, brainstorming style of problem solving performed by a group of human experts working together to solve problems that no single expert could solve alone. Multiagent system research approaches the collaborating software paradigm from an agent-centric orientation [7, 37], but they still use some ideas from blackboard systems, like shared memories. Blackboard systems are thoroughly described in [9, 17, 6], here we just give a short introduction.

In a blackboard system, domain knowledge is organized into a set of diverse and independent processes called *knowledge sources* (KSs) or specialists. Each knowledge source contains knowledge about one aspect of the overall problem-solving task. Knowledge sources do not call one another directly. Instead, they interact indirectly through a shared database called the *blackboard*.

The blackboard is subdivided into a set of distinct information levels, each representing a different view of the global solution space. The basic data unit of the blackboard is the *hypothesis*. A hypothesis represents a partial solution expressed at one of the information levels of the blackboard. The set of possible hypotheses at a level represents the search space at that level. Relationships among hypotheses at different levels on the blackboard are represented by links, which allow a partial solution at one level to constrain the search at another level.

Knowledge sources are invoked in response to particular kinds of changes on the blackboard, called *events*. An intelligent *controller* determines which knowledge sources should execute their actions at each step in the problem-solving process. Figure 2.4 shows the architecture of a basic blackboard system.

Some important characteristics of blackboard systems are:

- **Independence of expertise.** Each knowledge source is an expert on some aspects of the problem and can contribute to the solution independently of the particular mix of other specialists in the system.

- **Diversity in problem-solving techniques.** The internal representation and inferencing machinery used by each knowledge source are hidden from direct view.

- **Flexible representation of blackboard information.** The blackboard model does not place any prior restrictions on what information can be placed on the blackboard.

- **Common interaction language.** Knowledge sources must be able to correctly interpret the information recorded on the blackboard by other knowledge sources.

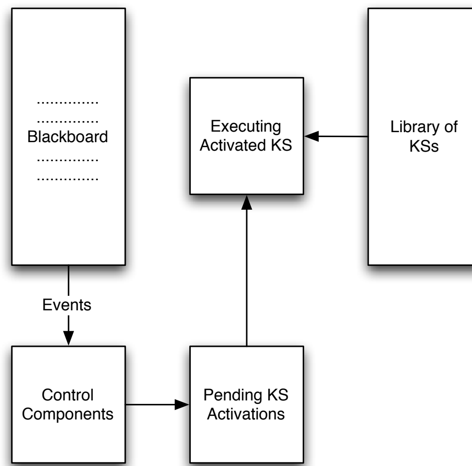

Figure 2.4: The architecture of a basic blackboard system, showing the blackboard, knowledge sources or agents, and control components, taken from [36].

- **Event-based activation.** Knowledge sources are triggered in response to blackboard and external events. Blackboard events include the addition of new information to the blackboard, a change in existing information, or the removal of existing information. Rather than having each knowledge source scan the blackboard, each knowledge source informs the blackboard system about the kind of events in which it is interested. The blackboard records this information and directly considers the knowledge source for activation whenever that kind of event occurs.

- **Need for control.** A control component that is separate from the individual knowledge sources is responsible for managing the course of problem solving.

- **Incremental solution generation.** Knowledge sources contribute to the solution as appropriate, sometimes refining, sometimes contradicting, and sometimes initiating a new line of reasoning.

2.3.2 The Publish/Subscribe Model

The publish/subscribe model and some applications are described in [11, 24, 3, 31, 30]. Well adapted to large-scale distributed applications, the publish/subscribe communication paradigm has recently received increasing attention. With systems based on the publish/subscribe interaction scheme, subscribers register with their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers.

The publish/subscribe interaction paradigm provides subscribers with the ability to express their interest in an event or a pattern of events, in order to be notified subsequently of any event, generated by a publisher, that matches their registered interest. In other terms, producers publish information on a

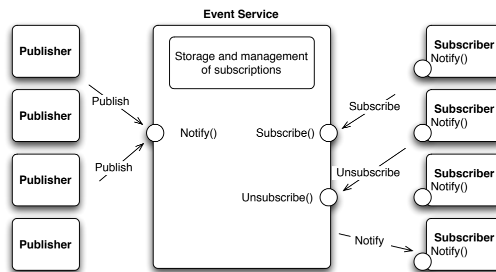

Figure 2.5: A simple object-based publish/subscribe system, consisting of publishers, subscribers and a central event service, taken from [11].

software bus (an event manager) and consumers subscribe to the information they want to receive from that bus. This information is typically denoted by the term *event* and the act of delivering it by the term *notification*.

The basic system model for publish/subscribe interaction, in figure 2.5, relies on an event notification service providing storage and management for subscriptions and efficient delivery of events. Such an event service represents a neutral mediator between publishers, acting as producers of events, and subscribers, acting a consumers of events. Subscribers register their interest in events by typically calling a *subscribe()* operation on the event service, without knowing the effective sources of these events. This subscription information remains stored in the event service and is not forwarded to publishers. The symmetric operation *unsubscribe()* terminates a subscription. To generate an event, a publisher typically calls a *publish()* operation. The event service propagates the event to all relevant subscribers. Thus, the event service can be seen as a shared memory structure.

Some advantageous properties introduced by this paradigm is:

- **Space decoupling.** The interaction parties do not need to know each other.
- **Time decoupling.** The interacting parties do not need to be actively participating in the interaction at the same time.
- **Synchronization decoupling.** Publishers are not blocked while producing events, and subscribers can get asynchronously notified of the occurrence of an event while performing some concurrent activity.

2.4 Summary

In summary, a multiagent system contains a number of autonomous, distributed agents which interact directly or indirectly trough respectively direct or indirect communication. Multiagent environments or multiagent architectures have to provide an infrastructure specifying communication and interaction protocols. The agents are also able to act in an environment where they have different "spheres of influence", when these spheres of influence overlap, relationships will

appear among agents and their actions [37]. When agent actions are dependent in some way the agents need to coordinate their action to be able to fulfill their own design objectives, and the overall design objectives of the multiagent system. To support this a multiagent system may provide different coordination mechanisms.

In CDPS (cooperative distributed problem solving), agents solve problems using four steps: 1) Problem decomposition, 2) Subproblem distribution; task allocation, 3) Subproblem solving, and 4) Solution Synthesis. How the different steps are solved, amongst other things, depends on which coordination mechanisms and which protocols that are used in the multiagent system.

Agents in a multiagent system may communicate indirectly via a shared memory. The CoPS framework for cooperative problem solving in multiagent systems, that is proposed in this thesis, uses a shared memory called the TEAM SPACE. This shared memory adopts ideas from blackboard systems, and the publish/subscribe and push communication models. The TEAM SPACE architecture is presented in chapter 4. And in chapter 5, the CoPS framework with the TEAM SPACE is compared to relevant work, including applications of blackboard systems and the publish/subscribe communication model.

Chapter 3

An Architecture for Cooperative Problem Solving: CoPS

CoPS is a framework that should enable easy implementation of software agents solving problems cooperatively. The framework architecture is partly designed and a prototype is implemented at NTNU in Trondheim. This work is described in the master thesis of Gundersen [34]. In a project at NTNU, autumn 2005, we worked with the CoPS framework, using it to model and implement a problem from a health center domain. This work is described in our project report [33]. Using CoPS on a real world problem, uncovered some lacks and errors in the existing CoPS prototype. Some of these findings are corrected as a part of the work with this thesis. In addition we extend the CoPS framework with a shared memory structure, used for the cooperative problem solving of agents.

This chapter gives a short introduction to the CoPS architecture, and it describes which parts of the architecture that are already developed and implemented, and which parts that will be covered by this thesis. In section 3.1, we give an overall view of the CoPS System. This is not done in earlier work with CoPS. In section 3.2, a three layered CoPS architecture is proposed. The CoPS architecture was also described in [34, 33], but in this chapter we take a different approach to describing the architecture. Instead of elaborating the different parts independently, we give a more wholesome view of the architecture. This description also includes the shared memory structure. An example of how a problem may be solved cooperatively by agents in CoPS is outlined in section 3.3. Section 3.4 points on parts of the CoPS framework, that have been developed in earlier work. And finally in section 3.5, we propose the corrections and extensions to CoPS.

Basically this chapter is provided to give an explanation of the CoPS framework, to introduce the idea of a shared memory structure in CoPS, and to outline the focus of our work described in the remaining chapters.

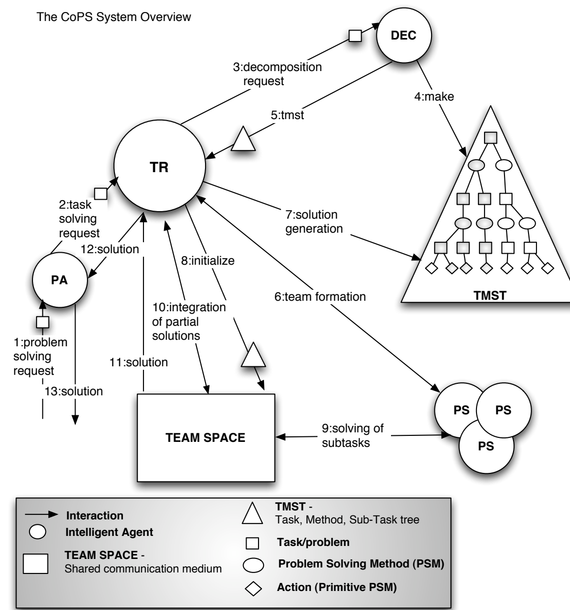

Figure 3.1: The overall CoPS system view. The CoPS system has four types of agents: Task Responsible (TR), Decomposer (DEC), Problem Solver (PS), and Personal Assistant (PA). Tasks handled by CoPS are decomposed into a TMST (Task Method Sub-Task tree). Problem solving and integration of partial results are supported by a shared memory - the TEAM SPACE.

3.1 The overall CoPS System

The CoPS framework specifies intelligent agents, protocols, coordination mechanisms and knowledge representation needed to support cooperative distributed problem solving (CDPS). Earlier work with CoPS focuses on the team formation part of the CDPS process. A capable team of software agents is formed each time the system receives a new problem, based on the current configuration of the system (which agents with which capabilities that are available). A team consists of software agents capable of solving problems as experts. Reconfiguration of the team should be possible if agents happen to change their mind about joining the team, are disconnected, or fail in any other way.

The overall CoPS system is pictured in figure 3.1. This view shows each step in the process of receiving a problem request from a user, decomposing the problem, allocating tasks from the decomposition, solving the sub-tasks and integrating partial results (solution synthesis) by cooperation, and finally returning the problem solution. Handling of failures and re-planning/reconfiguration are not incorporated into this view. Cooperation is achieved through the organization of agents into different types, a Task Method Sub-Task (*TMST*) tree, and task allocation. A shared memory structure - the *TEAM SPACE*, enables result sharing, and sharing of plans and resources needed for solving a problem. There are four different types of agents: *Personal Assistant* (PA), *Task*

Responsible (TR), *Problem Solver* (PS), and *Decomposer* (DEC). The most important interactions between the agents, the TMST and the TEAM SPACE, are enumerated in their corresponding order in the overall view of the system:

1. PA receives a **problem solving request** from the user. It converts the problem into a format that is understandable by the system; a task.

2. PA forwards the task in a **task solving request** to TR.

3. TR forwards the task in a **decomposition request** to DEC, for it to decompose the task into a TMST.

4. DEC **makes** the TMST (Task, -Method, Sub-Task tree) from the task received from TR. The TMST is a hierarchy of tasks which are solved by one of the attached problem solving methods (PSMs). At the leaves we have primitive PSMs or actions. The root of the TMST is the task received from the TR. The DEC also attaches a teamID to the TMST. The teamID is used to identify the group of agents that later are chosen to solve the problem.

5. DEC returns the decomposition of the received task from the TR in **tmst**. As the TR receives the task decomposition it can start forming a team of PSs. TR needs information about what agents are capable of the actions specified in the TMST. This information is requested from a Matchmaker (not shown in the overall system view), keeping a list of all available agents in the system and what services (actions, capabilities) they offer.

6. TR does the **team formation** by using a contract net interaction protocol. Each of the PSs capable of performing an action, receives a call-for-proposal message. Then, the PSs return a propose message with a price for performing the action. After step 7, TR sends an accept-proposal message to the chosen PSs, with an invitation to join the team. The accept-proposal messages contains the teamID, associated with the TMST representing the problem. PSs that was invited returns an inform-message as an acceptance of the invitation.

7. TR does the **solution generation** on the TMST. As the TMST represents several different ways of solving the initial task, the TR must find the best possible solution. The solution generation takes in to consideration the proposal messages received from the PSs in step 6. As a result of the solution generation, the TMST is updated by attaching chosen PSs for the final solution to their corresponding actions (task allocation), and TMST-nodes part of the solution are tagged. A solution is represented in figure 3.1 by the shaded nodes.

8. TR **initialize** the TEAM SPACE, by sending the TMST, now containing a solution.

9. TEAM SPACE uses the information stored in the TMST to coordinate the PSs in the **solving of subtasks** process.

10. TEAM SPACE uses the information stored in the TMST to coordinate the TR in the **integration of partial solutions** process.

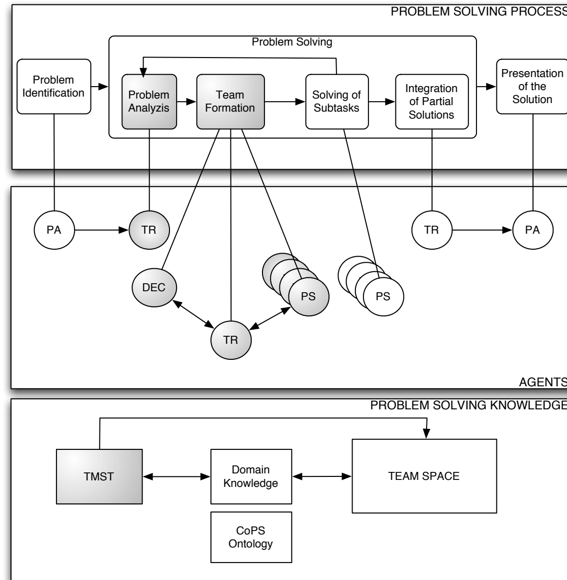

Figure 3.2: A three - layered CoPS Architecture. The layers are: PROBLEM SOLVING PROCESS, AGENTS and PROBLEM SOLVING KNOWLEDGE. The agents abbreviations are: Task Responsible (TR), Decomposer (DEC), Personal Assistant (PA), and Problem Solver (PS). The shaded shapes reflect the focus of the current work with CoPS.

11. When the initial task is solved, TEAM SPACE returns the **solution** of this task to the TR.

12. TR returns the **solution** of the initial task to the PA.

13. PA presents the **solution** of the requested problem for the user.

The overall view of the CoPS system described here is comprised of the three main architectural constructs: the *problem solving process*, the *agents*, and the *problem solving knowledge*. These constructs are represented in the next section as a three-layered architecture.

3.2 A three-layered CoPS Architecture

The three-layered CoPS architecture depicted in figure 3.2 gives three different but integrated views of the architecture. The *problem solving knowledge* at the bottom-layer is used by the *agent* at the middle-layer to carry through the *problem solving process* at the top-layer.

3.2.1 Problem Solving Knowledge

The bottom-layer of the CoPS Architecture has four different components: *TMST*, *Domain Knowledge*, *CoPS Ontology*, and *TEAM SPACE*. The TEAM

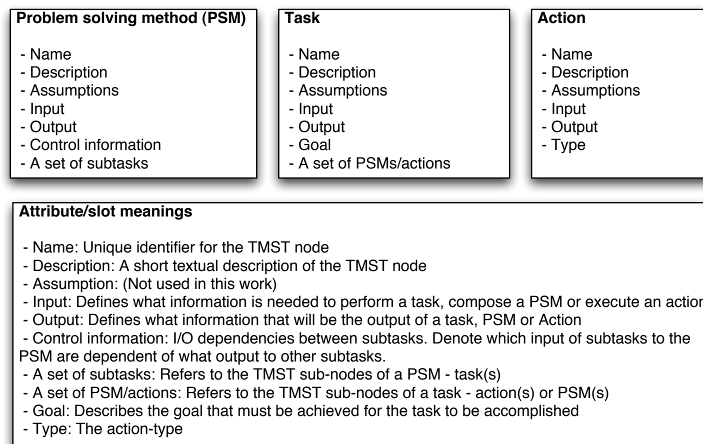

Figure 3.3: The nodes part of the TMST (Problem Solving Method - PSM, task and action) and their attributes. Short explanations to the attributes are provided. Modified from [34].

SPACE, when initialized for problem solving, contains a reference to a TMST. Data or knowledge stored in the TMST and in the TEAM SPACE is part of the domain knowledge. And finally, the domain knowledge is built using the CoPS ontology.

TMST

The TMST (Task Method Sub-Task tree) is comprised of *tasks* and *problem solving methods*. Tasks represent what to do. Each task specifies a goal, and the goal is achieved when the related task is accomplished. Problem solving methods (PSMs) represent how to do it - how to perform a task. There are two different types of PSMs. The first type is task decomposition methods, which define a set of subtasks needing to be achieved in order for the original task to be accomplished. The second type is a primitive problem solving method, an *action*, representing executable code which may be performed directly. Indicating that actions capture the skills of an agent. Tasks, PSMs and actions have several attributes. Attributes related to the TMST nodes and their meanings are described in figure 3.3.

Tasks and PSMs are structured in a directed acyclic graph, - a TMST, as shown in figure 3.4. A task may be achieved by one or more alternative PSMs, thus PSMs connected to a task have OR relations between them. A PSM specifies a set of tasks which have to be performed for the PSM to be fulfilled, thus tasks connected to a PSM have AND relations between them. This recursion continues until all leaf nodes in the graph are primitive PSMs. A PSM defines dependencies between its subtasks, and thus dependencies between the agents.

The root node of the graph is a task representing the initial problem. The TMST captures different ways of solving that root-node-task, it is a description

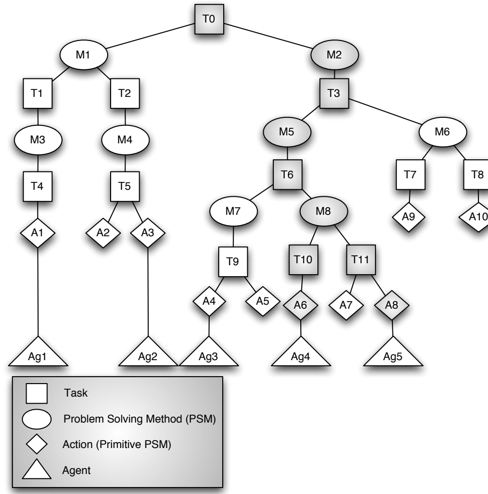

Figure 3.4: The TMST - the complete solution space for a task, and the (chosen) solution of the TMST (marked with grey nodes) - a specific solution for a task. Modified from [34].

of the *complete solution space* for the initial problem. The complete solution space is not needed for solving the problem, just one way of solving the problem would be sufficient. In the TMST shown in figure 3.4, the complete solution space is represented by the tree in total, and one particular solution to a problem is represented by the shaded nodes.

The TMST is generated in a top-down manner, and the solution is found by a bottom-up approach. The generation of a specific solution takes into consideration the agents that are currently in the system, and their skills and capabilities.

After the TMST is generated, each action is allocated, by a TR, to an agent having skills to perform this action. In figure 3.4, the agents Ag1, Ag2, Ag3, Ag4 and Ag5 are all capable of, and willing to perform the actions. Now, the solution generation of the TMST is done in three steps:

1. The solution space for the current situation (not the complete solution space) is activated. All actions connected to an agent are activated. All tasks connected to an activated action or PSM are activated. And, all PSMs having all of their subtasks activated are activated. If the initial task gets activated, the problem has at least one solution.

2. Costs are propagated through the solution space. Each agent - action relation has a cost. One starts at the bottom level and propagates the costs upwards in the tree of activated nodes. For all of the tasks one always choose the cheapest alternative of solving it (choose one of the sub-PSMs/sub-actions).

3. All chosen nodes (cheapest alternatives) in the cost propagation are tagged as part of the solution.

Domain knowledge and CoPS Ontology

An agent needs knowledge about the domain it works in and about the problem being solved. This knowledge is represented in the agent's knowledge base. If the knowledge is shared between agents, the same ontology for representing the knowledge must be used, in such a way that the agents are able to understand this knowledge. Remember that the ontology is the vocabulary used in the content of messages exchanged between agents.

The domain knowledge should be represented using a domain ontology in combination with a CoPS ontology. The CoPS ontology describes concepts used by the system. These concepts may involve the problem solving process, agents, structure of the TMST and the content of the TEAM SPAE. By wrapping the domain knowledge into this framework specific ontology, it would be easier to increase the domain independence of our framework. Using a CoPS Ontology results in standardization of the general parts of CoPS, in such a way that the integration of the architectural components is more elegant. Messages exchanged between agents may conform to one standard by encoding the content using the CoPS Ontology.

TEAM SPACE

The TEAM SPACE assists in coordinating agent actions in the *solving of subtask* and *integration of partial solutions* processes. The realization of the TEAM SPACE is the main contribution of this thesis and is thoroughly elaborated later.

3.2.2 Agents

The CoPS framework architecture proposes five different types of agents. The agents have a generic agent architecture. This generic architecture and detailed process charts for each of the agent's behaviours are detailed in [34] and in [33]. Here we just give a short description of the different agent types. The different agent types are:

- **Personal Assistant** (PA). The PA is the bridge between the user of the software system and the software system. It will receive problems and information, and present the software system's solution in a proper way.

- **Task Responsible** (TR). The TR is responsible for seeing to that the specified problem is solved at run-time. It must gather a team of PSs, administer the problem solving process and integrate partial solutions.

- **Matchmaker**. The matchmaker knows the different agents in the software system, it is also responsible for knowing which services they offer and where to find them. The matchmaker responds with the services offered by agents in the system on request, and in this way helps the agents finding each other. This agent is not a "visible" component of the architectural view in figure 3.2, but it is still a part of CoPS. Any agent that enters the system has to register its services with the matchmaker. And any agent, at any time, can communicate with the matchmaker to get information about other agents.

- **Decomposer** (DEC). The job of the DEC is to decompose tasks - make TMSTs. It knows how complex problems can be decomposed and how the components of the decomposition are related to each other. A DEC decomposes tasks on request, and returns a TMST describing the decomposed task, attached with a teamID.

- **Problem Solver** (PS). PSs are the workers in the system, and are capable of solving specific problems or tasks. They may join a team to help out in a cooperative problem solving process in the solving of subtasks, or they may solve problems on their own - without joining a team. PSs solve problems on request.

Interaction patterns between the different agent types, also marked with arrows in figure 3.2, is described together with the different stages of the problem solving process, in the next section. We also describe how the problem solving knowledge is used.

3.2.3 Problem Solving Process

Different agents take part in different steps of the problem solving process. Which agents are involved in each step of the problem solving process is indicated in figure 3.2. During the process, agents have to communicate. For the conversations, FIPA interaction protocols are used. Remember that interaction protocols guide the flow of messages between agents. FIPA (The Foundation for Intelligent Physical Agents) is an IEEE Computer Society standards organization that promotes agent-based technology and its standards with other technologies [13]. The FIPA web-page referenced by [13], has links to a collection of articles describing the different FIPA standards, as these articles are referenced in the text, the citation will be extended by the number of the article, so that it can easily be found at the web-page.

Problem Identification and Presentation of the Solution

The only agent involved in the *problem identification* - and *presentation of the solution* processes is the PA. The interactions 1 and 13 in the overall system view, depicted by figure 3.1, are part of these processe. CoPS does not specify how to implement the PA, since the focus of this framework is the cooperative problem solving process.

After the PA has finished the problem identification, turning the problem into a (initial) task, it communicates the task to the agent(s) performing the problem solving process. When the result of the problem solving process has been reached, the solution (to the initial task) has to be communicated to the PA. All conversations conform to FIPA standards, and the conversation between the PA and the agents solving the problem is similar to the request interaction protocol specified by FIPA in [13] (document number: SC00026). The request interaction protocol is pictured in figure 3.5. Thus, the PA requests the TR to solve the problem specified by the user. If the TR accepts the request, it should either answer with an inform message specifying the solution, or a failure message specifying what went wrong. This interaction between the TR and the PA frames the whole problem solving process. The protocol is initiated

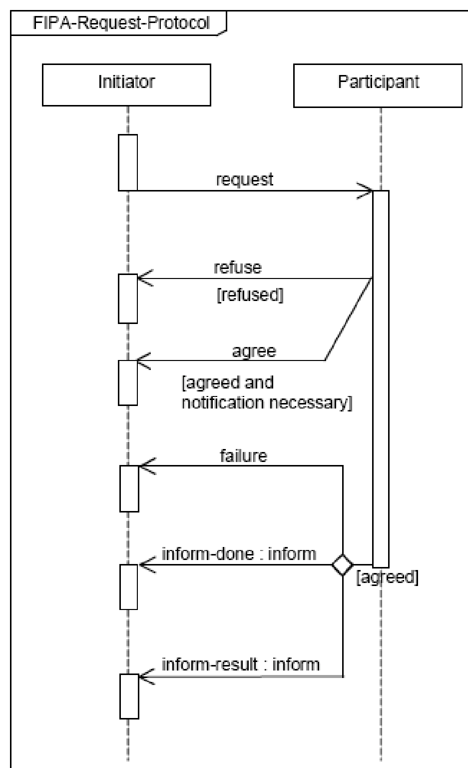

Figure 3.5: The FIPA Request Interaction Protocol, describing a conversation starting with a request - message being sent from an Initiator to a Participant. Taken from [13].

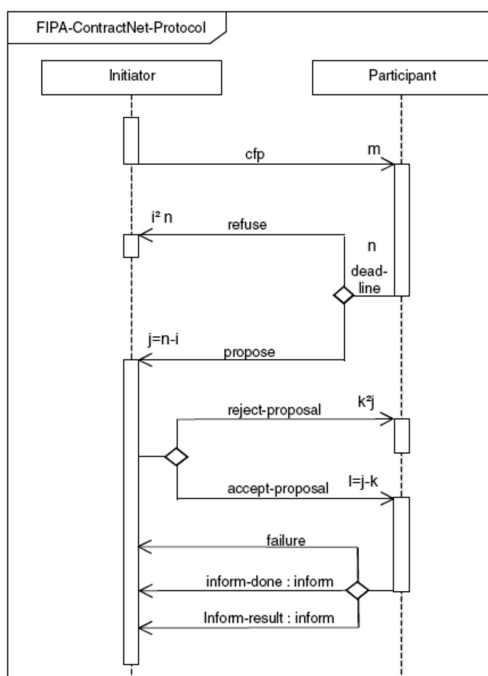

Figure 3.6: The FIPA Contract Net Interaction Protocol, describing a conversation starting with a call-for-proposal (cfp) message being sent from an Initiator to a Participant. Taken from [13].

at the beginning of the problem solving process and finally completed when the solution is attained, or a failure has occurred.

Problem Solving

The *problem solving* process involves the interactions 2-12 in the overall system view depicted by figure 3.1. The *problem analysis*-process is performed by the TR. First, the TR checks whether the request is coded in a language and uses ontologies that the agent knows of. Next, it checks whether it may solve the problem without involvement from other agents. If it is not able to do the job by itself, a team is required. The team is formed on basis of the TMST, in the *team formation* -process.

The TMST is generated by the DEC. In order to get the TMST, the TR has to initiate a conversation with the DEC. This conversation also conforms to the specified FIPA Request Interaction Protocol. When a TMST is received, the TR may start finding PSs that should join the problem solving team. TR needs information about what agents are capable of the actions specified in the task structure tree. This information is requested from the Matchmaker. Again, the FIPA Request Interaction Protocol is used.

Next, agents for the problem solving team are recruited by using the FIPA Contract Net Interaction Protocol [13] (document number: SC00029), pictured

in figure 3.6. CoPS uses this protocol with kind of a twist. Instead of sending the participant an accept-proposal message containing the action to be performed, the initiator sends an accept-proposal message containing an invitation to join the problem solving team, with an attached teamID. If the participant is a PS, and therefore knows about team work, it will answer with a inform message saying that it joins the team. If the participant, on the other hand, does not know the CoPS way of cooperating in teams, it will respond the call-for-proposal message with a not-understood message. The team is formed, on the basis of the TMST solution part.

The *solving of subtasks* process is performed by the PSs, and the *integration of partial solutions* process is performed by a TR. These two processes are coordinated by the TEAM SPACE. PSs perform (solve) subtasks by executing actions. Their partial results, representing task solutions, are shared through the TEAM SPACE. When all subtasks of a PSM have a solution in the TEAM SPACE, the TR integrates these partial solutions. The partial result from this integration, representing a task solution, is shared through the TEAM SPACE. When there is a solution to the initial task, the result is returned to the PA.

3.3 An Example

In this section we describe how a problem is solved by the CoPS system, step by step. Details of what goes on in the TEAM SPACE is left out though, and properly described in the next chapter, 4. Imagine that we have a system with these agents and capabilities:

- PA - Receives request from a user
- TR - Administrates the problem solving process
- DEC - Decomposes a task
- PSs - Solve simple problems:
  - PS-LA (Laboratory Agent) - Knows how to take a bloodtest and how to analyze it manually
  - PS-LSA (Laboratory System Agent) - Knows how to operate the laboratory system, and can analyze a bloodtest mechanically.
  - PS-PDBA (Patient Database Agent) - Knows how to (update and) get patient info in the patients database.
  - PS-RA (Receptionist Agent) - Knows how to (update and) get patient info in a manual patients archive.

The steps of the problem solving process, performed by the listed agents, are described according to the enumerated interactions pictured in figure 3.1. The steps are:

1. PA Receives a problem solving request from the user: "Perform a laboratory test on a patient. The patient's name is Paul".

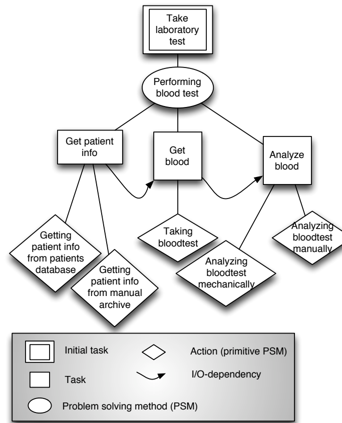

Figure 3.7: The TMST returned by the Decomposer (DEC) when the task, *(Task :name takeLaboratoryTest :input (Patient :name Paul))*, is received.

2. PA forms the request into a task, which is part of the *Problem Identification*-process. The task looks like this: *(Task :name takeLaboratoryTest :input (Patient :name Paul))*. This format reflects an ontology, where a *Task*-entity has values for the attributes *name* and *input*. The Task-entity could belong to the CoPS ontology. A *Patient*-entity (from the domain ontology) has a value for the attribute *name*. The instance of Patient serves as the Task-input-value. A task solving request containing this task is forwarded to the TR agent.

3. The TR agent performs the *Problem Analysis*-process. As it is not capable of solving the task alone, it has to start a *Team Formation*-process, and forwards the task in a decomposition request to DEC.

4. The DEC makes the TMST illustrated in figure 3.7. Each of the nodes in the TMST also has some attributes that are defined using the domain ontology. These attributes are listed in 3.1.

5. DEC returns the TMST to the TR. The TR then requests the Matchmaker of PSs that knows how to perform the actions: gettingPatientInfoDB, gettingPatientInfoMA, takingBloodtest, analyzingBloodMech, analyzingBloodMan. The matchmaker responds with the list: PS-DBA -> gettingPatientInfoDB, PS-RA -> gettingPatientInfoMA, PS-LA -> takingBloodtest and analyzingBloodMan, PS-LSA -> analyzingBloodMech.

6. TR makes a call-for-proposal message to all the PSs received from the Matchmaker, asking them what it would cost to perform the associated action. These propose messages are received from the PSs:

   - PS-DBA: *(Cost :action gettingPatientInfoDB :price 5)*
   - PS-RA: *(Cost :action gettingPatientInfoMA :price 8)*
   - PS-LA: *(Cost :action takingBloodtest :price 4)*, *(Cost :action analyzingBloodMan :price 9)*

Table 3.1: The attributes related to the TMST-nodes in the TMST returned by the DEC when the task *(Task :name takeLaboratoryTest :input (Patient :name Paul)), is received.*

task	takeLaboratoryTest
input	*(Patient :name Paul))*
output	*(Patient :name x :test_results (TestResult :description y))*
goal	*(Patient :name x :test_results (TestResult :description y))*
PSM	performingBloodTest
input	*(Patient :name x)*
output	*(Patient :name x :test_results (TestResult :description y))*
task	getPatientInfo
input	*(Patient :name x)*
output	*(Patient :name x :info y)*
goal	*(Patient :name x :info y)*
action	gettingPatientInfoDB
input	*(Patient :name x)*
output	*(Patient :name x :info y)*
action	gettingPatientInfoMA
input	*(Patient :name x)*
output	*(Patient :name x :info y)*
task	getBlood
input	*(Patient :name x :info y)*
output	*(Patient :name x :info y :blood_sample_reference z)*
goal	*(Patient :name x :info y :blood_sample_reference z)*
action	takingBloodtest
input	*(Patient :name x :info y)*
output	*(Patient :name x :info y :blood_sample_reference z)*
task	analyzeBlood
input	*(Patient :name x :info y :blood_sample_reference z)*
output	*(Patient :name x :info y :test_results (TestResult :description m))*
goal	*(Patient :name x :info y :test_results (TestResult :description m))*
action	analyzingBloodMech
input	*(Patient :name x :info y :blood_sample_reference z)*
output	*(Patient :name x :info y :test_results (TestResult :description m))*
action	analyzingBloodMan
input	*(Patient :name x :info y :blood_sample_reference z)*
output	*(Patient :name x :info y :test_results (TestResult :description m))*

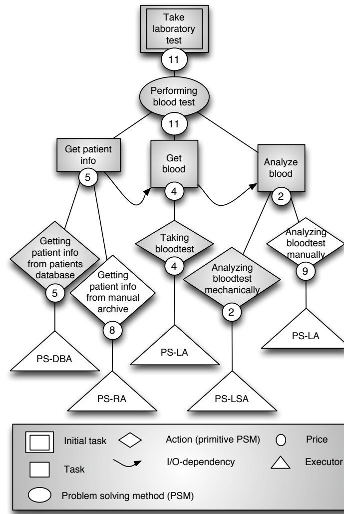

Figure 3.8: The TMST showing the solution, marked with shaded nodes. The lowest cost suggested for actions or PSMs propagates upwards in the tree-structure. Actions are connected to executors (Problem Solvers, PSs)- that has offered to execute them for the price attached to the corresponding action.

- PS-LSA: *(Cost :action analyzingBloodMech :price 2)*

7. TR tries to generate the solution of the TMST with the lowest cost. The solution and its costs are marked in figure 3.8. The figure also shows which agents are added as executors to the different actions. The agents connected to the shaded actions, receive an invitation to join the problem solving team. Each invitation is attached with the same teamID.

8. TR requests for initialization of the TEAM SPACE. The TMST in figure 3.8, and all the pictured information is used to initialize the TEAM SPACE.

9. TEAM SPACE uses the information in the TMST to coordinate the PSs in the problem solving. Details about this, are described in the example presented in chapter 4, after the TEAM SPACE architecture has been outlined.

10. TEAM SPACE uses the information in the TMST to coordinate the TR in the integration of partial solutions. Details about this, are described in the example presented in chapter 4, after the TEAM SPACE architecture has been outlined.

11. When the initial task is solved, TEAM SPACE returns the solution of the task to the TR: *(Task name: takeLaboratoryTest output: (Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))).*

12. TR returns the solution of the initial task to the PA: *(Task name: takeLaboratoryTest output: (Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))).*
13. PA presents the solution for the user: "The laboratory test performed on patient Paul says that NA is normal, Glucose is high and HB is low".

In chapter 4, the same example is extended to show how the TEAM SPACE works.

3.4 Current work with CoPS

The focus of earlier work with CoPS is marked with the shaded components in our three-layered architecture, in figure 3.2. The implemented prototype and the work with the architecture does not complete all of the elements introduced so far. The implemented prototype include this functionality:

- **PROBLEM SOLVING PROCESS**. Problem Analyzis and Team Formation.
- **AGENTS**. The TR, DEC and PS are implemented with the functionality needed for realizing the implemented parts of the problem solving process. The PA is implemented as an empty shell. The Matchmaker is implemented in the pre-existing framework used for implementing the prototype. All of the conversations mentioned are implemented, these include: The conversation between TR and PA (request protocol), the conversation between TR and DEC (request protocol), the conversation between TR and Matchmaker (request protocol), and finally the conversation between TR and PS (modified contract net).
- **PROBLEM SOLVING KNOWLEDGE**. TMST.

In our 2005-project [33] we used the CoPS framework for a medical domain application - checkup system. Results from the modeling and implementation, showed that the CoPS framework was easy to use, and that it to a certain extent was applicable in a real-world scenario. The CoPS prototype had some failures and lack of flexibility. These results lay out the direction of work in this thesis, in addition to the extensions considering architectural elements.

3.5 Corrections and Extensions to CoPS

The corrections should improve the existing work with the CoPS framework. Extensions to the CoPS framework are a further realization of architectural components.

3.5.1 Corrections

Here we describe the corrections to CoPS that are supposed to be done in this work. We come back to which corrections that are actually done, while describing our results in chapter 7. The corrections solely involve implementation issues that were uncovered during implementation and test-run of the checkup

system, in our 2005-project. First, there are some code bugs (problems) that made the test-run of the checkup system fail. The problems should be removed by performing some extensive debugging. The problems are:

- The PSs send two exactly alike copies of their proposals to the TR.
- The TR does not answer the PSs with a proposal accepted message when the solution being pursued is chosen.
- The TR concludes that the team formation process succeeded, even though this was not the case.
- In certain configurations of the system the TR dies without being properly terminated.

Second, there are some problems that limited the implementation possibilities, related to lack of functionality, and thus to lack of flexibility. The problems are:

- PS agents can only have one single capability.
- PS agents can only attend to one conversation at a time.
- Dynamic formation of problem solving teams are not fully realized. Reconfiguration of the team is not possible if agents happen to change their minds about joining the team, are disconnected, or fail in any other way. By now, team formation is dynamic in the sense that it is formed according to the state of the system at the exact time it is needed. But, the CoPS framework architecture proposes that the team may change in real-time, depending on external changes to the environment.

Corrections to all of the problems limiting the implementation could as well be seen as extensions to CoPS. The limitations even point to important parts of the architecture that are not yet fully developed. These extensions are necessary to add, in order to fully test and integrate the TEAM SPACE with the CoPS framework.

3.5.2 Extensions

Here we describe the extensions to CoPS that are supposed to be done in this work. Like with the corrections, we come back to which extensions that are actually performed, while describing our results in chapter 7. The extensions to CoPS are related to architectural components. If we look at the architectural view of CoPS in figure 3.2 again, these are are the extensions to each of the three layers:

- **PROBLEM SOLVING PROCESS**. *solving of subtasks* and *integration of partial solutions* should be realized.
- **AGENTS**. TR - and PS behaviours should be extended in such a way that these agents are able to perform the problem solving process steps; *solving of subtasks* and *integration of partial solutions*.
- **PROBLEM SOLVING KNOWLEDGE**. A shared memory structure - the TEAM SPACE and a CoPS Ontology should be realized. And the TMST should support functionality and keep information needed by the TEAM SPACE.

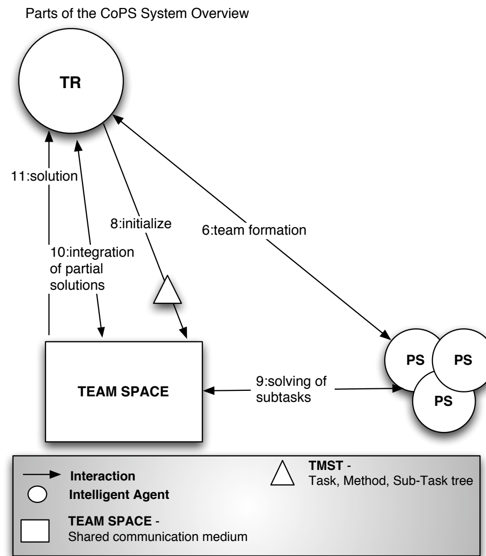

Figure 3.9: Parts of the overall CoPS system view depicted in figure 3.1, interacting with the TEAM SPACE. Agents interaction with the TEAM SPACE are Task Responsible (TR) and a group of Problem Solvers (PSs).

The different parts of the architecture involving our extensions are tightly connected. The TEAM SPACE is used to support the TR and PSs in the problem solving processes: *solving of subtasks* and *integration of partial solutions*. The TEAM SPACE is the main extension though. A part of the overall view of the CoPS system illustrated in figure 3.9, might help put some implications on what is the responsibilities of the TEAM SPACE.

When the TR has formed a team (figure 3.9, 6), and generated a solution to TMST (figure 3.1, 7 - not in the figure 3.9), it initializes the TEAM SPACE by submitting the TMST (figure 3.9, 8). This initialization can be seen as a request, that needs to be handled by the TEAM SPACE. The TEAM SPACE needs to be initialized according to the current problem to be solved.

All information needed about a problem is stored in the TMST. Problem Solving methods (PSMs) have control information, considering the I/O dependencies of their sub-tasks. Using this control information it is possible to infer in what order agent actions need to be executed, and what pieces of information an agent needs in order to perform an action. Each task in the TMST is related to a goal. Each action in the TMST is connected to an executor which stores information about the agent performing the action. PSs solve subtasks by executing actions.

The TEAM SPACE should be able to recognize when an action waiting for some input is ready to be executed, and inform the PS allocated to that action (figure 3.9, 9). When the PS has executed the action, the result is returned to the TEAM SPACE (figure 3.9, 9). Next, the TEAM SPACE should be able to recognize when goals at different levels in the TMST are met. When all of

the tasks (goals) of a PSM are performed (fulfilled), the TR should be notified so that it can integrate these partial solutions (figure 3.9, 10). After the TR has integrated a set of partial solutions, the result is again returned to the to the TEAM SPACE (figure 3.9, 10). And finally, when the initial task (goal) is performed (fulfilled), the solution is returned to the TR (figure 3.9, 11).

By looking at this description of the CoPS system we uncover some of the TEAM SPACE responsibilities. These responsibilities are translated into a set of functional requirements for the TEAM SPACE:

1. The TEAM SPACE must keep a structure for storing partial results from both the PSs and TR.

2. The TEAM SPACE must be able to reason about the partial results and the current problem to be solved. In such a way that it always knows which actions are ready to be executed, which partial results are ready to be integrated, and which goals are met. Said in another way, the TEAM SPACE must keep the correct state of the problem solving process at any time.

3. The TEAM SPACE must be able to communicate the correct information,efficiently, to the PSs and the TR (the problem solving team).

4. The TEAM SPACE must be able to handle the information received from the PSs and TR, and update the state of the problem solving process accordingly.

5. The TEAM SPACE must handle concurrent problem solving, by allocating a separate working area for each problem solving team.

6. The TEAM SPACE must be able to convert the knowledge stored in the TMST, so that it could be used to initialize the working area for a problem solving team.

In the next chapter, the TEAM SPACE architecture is presented. We review these requirements at the end of that chapter, and explain how they are fulfilled by the proposed TEAM SPACE architecture.

3.6 Summary

This chapter has given an overview of the CoPS architecture, describing both old and new features. On the basis of previous work with CoPS, the focus of the work with our thesis was outlined. This work involves corrections to the existing CoPS prototype, and extensions related to old and new components of the CoPS architecture. The main extension to CoPS is the TEAM SPACE, representing a shared memory structure for cooperative problem solving performed by the CoPS agents.

In the next chapter, the TEAM SPACE architecture is fully sketched and described, interfaces between the TEAM SPACE and other architectural elements in CoPS are defined, and necessary extensions to the existing CoPS architectural componets are realized.

Chapter 4

The TEAM SPACE Architecture

The TEAM SPACE architecture conceptualizes a shared memory structure, or a shared workspace used by the CoPS agents during the problem solving process. This architecture is an extension to the CoPS architecture introduced in chapter 3. There we also described how the TEAM SPACE is integrated with existing elements of the CoPS architecture.

In this chapter, section 4.1 starts out describing the core TEAM SPACE architecture. The core architecture contains the basic elements to make the the TEAM SPACE work. Section 4.2 presents an example which is based on the same problem as was exemplified in chapter 3 - section 3.3. More complicated issues, like parallelism and re-planning are discussed in section 4.3. The TEAM SPACE assists the Task Responsible (TR) and Problem Solvers (PSs) in coordinating their actions during the problem solving processes; *solving of subtasks* and *integration of partial results*. This necessarily means that these parts of the CoPS architecture (described in chapter 3) need to be extended and modified. These extensions and modifications are described in section 4.4. Finally, section 4.5 shows how the TEAM SPACE architecture meet the functional requirements listed in chapter 3.

Description of the TEAM SPACE architecture also serves as a conceptual design that is used to develop the existing CoPS prototype further.

4.1 The Core TEAM SPACE Architecture

In this section the core TEAM SPACE architecture is described. The TEAM SPACE is divided into two main components; the *TEAM SPACE (TS) Agent* and the *TEAM SPACE (TS) Structure*. A TEAM SPACE may consist of several TS Agents and TS Structures. One TS Structure is dedicated for each problem solving team. A TS Agent is the interface between a problem solving team and its TS Structure. Only the TS Agent may access the team's TS Structure. This is done to encapsulate all functionality considering the TEAM SPACE, within this architecture, and to make a clean, standardized and message based interface.

In this section, we first look at the interactions between the TS Agent, TS

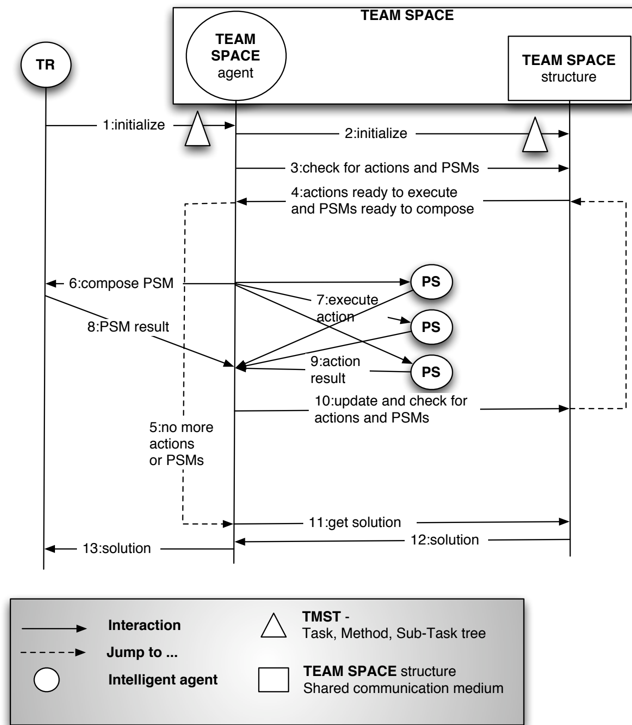

Figure 4.1: The Team Space Interface shows how Task Responsible (TR) and Problem Solvers (PSs) interact with the TEAM SPACE (TS) Agent, and how the TEAM SPACE (TS) Agent interacts with the TEAM SPACE (TS) Structure.

Structure, Task Responsible (TR), and Problem Solvers (PSs). Second, we look at the TS Agent behaviour. And finally, we look at the content and functions of the TS Structure.

4.1.1 TEAM SPACE Interactions and Interfaces

The TS Structure represents a shared memory, used by the agents during the problem solving process steps; *solving of subtasks* and *integration of partial results*. A TS Structure consists of different architectural components. For now, we will just look at the TS Structure as merely a black box. The only things we need to know here is that the TS Structure keeps track of the problem solving state and has functions for manipulating it. These function include: initializing the first state of the problem solving with information from a TMST, updating the problem solving state during the problem solving, and extracting information from the current problem solving state. The problem solving state, among other things, defines which actions are ready to execute (*solving of subtasks*) and which PSMs are ready to compose (*integration of partial solutions*).

When a team is successfully formed, the TR wants to initialize a TS Struc-

Table 4.1: How the interactions in the figure 3.1 relate to the interactions in figure 4.1

CoPS system view interactions	TEAM SPACE interactions
8:initialize	1:initialize
9:problem solving	7:execute action and 9:action result
10: integration of partial results	6:compose PSM and 8:PSM result
11:solution	13:solution

ture that will guide the problem solving team in solving the initial task. A TS Structure is not directly accessible for the TR though. The TS Structure may only be accessed by a TS Agent. The TS Agent, like the other CoPS agents, registers with the Matchmaker when entering the environment. When the TR wants to initialize a TS Structure, it first has to request the Matchmaker for available TS Agents, it chooses one of them, and then sends a request to initialize a TS Structure. This is the starting point for the interactions depicted in figure 4.1.

Interactions between the TEAM SPACE and other CoPS components were also described in chapter 3 - figure 3.1, showing the overall CoPS system view. These interactions are further detailed in figure 4.1. How the interactions in these two figures relate are listed in table 4.1. Interactions illustrated in figure 4.1 are:

1. TR sends a request to **initialize** a TS Structure to the TS Agent. Attached to this request is a copy of the TMST kept by the TR. At this point nodes part of the TMST solution are tagged, and the TMST has a teamID. This interaction actually contains several messages that are sent between the TR and the TS Agent. It is realized by the FIPA request interaction protocol shown in 3 - figure 3.5. The agree and refuse messages of the protocol have information telling the TR if the initialization went OK or not. The inform-result message of the protocol is not sent for the TR until interaction 13.

2. The TS Agent **initializes** a TS Structure by creating a new TS Structure component and providing it with the TMST. Then the TS Structure initializes the first problem solving state, with information from the TMST. The TS Structure component then "belongs to" the team of PSs with the team ID stated by this TMST. The TS Agent keeps a list of TS Structure components, for each of the problem solving teams it supports.

3. After the TS Structure is initialized, the TS Agent **checks for actions and PSMs**, in the TS Structure, ready to be executed or composed.

4. The TS Structure extracts information from the current problem solving state and returns a list of **actions ready to execute and PSMs ready to compose**. Actions and PSMs in the list are attributed with the necessary information. An action has a name, an address for the PS agent that should execute it, any needed input for the action, and a team ID. A PSM has a name, and address for the TR agent that should compose

it, any needed input for the PSM, the partial results that from the PSMs sub-tasks that are to be composed and a team ID.

5. If there are **no more actions** ready to execute **or PSMs** ready to compose, the TS Agent jumps to interaction 11.

6. If there are any PSMs ready to compose, the TS Agent requests a TR to **compose the PSM**. The TS Agent gets the address of the TR from the PSM's attributes. The rest of the attributes are attached to the request. This interaction is a request message, part of the FIPA request protocol, and interaction 8 is the inform message of the protocol. The TS Agent sends one request for each of the PSMs.

7. If there are any actions ready to execute, the TS Agent requests a PS to **execute the action**. The TS Agent gets the address of the PS from the action's attributes. The rest of the attributes are attached to the request. This interaction is the request message part of the FIPA request protocol, and interaction 9 is the inform message of the protocol. The TS Agent sends one request for each of the actions.

8. TR uses information in the corresponding PSM node in the TMST to compose the partial results. The **PSM result** is returned to the TS Agent.

9. A PS keeps the necessary knowledge about how an action is executed, and uses this knowledge and the action - input to generate a result. The **action result** is returned to the TS Agent.

10. After the TS Agent has received all the results from the PSMs and actions it forwards them to the TS Structure and tells it to **update** the problem solving state. At the same time, the TS Agent **checks for actions and PSMs**, in the TS Structure, ready to be executed or composed. After this we jump back to interaction 4.

11. The TS Agent tries to get the solution from the TS Structure.

12. The TS Structure checks if the initial task of the TMST has a solution by extracting information from the current problem solving state, and returns it to the TS Agent as the final solution of the problem.

13. The TS Agent forwards the solution found in the TS Structure to the TR agent. This interaction is a completion of the request protocol initiated by interaction 1, and it represents the inform-result message of the protocol.

From these listings of interactions we see that the interface between the TS Agent and TR agent is realized by instances of the FIPA request protocol, and the interface between the TS Agent and a PS agent is also realized by the FIPA request protocol. The interface between the TS Agent and the TS Structure could be realized by method invocations. The interaction pointing from the TS Agent to the TS Structure invoke a method, and the interaction pointing from the TS Structure to the TS Agent are results from the method invocations (the TS Structure can not access the TS Agent). Finally, the TS Agent is the TR's and PSs' interface to the TS Structure.

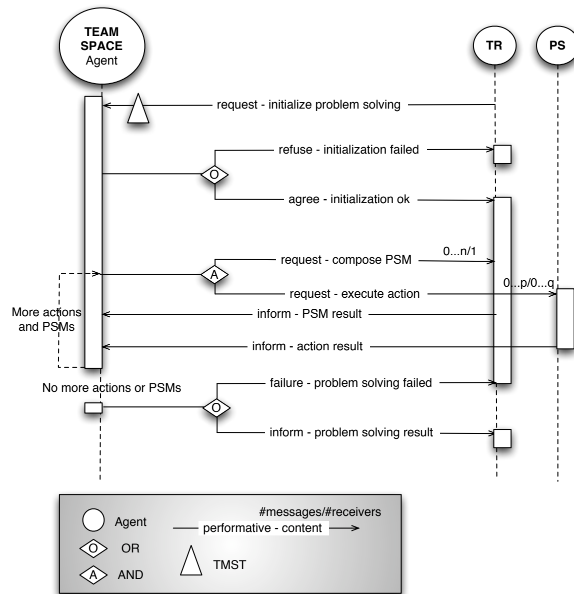

Figure 4.2: A protocol for collaborative work in CoPS. The protocol defines the messages exchanged between the TS Agent, PSs (Problem Solvers) and TR (Task Responsible) during problem solving in the TEAM SPACE.

The messages exchanged between TS Agent, TR and PSs during the problem solving process are represented as an interaction protocol in figure 4.2. This is an interaction protocol for the collaborative work in CoPS. The protocol is composed of different instances of the FIPA request protocol. Thus, we label the messages part of our protocol with FIPA performatives.

4.1.2 TEAM SPACE Agent

The general agent architecture

All of the different CoPS agents, including the TS Agent have a *generic agent architecture* consisting of three modules, which are: *Behavioral Control Unit*, *Communication Module* and *Knowledge Base*, as illustrated in figure 4.3. The Behavioral Control Unit controls the agent's behavior, by deciding what is the next action that the agent should perform. As communicating is an action, the Behavioral Control Unit will also instruct the communication module, about when to send a message, and what to do about incoming messages.

When the Behavioural Control Unit handles an incoming message, the content (knowledge) kept by that message is added to the Knowledge Base's *Working Memory*. The Working Memory keeps knowledge about the current activities of the agent, like knowledge about the current problems being solved. Knowledge about the agent's capabilities are stored in the Long Term Memory.

The knowledge stored in the Knowledge Base represents the state of an agent. A type of states called *internal processes*, indicate what the next proper

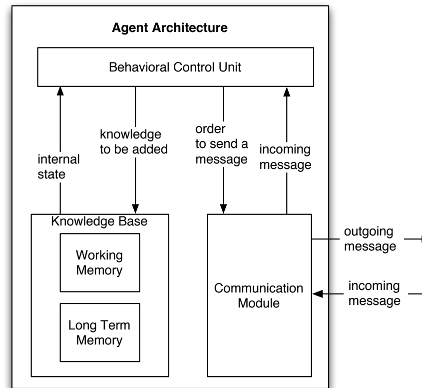

Figure 4.3: The general agent architecture used by all of the CoPS agents, and also by the TEAM SPACE (TS) Agent. The components of the architecture are represented by boxes, and the interactions between them with arrows.

behaviors for an agent should be. The transitions between these states are guided by different pieces of knowledge in the Knowledge Base. The different agent types (Task Responsible, Problem Solver, TS Agent, etc.) have different internal process states and transitions between these states. Next, the behavior of the TS Agent is elaborated in the context of its corresponding *process state diagram*. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows, see figure 4.4.

TEAM SPACE Agent Behaviour

The internal processes of the TS Agent are illustrated in figure 4.4. These processes make up the TS Agent's behaviour. Indicators for the TS Agent behaviour are found in the TEAM SPACE interactions, described in section 4.1.1.

When the TS Agent enters the CoPS environment, it starts registering its services at the *Matchmaker*. Next, the TS Agent waits for a request about performing its service. When a request is accepted it uses the TMST attached to this request to initialize the TS Structure. The TS Agent initializes a TS Structure by instantiating a new TS Structure component, providing it with the TMST and adding it to the list of TS Structure components, belonging to different problem solving teams.

If the TS Structure was initialized without failure, the TS Agent checks for actions and PSMs that are ready to be composed or executed, in the TS Structure. If there are any actions or PSMs, these are handled. The handling involves requesting TR to compose the PSMs, requesting the PSs to execute the actions and updating the TS Structure with the results from the requests. After updating the TS Structure the TS Agent checks for new actions or PMSs that are ready. If there are no more actions or PSMs that are ready to be executed or composed, the TS Agent ends the problem solving. At this state the TS Agent has to check if the TS Structure has a solution to the initial problem. If there

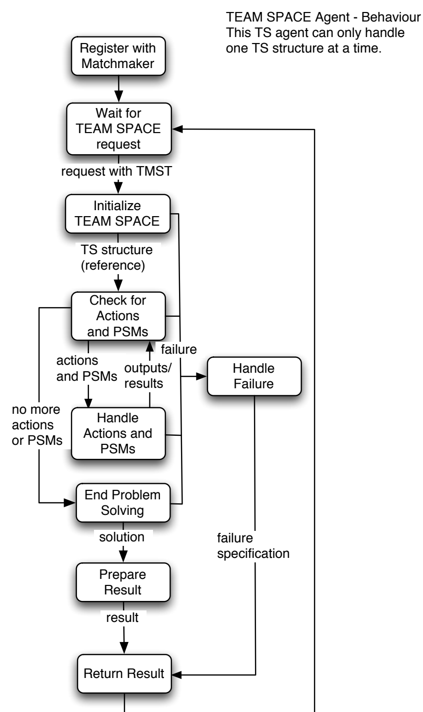

Figure 4.4: The TEAM SPACE (TS) Agent process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of the TS Agent. Here we suppose that the TS Agent only handles one TS Structure at a time.

is a solution, the result that is going to be returned to the TR is prepared. If everything goes as planned, the TS Agent behaviour ends in the *Return Result* - state, where the result is returned as an answer to the request that is accepted in the *Wait for TEAM SPACE request* - state. If a failure happens during some of the states, the TS Agent transits to the *Handle Failure* - state, where the failure is specified. Then, the TS Agent ends in the *Return Result* - state, only now the failure specifications is returned as a response to the initial request.

4.1.3 TEAM SPACE Structure Components

In subsection 4.1.1, we said that the TS Structure keeps track of the problem solving state and has functions for manipulating it. Interactions between the TS Agent and TS Structure also showed that the TS Agent had to use these functions in order to coordinate a team of agents in executing actions and composing PSMs. The TS Structure architecture is pictured in figure 4.5. Here we look at the different architectural components of this structure, and how these components interact to provide the functionality needed by the TS Agent.

TEAM SPACE (TS) Structure

The TS Structure is its own component containing a set set of other components. None of these other components can be accessed directly by the TS Agent. The TS Structure serves as their interface.

- **Content:** Plan Library, Rule Base Container, Result Library and Goal Stack. These components can only be reached through the TS Structure component
- **Functions:** TS Structure realizes the functions it provides through the other components. Detailed information about how functions are realized is presented in subsection 4.1.4. Functions provided by the TS Structure and their return values are:
  - **Initialize.** This function initializes the TS Structure with all of its components and returns the status of the initialization. The status describes if the TS Structure was properly initialized or not.
  - **Check for actions and PSMs.** This function executes the rules kept by the Rule Base Container (updates the TS Problem Solving State) to find actions ready to execute and PSMs ready to compose (extracts the TS Problem Solving State). If there are any actions or PSMs ready to be processed these are returned in a list. If there are no actions or PSMs ready to be processed, an empty list is returned. And if a failure occurs, a failure specification is returned. The function mainly has two parts:
    - **Update the TS Problem Solving State.**
    - **Extract the TS Problem Solving State.**
  - **Update the Result Library.** This function updates the TS Problem Solving State by adding new facts to the Result Library. The functions returns the status of the update. The status describes if the TS Problem Solving State was properly updated or if a failure did occur during the update.

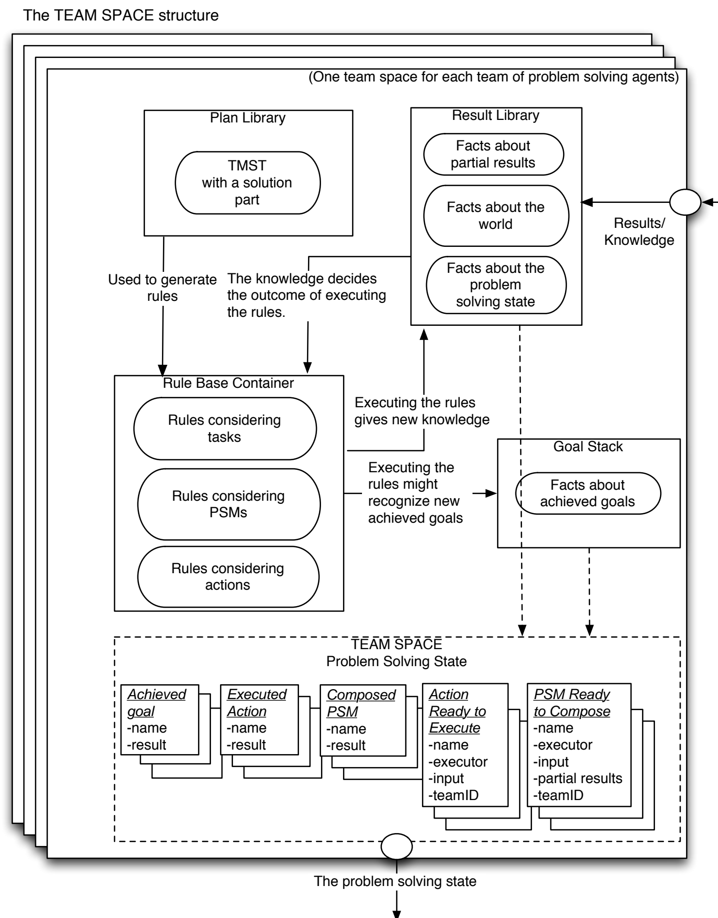

Figure 4.5: The architecture of the TEAM SPACE (TS) Structure. Rule Base Container keeps and manipulates a set of rules. Result Library and Goal Stack keeps and manipulates facts. Plan Library provides information stored in the TMST. TEAM SPACE (TS) Problem Solving State, represents a view of the facts stored in the Result Library and the Goal Stack.

  - **Get the solution.** This function returns the solution to the initial task, if it exists. If there is no solution this is also reported. The solution can be found from the TS Problem Solving State.

- **Accessed by:** The TS Agent accesses the TS Structure to update and extract necessary information about the TS Problem Solving State. TS Agent uses information about the TS Problem Solving State to coordinate a team of agents.

Plan Library

The Plan Library basically serves as the interface to the TMST.

- **Content:** A copy of the TMST that has all the necessary information about the problem to be solved. The TMST is placed in the Plan Library when the TS Structure initializes it.

- **Functions:** Extract different kinds of information from the TMST.

- **Accessed by:** TS Structure accesses the Plan Library during the *initialize* function by setting the TMST. The Rule Base Container generates a set of rules during the initialization. The rules that are generated are specialized for the problem being solved, and the Rule Base Container therefore access the Plan Library to get the necessary information from the TMST.

Rule Base Container

The Rule Base Container has a rule base and functions to manipulate the rules.

- **Content:** A set of rules. The rules are categorized into three types: Rules considering tasks, rules considering PSMs and rules considering actions. The rules are generated by the Rule Base Container from information in the TMST, when the TS Structure is being initialized. Information from the TMST includes: Node attributes (name, input, output, etc.), relationships between sub-nodes and super-nodes, and I/O dependencies between subtasks of a PSM.

- **Functions:** Generate and execute rules. Detailed description of how the Rule Base Container converts the information from the TMST into rules, and how the rules are executed is described after this introduction to the TS Structure components.

- **Accessed by:** TS Structure accesses the Rule Base Container during the *initialize* function, to make it generate a set of rules. TS Structure also accesses the Rule Base Container during the *check for actions and PSMs* function, to make it execute the rules. By executing the rules the TS Problem Solving State is updated.

Result Library

The Result Library has a knowledge base with facts represented by using specific templates, and functions to manipulate those facts.

- **Content:** A list of facts, which evolves during the problem solving process, and a set of templates. The templates describe how the different types of facts should be represented, and are added to the Result Library during the initialization of the TS Structure. The first fact is added during the initialization of the TS Structure, and represents the initial input to the problem solving process.
- **Functions:** Extract certain types of facts, add new facts, remove old facts and add templates.
- **Accessed by:** TS Structure accesses the Result Library during the *initialize* function to make it add a set of templates and the initial input to the list of facts. TS Structure also accesses the Result Library during the function *update the Result Library* - facts about the result of executed actions and composed PSMs are added. The Rule Base Container uses the facts stored in the Result Library when executing its rules. Facts in the Result Library are used to evaluate the left hand side conditions of a rule, and if evaluated to be true, the rule fires. All of the rules kept by the Rule Base Container are evaluated during an execution, and the rules that fire may add new facts to the Result Library.

Goal Stack

The Goal Stack also has a knowledge base with facts represented using one specific template, and functions to manipulate those facts.

- **Content**: A list of facts representing goals part of the TMST that have been achieved until the current state of the problem solving process, and a template describing how the goal - facts should be represented. The template is added during the initialization of the TS Structure.
- **Functions:** Extract a certain fact, add new facts and add a template.
- **Accessed by:** TS Structure accesses the Goal Stack during the *initialize* function to make it add a template. Some of the rules in the Rule Base Container decide when a goal is achieved. When one of these rules fires a fact about the achieved goal is added to the Goal Stack.

The TEAM SPACE (TS) Problem Solving state.

The TS Problem Solving State is not a component, but rather a view of facts kept by the Result Library and the Goal Stack. The state is made up of fulfilled goals attributed with name and result, executed actions attributed with name and result, composed PSMs attributed with name and result, actions ready to execute attributed with name, executor, input and teamID, and PSMs ready to compose attributed with name, executor, input, partial results and teamID. The TS Problem Solving State changes when new facts are added or removed from the Result Library and Goal Stack.

4.1.4 TEAM SPACE (TS) Structure Functions

Here, detailed information about the TS Structure functions and how they are realized by the different components of the TS Structure are given.

Initialize

After the TS Agent has created a TS Structure component with a TMST, it calls the *initialize* function in the TS Structure with the TMST as an input. The TS Structure component initializes itself and its components, using the information in the TMST. Finally, it returns the status of the initialization.

The TS Structure is initialized through the initialization of its components. The Plan Library is initialized as the TS Structure provides it with the TMST. Besides the information about the decomposition of the initial task, the TMST has these attributes: team ID, address of the requester (Task Responsible) of the TS Structure, and initial input to the problem solving process. The Result Library is initialized by the TS Structure in two stages. First, the TS Structure tells the Result Library to define a set of templates describing how facts should be presented. And second, the TS Structure gets "the initial input to the problem solving process" from the Plan Library and tells the Result Library to add this information. The Result Library translates "the initial input to the problem solving process" into a fact represented on the form of one of the templates. The Goal Stack is initialized by the TS Structure telling it to define a template describing how its facts about achieved goals should be represented.

Templates in the Goal Stack and Result Library are pre-defined and generic. They may be used to represent facts used in any problem solving process, where the initial problem (or task) is decomposed by a TMST. The generic templates look like this (the first template is used in the Goal Stack, and the rest of them in the Result Library):

```
(achieved
    (slot goal)
    (slot output))

(taskinput
    (slot name)
    (slot value))

(taskoutput
    (slot name)
    (slot value))

(actioninput
    (slot name)
    (slot value))

(actionoutput
    (slot name)
    (slot value))

(psminput
    (slot name)
    (slot value))

(psmoutput
    (slot name)
```

```
    (slot value))

(ready_to_execute
    (slot action)
    (slot action_type))
    (slot input)
    (slot executor))

(ready_to_compose
    (slot psm)
    (slot input)
    (multislot partial_results)
    (slot executor))

(executed
    (slot action)
    (slot output))

(composed
    (slot psm)
    (slot output))
```

The first string in the constructs is the name of the template or the concept it defines. Names following *slot* are attributes of the concept. And finally, names following *multislot* refer to a list of attributes. Slot-values of the templates may contain plain strings or strings represented in a predefined ontological knowledge representation. Meaning that the slot-values are specific to the domain of the problem being solved. Slot-values refer to information stored in the TMST. The templates, and therefore the problem solving, are domain independent, but they are restricted to our way of solving the problem. Therefore these templates should reflect a CoPS Ontology (not defined in this thesis).

The last step of the initialization, is the most complicated one. Then the TS Structure tells the Rule Base Container to generate a set of rules, that will be used to coordinate the problem solving process. The Rule Base Container uses the information in the TMST, kept by the Plan Library, to generate the rules.

As is described before, the TMST has actions, PSMs and tasks, each task has a goal. PSMs describe how a task (the goal of the task) is accomplished. When the TMST has a solution, the nodes part of the solution are tagged. Each tagged PSM has a list of subtasks, which results need to be composed. There are I/O dependencies between these PSM subtasks which are described in the PSM. Each tagged task only has one PSM or action. A tagged action is a directly executable PSM. Information needed by the Rule Base Container, for it to generate the rules, is extracted from the different tagged nodes of the TMST by the Plan Library, like described in table 4.2.

There have been made some assumptions to simplify the the generation of rules, owing to the fact that the CoPS ontology is not prioritized. These assumptions should be considered removed later. We assume that the output of an action or a PSM directly serves as the output of the task that this action or PSM accomplishes. Further, we assume that the input of a task directly serves

Table 4.2: Information needed from each of the nodes (action, PSM, task) in the TMST to make the rules in the Rule Base Container.

Node in the TMST	Piece of information
action	name
	type
	inputname
	input/input value
	output name
	executor
PSM	name
	input name
	input value
	output name
	executor
	subtasks
	I/O dependencies among subtasks
task	goal (name)
	input name
	input value
	output name
	PSM/action that accomplishes the task (sub-node)

as the input of its sub-node, either an action or a PSM. And finally, we assume that the input of a PSM serves directly as input of at least one of its sub-tasks. These assumptions give directions for the different rule constructs.

There are different types of rules, divided into three classes. The first class has rules instantiating *rule constructs considering tasks*, the second class has rules instantiating *rule constructs considering PSMs*, and the third class has rules instantiating *rule constructs considering actions*. The rule constructs are skeletons (templates), describing general rules. Rule constructs are built from the templates defining how facts should be represented in the Result Library and Goal Stack. Rule constructs are pre-defined and generic, like the templates. And these constructs may be instantiated for any problem solving process, where the initial problem (or task) is decomposed by a TMST.

Fact templates define concepts with slot-values related to the TMST. When a rule construct is instantiated, slots in the fact templates are filled in with the correct values, from the TMST. The *rule instantiations* specializes the Rule Base Container for assisting in solving the task, decomposed by the TMST. Instantiation of all the necessary rules are done by traversing the tagged nodes part of the TMST solution. The traversing starts with the initial task. Instantiation of rule constructs has the same meaning as generating the rules.

When the Rule Base Container starts generating the rules, it first gets the name of the initial task of the TMST from the Plan Library, and generates rules related to this task. It then gets the name of the task's PSM or action sub-node. If the task's sub-node is a PSM, the Rule Base Container generates rules related to that PSM, and then gets the names of the PSM's sub-tasks. Next, rules related to each of these sub-tasks are generated. The tagged nodes

in the TMST are traversed until all of the actions are reached, and rules related to these actions are generated.

When a task is reached during the traversing of tagged nodes in the TMST, the Rule Base Container uses the task name to get the additional information about that task, listed in table 4.2, from the Plan Library. This information includes the name of the PSM or action that is the sub-node of the current task. Some of the rules considering a task also need information about that task's sub-node. Thus, additional information about the PSM or action is also gotten from the Plan Library. Then the Rule Base Container instantiates all of the rule constructs considering tasks, with the information gotten from the Plan Library. The rules are kept by the Rule Base Container. Next, rules related to the task's sub-node (an action or PSM) is generated. Rule constructs considering tasks are:

```
(rule-number
        (taskoutput (name xoutput) (value ?out))
        (test (neq ?out nil))
=>
        (assert (achieved (goal yname) (output ?out))))
```

Meaning: When task x has an output value, it means that the goal y of this task is reached. The fact that the goal y is achieved and has the output value of task x is added(asserted) to the Goal Stack.

```
(rule-number
        (actionoutput (name xoutput) (value ?out))
        (test (neq ?out nil))
=>
        (assert (taskoutput (name youtput) (value ?out))))
```

Meaning: This rule is only made if the task is linked to an action. When action x has an output value, we add a fact to the Result Library saying that task y has an output value which is the same as the output of the action x. Action x performs task y.

```
(rule-number
        (taskinput (name xinput) (value ?in))
        (test (neq ?in nil))
=>
        (assert (actioninput (name yinput) (value ?in))))
```

Meaning: This rule is only made if the task is linked to an action. When task x has an input value, we add a fact to the Result Library saying that action y has an input value which is the same as the input of the task x. Action y performs task x.

```
(rule-number
        (psmoutput (name xoutput) (value ?out))
        (test (neq ?out nil))
=>
        (assert (taskoutput (name youtput) (value ?out))))
```

Meaning: This rule is only made if the task is linked to a PSM. When PSM x has an output value, we add a fact to the Result Library saying that task y has an output value which is the same as the output of the PSM x. PSM x performs task y.

```
(rule-number
        (taskinput (name xinput) (value ?in))
        (test (neq ?in nil))
=>
        (assert (psminput (name yinput) (value ?in))))
```

Meaning: This rule is only made if the task is linked to a PSM. When task x has an input value, we add a fact to the Result Library saying that PSM y has an input value which is the same as the input of the task x. PSM y performs task x.

When a PSM is reached during the traversing of tagged nodes in the TMST, the Rule Base Container uses the PSM name to get the additional information about that PSM, listed in table 4.2, from the Plan Library. This information includes the name of the sub-tasks of the current PSM. Some of the rules considering a PSM also need information about that PSM's sub-tasks. Thus, information about the sub-tasks is also gotten from the Plan Library. Then the Rule Base Container instantiates all of the rule constructs considering PSMs, with the information gotten from the Plan Library. The rules are are kept by the Rule Base Container. Next, rules related to the PSM's sub-tasks are generated, for one task at a time. Rule constructs considering PSMs are:

```
(rule-number
        (taskoutput (name xoutput) (value ?out))
        (test (neq ?out nil))
=>
        (assert (taskinput (name yinput) (value ?out))))
```

Meaning: This rule is added when there are I/O -dependencies between the sub-tasks of a PSM. One rule is added for each pair of dependent tasks. When task x has an output value, we add a fact to the Result Library saying that task y has an input value which is the same as the output of the task x. Task y can not be achieved before task x has an output(solution).

```
(rule-number
        (taskoutput (name x1output) (value ?out1))
        (test (neq ?out1 nil))
        (taskoutput (name x2output) (value ?out2))
        (test (neq ?out2 nil))
        ...
        (taskoutput (name xnoutput) (value ?outn))
        (test (neq ?outn nil))
=>
        (assert (ready_to_compose
                (psm yname)
```

```
        (partialresults ?out1 ?out2 ... ?outn)
        (executor agentname))))
```

Meaning: When all of PSM y's subtasks (x1 - xn) has an output value, we add a fact to the Result Library saying that PSM y is ready to be composed, and the partial results are set to be the output values from the subtasks.

```
(rule-number
    (composed (psm xname) (output ?out))
    (test (neq ?out nil))
    ?fact <- (ready_to_compose (psm xname))
=>
    (assert (psmoutput (name xoutput) (value ?out)))
    (retract ?fact))
```

Meaning: When PSM x is composed, we add a fact to the Result Library saying that the output value of PSM x is the same as the output value from the composition. The rule also sees that the fact saying that PSM x is ready to compose is removed from the Result Library.

```
(rule-number
    (psminput (name xinput) (value ?in))
    (test (neq ?in nil))
=>
    (assert (taskinput (name yinput) (value ?in))))
```

Meaning: When PSM x has an input value, we add a fact to the Result Library saying that the input value of task y is the same as the input value of PSM x. The rule is created if the information in the TMST says that the input value of PSM x is the same as the input value of task y, which is one of the PSM's sub-tasks.

When an action is reached during the traversing of nodes in the TMST, the Rule Base Container uses the action name to get the additional information about that action, listed in table 4.2, from the Plan Library. Then the Rule Base Container instantiates all of the rule constructs considering actions, with the information gotten from the Plan Library. The rules are kept by the Rule Base Container. Rule constructs considering actions are:

```
(rule-number
    (actioninput (name xinput) (value ?in))
    (test (neq ?in nil))
=>
    (assert (ready_to_execute
        (action xname)
        (action_type xtype) (input ?in)
        (executor agentname))))
```

Meaning: When action x has an input value, we add a fact to the Result Library saying that action x is ready to be executed. The input value of action x is set as the input value of the added fact.

```
(rule-number
        (executed (action xname) (output ?out))
        (test (neq ?out nil))
        ?fact <- (ready_to_execute (action xname))
=>
        (assert (actionoutput (name xoutput) (value ?out)))
        (retract ?fact))
```

Meaning: When action x is executed, we add a fact to the Result Library saying that the output value of action x is the same as the output value from the execution. The rule also sees that the fact saying that action x is ready to be executed is removed from the Result Library.

After the Rule Base Container has rules for all of the tagged nodes of the TMST, the initialization of the Rule Base Container is done. And thus, the initialization of the TS Structure is finally done as well. If the information kept by the TMST is correctly defined, these rules will guide the problem solving process in an effective manner until the solution to the initial task (goal) is found.

Check for Actions and PSMs

When the TS Agent calls the *check for actions and PSMs* function in the TS Structure, the TS Structure starts *updating the TS Problem Solving State*. The TS Structure tells the Rule Base Container to execute the rules. When the Rule Base Container executes the rules, all of the rules generated during the initialization is matched against the facts stored in the Result Library.

A Rule's left-hand side lists a set of conditions that has to be true for the rule to fire. These conditions are represented using the same templates as are used in the Result Library. For a condition to be true, there has to be a fact in the Result Library that matches this condition. And both the fact - and the condition representation must be using the same template. When a rule is matched against the facts in the Result Library, all of the conditions on its left hand side must evaluate to be true, for this rule to fire.

A rule's right-hand side lists a set of consequences that are realized if the rule fires. These consequences are represented using the templates in the Result Library or the Goal Stack. If we look at the rule constructs described before, we see that a rule's right hand sides define two different consequence-types: *(assert (fact))* and *(retract (fact))*. The assert-consequence means that a fact should be added, and the retract-consequence means that a fact should be removed. The facts are added or removed from the Result Library or the Goal Stack, depending on which templates are used.

Thus, when the Rule Base Container executes the rules and some of the rules fire, the Rule Base Container tells the Result Library or the Goal Stack to modify their lists of facts by adding or removing facts. As the facts stored in the Result Library and the Goal Stack at any time defines the TS Problem Solving State, the TS Problem Solving State is now updated.

When the TS Problem Solving State is updated, the TS Structure moves on to the next step of the *check for actions and PSMs* function. This step involves *extracting information from the TS Problem Solving State*. More pre-

cisely this means finding out if there are new actions or PSMs that are ready to be processed. The TS Structure tells the Result Library to extract a facts on the form: *(ready_to_execute (action action_name) (action_type type) (input input_value) (executor agent_name))*, *(ready_to_compose (psm psm_name) (partialresults result1 result 2 result3) (executor agent_name))*. The Result Library looks for those facts, and translates them into lists of actions and PSMs (with attributes), and returns them to the TS Structure. The list is finally returned by the TS Structure to the TS Agent.

Update the Result Library

When the TS Agent has received the results from the PSs' execution of actions and TR's composition of partial results, it translates them into a list of actions and PSMs (with the results as attributes). The TS Agent calls the TS Structure function *update the result library* with the list as an input. The TS Structure forwards this list to the Result Library and tells it to add the information as new facts. The Result Library translates the information in the list into facts. The facts are represented by using the pre-defined templates. Then, these facts are added to the Result Library. The facts may look like this: *(executed (action action_name) (output output_value))* and *(composed (psm psm_name) (output output_value))*. By adding these facts to the Result Library, the TS Problem Solving State has changed, and it may cause new rules to fire next time the rules are executed.

Get the solution

When the TS Agent tries to get the solution to the initial task from the TS Structure it calls the *get the solution* function. The TS Structure asks the Plan Library for the initial goal. Then, the TS Structure asks the Goal Stack to get the output of the initial goal by providing it with the goal name. The Goal Stack searches through its facts for facts on the form *(achieved (goal goal_name) (output output_value))*, where the goal slot-value matches the name of the initial goal. The output slot-value is returned as the solution. The value that is returned to the TS Structure, is the solution to the initial goal, and at the same time the solution to the initial task, which the goal is related to in the TMST. The TS finally structure returns the solution to the TS Agent.

4.2 An Example

In this section we see how a problem is solved in the TEAM SPACE, step by step. It is the same example as introduced in chapter 3. Remember that the agents in the system are: PA, TR, DEC, and the PSs - PS-LA, PS-LSA, PS-PDBA and PS-RA. This example focus on the steps 8-11 in the example given in chapter 3. The steps are also describing the TEAM SPACE interfaces, pictured in figure 4.1. A new agent is introduced in the system, which is the TS Agent.

In this example, every time the TS Structure get the Rule Base Container to execute the rules we show the TEAM SPACE problem solving state. Here we list all the facts present in the Goal Stack - and Result Library that contribute to the TS Problem Solving State. The numbers initiating each described step of the problem solving corresponds to the interactions in figure 4.1.

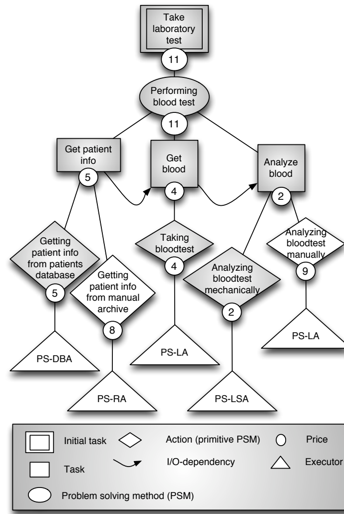

Figure 4.6: The TMST showing the solution, marked with shaded nodes. The lowest cost suggested for actions or PSMs propagates upwards in the tree-structure. Actions are connected to executors (Problem Solvers, PSs)- that has offered to execute them for the price attached to the corresponding action.

1)The TR sends a request to the TS Agent to initialize the TS Structure. Attached to this request is the TMST in figure 4.6. Each of the nodes in the TMST has the attributes listed in chapter 3 - table 3.1.
2)The TS Agent creates a TS Structure component. And calls the function *initialize* on the TS Structure with the TMST as an input. The TS Structure uses the information of the TMST to initialize for problem solving. The TMST is set in the Plan Library, the Result Library and Goal Stack is initialized with templates, and the rules are generated by the Rule Base Container using the domain specific knowledge stored in the TMST and the structure of the TMST itself. The "initial input to the problem solving process" is also translated into the fact *(taskinput (name "takeLaboratoryTest_input) (value "(Patient :name Paul)"*, and added to the Result Library.
3) The TS Agent checks for actions ready to execute and PSMs ready to compose (for the first time after the initialization), by calling the *check for actions and PSMs* function of the TS Structure.
4) The TS Structure gets the Rule Base Container to execute the rules, and the TS Problem Solving State is updated:

- Fulfilled goals: (no)
- Executed actions: (no)
- Composed PSMs: (no)
- Actions ready to Execute:
  (ready_to_execute (action "gettingPatientInfoDB") (input "(Patient :name

Paul)") (executor "PS-PDBA") (teamID "1"))

- PSMs ready to compose: (no)

The action ready to execute (gettingPatientInfoDB), with its information, is returned to the TS Agent.
7-9-10) The TS Agent requests PS-PDBA to execute gettingPatientInfoDB, and submits the attributes in the request. PS-PDBA executes the action and returns the result. TS Agent asks the TS Structure to update the TS Problem Solving State, by calling the function *update the Result Library* with the result as an input. The TS Structure makes sure that this fact is added to the Result Library: *(executed (action "gettingPatientInfoDB") (output "(Patient :name Paul :into Information about Paul)")).* Then, the TS Agent checks for actions ready to execute and PSMs ready to compose (for the second time), by calling the *check for actions and PSMs* function of the TS Structure.
4) The TS Structure gets the Rule Base Container to execute the rules, and the TS Problem Solving State is updated:

- Fulfilled goals:
  (achieved (goal "getPatientInfo_goal") (output "(Patient :name Paul :info Information about Paul)"))
- Executed actions:
  (executed (action "gettingPatientInfoDB") (output "(Patient :name Paul :info Information about Paul)"))
- Composed PSMs: (no)
- Actions ready to Execute:
  (ready_to_execute (action "takingBloodtest") (input "(Patient :name Paul :info Information about Paul)") (executor "PS-LA") (teamID "1"))
- PSMs ready to compose: (no)

The action ready to execute (takingBloodtest), with its information, is returned to the TS Agent.
7-9-10) The TS Agent requests PS-LA to execute gettingPatientInfoDB, and submits the attributes in the request. PS-LA executes the action and returns the result. TS Agent asks the TS Structure to update the TS Problem Solving State, by calling the function *update the Result Library* with the result as an input. The TS Structure makes sure that this fact is added to the Result Library: *(executed (action "takingBloodtest") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)")).* Then, the TS Agent checks for actions ready to execute and PSMs ready to compose (for the third time), by calling the *check for actions and PSMs* function of the TS Structure.
4) The TS Structure gets the Rule Base Container to execute the rules, and the TS Problem Solving State is updated:

- Fulfilled goals:
  (achieved (goal "getPatientInfo_goal") (output "(Patient :name Paul :info Information about Paul)"))
  (achieved (goal "getBlood_goal") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))

- Executed actions:
  (executed (action "gettingPatientInfoDB") (output "(Patient :name Paul :info Information about Paul)"))
  (executed (action "takingBloodtest") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))

- Composed PSMs: (no)

- Actions ready to Execute:
  (ready_to_execute (action "analyzingBloodMech") (input "(Patient :name Paul :info Information about Paul :blood_sample_ reference 3456)") (executor "PS-LSA") (teamID "1"))

- PSMs ready to compose: (no)

The action ready to execute (analyzingBloodMech), with its information, is returned to the TS Agent.
7-9-10) The TS Agent requests PS-LSA to execute analyzingBloodMech, and submits the attributes in the request. PS-LSA execute the actions and returns the result. TS Agent asks the TS Structure to update the TS Problem Solving State, by calling the function *update the Result Library* with the result as an input. The TS Structure makes sure that this fact is added to the Result Library: *(executed (action "analyzingBloodMech") (output "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))*. Then, the TS Agent checks for actions ready to execute and PSMs ready to compose (for the fourth time), by calling the *check for actions and PSMs* function of the TS Structure.
4) The TS Structure gets the Rule Base Container to execute the rules, and the TS Problem Solving State is updated:

- Fulfilled goals:
  (achieved (goal "getPatientInfo_goal") (output "(Patient :name Paul :info Information about Paul)"))
  (achieved (goal "getBlood_goal") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))
  (achieved (goal "analyzeBlood_goal") (output "(Patient :name Paul :info Information about Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))

- Executed actions:
  (executed (action "gettingPatientInfoDB") (output "(Patient :name Paul :info Information about Paul)"))
  (executed (action "takingBloodtest") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))
  (executed (action "analyzingBloodMech") (output "(Patient :name Paul :info Information about Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))

- Composed PSMs: (no)

- Actions ready to Execute: (no)

- PSMs ready to compose:
  (ready_to_compose (psm "performingBloodTest") (input "(Patient :name Paul)") (partial_results "(Patient :name Paul :info Information about Paul)" "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)" "(Patient :name Paul :info Information about Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))") (executor "TR"))

The PSM ready to compose (performingBloodTest), with its information, is returned to the TS Agent.
6-8-10) The TS Agent requests TR to compose performinBloodTest, and submits the attributes in the request. TR composes the partial results from the PSM's subtask and returns the final result. TS Agent asks the TS Structure to update the TS Problem Solving State, by calling the function *update the Result Library* with the result as an input. The TS Structure makes sure that this fact is added to the Result Library: *(composed (psm "performingBloodTest") (output "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))*. Then, the TS Agent checks for actions ready to execute and PSMs ready to compose (for the fifth time), by calling the *check for actions and PSMs* function of the TS Structure.
4) The TS Structure gets the Rule Base Container to execute the rules, and the TS Problem Solving State is updated:

- Fulfilled goals:
  (achieved (goal "getPatientInfo_goal") (output "(Patient :name Paul :info Information about Paul)"))
  (achieved (goal "getBlood_goal") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))
  (achieved (goal "analyzeBlood_goal") (output "(Patient :name Paul :info Information about Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))
  (achieved (goal "takeLaboratoryTest_goal") (output "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))

- Executed actions:
  (executed (action "gettingPatientInfoDB") (output "(Patient :name Paul :info Information about Paul)"))
  (executed (action "takingBloodtest") (output "(Patient :name Paul :info Information about Paul :blood_sample_reference 3456)"))
  (executed (action "analyzingBloodMech") (output "(Patient :name Paul :info Information about Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))

- Composed PSMs:
  (composed (psm "performingBloodTest") (output "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))

- Actions ready to Execute: (no)

- PSMs ready to compose: (no)

An empty set of PSMs to compose and actions ready to execute is returned to the TS Agent.
5-11-12) Since there are nothing more to do, the TS Agent tries to get the solution, by calling the function *get the solution* in the TS Structure. The TS Structure finds that the initial task "takeLaboratoryTest" is solved, since the TS Problem Solving State has the fact: *(achieved (goal "takeLaboratoryTest_goal") (output "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))"))*. This fact exists in the Goal Stack, and indicates that the goal of the initial task is accomplished. The output of the achieved goal is returned as the solution for the TS Agent.
13) Finally, the TS Agent returns the solution, "(Patient :name Paul :test_results (TestResult :description NA - normal, Glucose - high, HB - low))", to the TR.

4.3 TEAM SPACE Architecture Additions

Here we look at additions to the core TEAM SPACE architecture. In section 4.3.1 we describe how several problem solving teams may solve their problems at the same time in parallel TEAM SPACEs. Section 4.3.2 explains how failures are handled and reported by the TEAM SPACE Architecture. And finally, section 4.3.3 mentions the problem of re-planning if any unanticipated events occur during the problem solving process, and how this may be solved by the TEAM SPACE architecture.

4.3.1 Parallel TEAM SPACEs

Providing parallel TEAM SPACEs in our architecture can be done in two different ways. In the first way, illustrated by figure 4.7, there are only one TS Agent. In the second way, illustrated by figure 4.8, there are several TS Agents. Some explanations to these figures follow.

In figure 4.7 we have only one TS Agent, which assists in solving three different problems (tasks) at once. There has to be one TR for each of the problems. The TR knows about the *teamID* and the *TMST* which decomposes the initial task. The PSs can join several problem solving teams at the same time. In our figure one PS joins team 1 and team 2, and the other PS joins team 1 and team 3. A PS knows about which actions it has committed to perform in each of the teams it joins.

When the TS Agent receives requests from a TRs to initialize a TS Structure, the TS Agent creates a new TS Structure component and calls the *initialize* function, for each of these requests. Each TS Structure component is associated with the requesting TR's team ID, which is stored in the TMST attached to that TR's request. The TS Agent handles problem solving in all of the created TS Structures at the same time by "running its behaviour in parallel". Exactly what part of the behaviour that runs in parallel, and how this works, is shown in figure 4.9. This figure is almost the same as figure 4.4. The difference is that, after receiving a number of requests for for initializing a TS Structure, the behaviour splits into as many parallel behaviours as the number of requests (or the number of TS Structures). When all of the requests are readily processed (*Done*), the TS Agent can accept new requests for initializing a TS Structure.

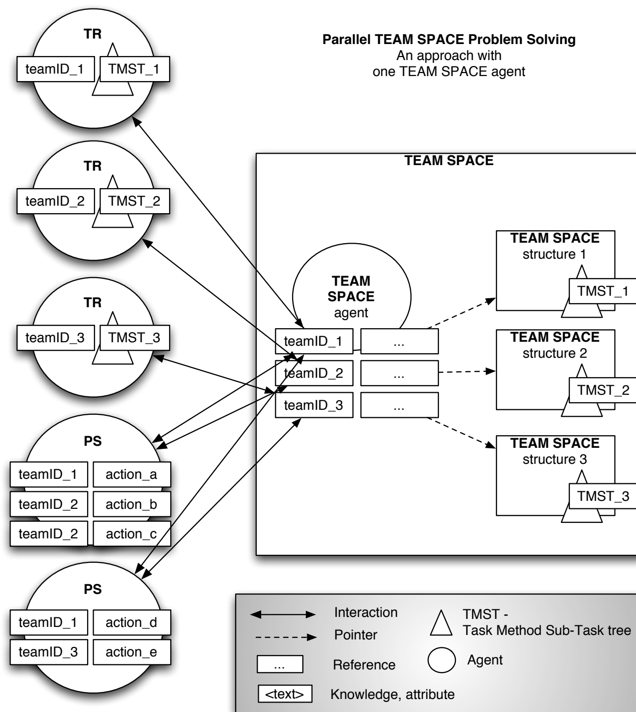

Figure 4.7: Parallel problem solving in the TEAM SPACE with a single TEAM SPACE (TS) Agent. The TS Agent has references to a set of TEAM SPACE (TS) Structures, and it interacts with several problem solving teams at once. The problem solving teams are comprised of Problem Solvers (PSs) and a Task Responsible (TR).

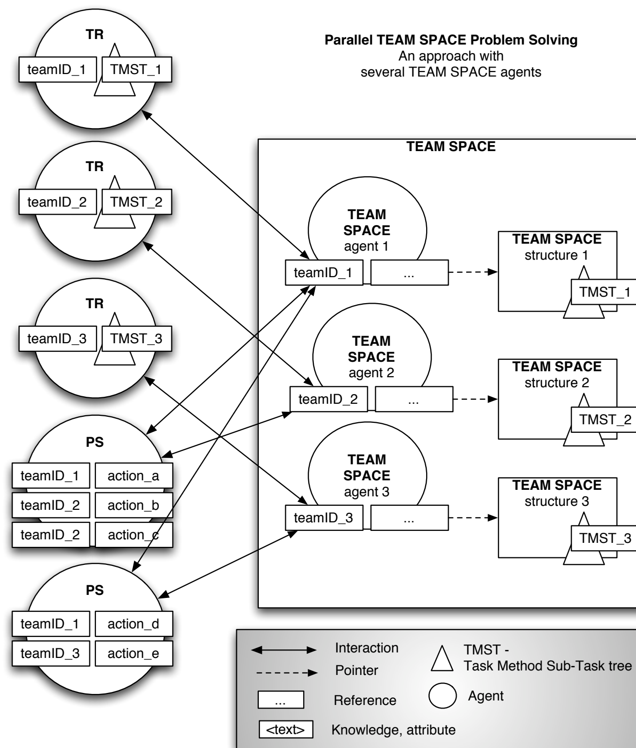

Figure 4.8: Parallel problem solving in the TEAM SPACE with multiple TEAM SPACE (TS) Agents. Each TS Agent only has reference to one TEAM SPACE (TS) Structure, and it interacts with one problem solving team at a time. The problem solving team is comprised of Problem Solvers (PSs) and a Task Responsible (TR).

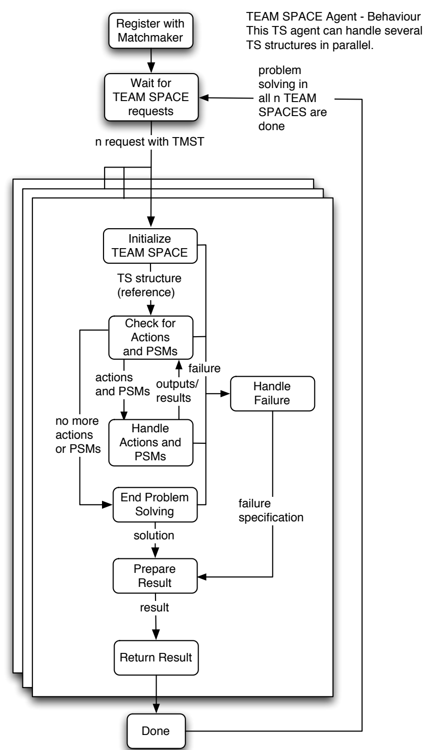

Figure 4.9: The TEAM SPACE (TS) Agent process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of the TS Agent. Here we suppose that the TS Agent may handle several TS Structures at a time.

During the problem solving, the TS Agent interacts with all of the PSs. When a PS receives a request to perform an action, it verifies that he has committed to performing this action by looking at its teamID - action tuples. The messages between the PSs and the TS Agent and the messages between the TRs and the TS Agent all are tagged with the teamID. Thus, the TS Agent knows which TS Structure to update after a received message, and the PSs and TRs can verify if the messages they receive from the TS Agent really are meant for them.

In figure 4.8, the same three problems (tasks) are solved, as in figure 4.7. This is a more decentralized approach to parallelism, since we have one TS Agent for each problem solving team. The PSs interact with different TS Agents, if they join several teams at once. This approach is appropriate if the problems to be solved are large (many levels of decomposition in the TMST) and therefore demands many interactions between the TS Agent and the PSs, and between the TS Agent and the TRs.

The two different approaches to parallel problem solving in the TEAM SPACE may also be combined.

4.3.2 Failure Handling

It is the TS Agent that discovers and handles failures during the problem solving in a TS Structure. Figure 4.4 and 4.9 illustrates that the TS Agent has a set of behavioural states that may transit to the state *Handle Failure*. During these states the TS Agent interacts with the TS Structure, PSs and TR. Failures may be discovered in the return values from TS functions or in messages received from PSs and TR. Next, we describe failures that may occur in the different behavioural states of the TS Agent:

- **Initialize TEAM SPACE.** In this state, the TS Agent calls the *initialize* function of the TS Structure. If the TS Structure cannot be properly initialized, this is returned. And the TS Agent transits to the state *Handle Failure*. In this case the failure is specified and returned in the state *Return Result* as a refuse message (part of the FIPA request protocol) to the TR. This refuse message is a response to the TR's request of initializing the TEAM SPACE.

- **Check for Actions and PSMs.** In this state, the TS Agent calls the *check for actions and PSMs* function of the TS Structure. If a failure occurs while the TS Structure executes this function, a failure specification is returned. And the TS Agent transits to the state *Handle Failure*. The failure specification from the TS Structure function is handled and returned to the TR in the state *Return Result*. The return message this time is a failure message (part of the FIPA request protocol), which is the result notification to the TR's request of initializing the TEAM SPACE.

- **Handle Actions and PSMs.** In this state the TS Agent requests TR to compose PSMs and PSs to execute actions that are ready to be processed. If any of these agents return a refuse message or a failure message, the TS Agent transits to the *Handle Failure* state. If everything goes well with the processing of PSMs and actions, the TS Agent calls the *update the Result Library* function of the TS Structure. This function returns the

status of the update. If the return value states that the update failed, the TS Agent transits to the *Handle Failure* state. The failure is handled, and a specification is returned for the TR in a failure message (like described for the previous state).

- **End Problem Solving.** In this state the TS Agent calls the *Get the solution* function of the TS Structure. If the function reports that there are no solution, the TS Agent transits to the *Handle Failure* state. The failure is handled, and a specification is returned for the TR in a failure message (like described over).

How the different failures are handled is not proposed here, and will be suggested for future work in chapter 8. For now, when a TS Agent discovers a failure, the problem solving in the corresponding TS Structure is terminated, and the failure specification is returned to the TR that requested the TS Structure in the first place. The handling of failures are closely related to dynamics and re-planning that is shortly introduced in the next subsection.

4.3.3 Dynamics and Re-Planning

This part of the architecture is not going to be implemented. But, here we describe how the TEAM SPACE architecture may be useful for the purpose of dynamics and re-planning during the problem solving process.

The pre-existing CoPS architecture proposes that a team should be reconfigured if agents change their minds about joining the team, are disconnected or fail in any other way (this was not implemented in previous work though). If a PS fails to execute its action, the TR should allocate this action to another PS. If there are no available PSs that have the capability of executing the action, the task connected to the action has to be re-planed. Re-planning the task involves modifying the solution of the TMST.

If a PS fails to execute its action, this is recognized by the TS Agent, as described in the previous subsection. The TS Agent transits to its *Handle Failure* state, and it has the knowledge of which PS that did fail, and which action the PS failed to solve. This is what the TS Agent could do to assist the TR in re-allocating the action or re-planning the task of this action:

- TS Agent halts the problem solving process and requests the TR to allocate the action to another PS.

- If the TR achieved to find another PS to execute the action, this PS is invited to join the team and is informed about the team ID. The name of the PS is returned to the TS Agent. The TS Agent then makes sure that the rules which involve the failing PS and the action that it failed to solve, are updated with the information about the new PS. In addition the Result Library containing a fact defined by the template *ready_to_execute*, representing the failed action is updated: the value of the *executor* slot is replaced with the name of the new PS. Then the TS Agent may start the problem solving process again.

- If the TR did not find another PS to execute the action, the TS Agent may help in re-planning the task of the action that failed. Then the TS Agent get the TS Structure to find out if the Plan Library may find any other

solution to that task in the TMST. The Plan Library extracts information from the TMST and may find that the task could be accomplished by executing another action. The TS Agent requests the TR to allocate this action to a PS.

- If the TR achieved to find another PS to execute the action, this PS is invited to join the team and is informed about the team ID. The name of the PS is returned to the TS Agent. Now the TS Agent makes sure that the rules which involve the failing PS and the action that it failed to solve, are updated with the information about the new PS, in addition to the information about the new action. The Result Library is also updated. Now the *ready_to_execute* fact representing the failed action, is removed and replaced by a fact representing the new action. Then the TS Agent may start the problem solving process again.

If the task that has the failing action cannot be achieved at all, a bigger part of the TMST must be re-planned. This may be done by the TS Agent in almost the same way, only modifying a bigger set of rules in the TS Structure Rule Base Container and facts in the TS Structure Result Library.

The rules in the Rule Base Container and the facts in the Result Library and Goal Stack may be seen as an executable version of the TMST. Earlier in this subsection we state that re-planning a task involves modifying the solution of the TMST. The solution of the TMST was used to generate the rules in the Rule Base Container and to initialize the Result Library and Goal Stack in the first place. This means that manipulating rules and facts in the TS Structure can be seen as modifying the solution of the TMST.

At any time the collection of facts represent the current TS Problem Solving State. The "executable version of the TMST" brings us from one TS Problem Solving State till another. When, for example, a PS fails to execute an action, the TS Problem Solving State is "stuck". Then the TS Agent cooperates with the TR to find a way out of the problematic TS Problem Solving State, using the TS Structure. Rules are manipulated and facts are modified, removed and replaced until the facts represent a TS Problem Solving State that is no longer "stuck", and that conforms to one of the possible solutions of the TMST. In this way, the TEAM SPACE may add dynamics to the team work of CoPS agents.

4.4 Other Extensions to the CoPS Architecture

The TEAM SPACE is the main extension to the pre-existing CoPS architecture. Other extension to the CoPS architecture were proposed in chapter 3. We have chosen to leave the CoPS Ontology out here, because it is not that important considering the objectives of this thesis. What is important though, is to make sure that the Problem Solver (PS) and the Task Responsible (TR) are fit to join in the *solving of subtasks* - and *integrating partial solutions* process steps using the TEAM SPACE.

CoPS Problem Solver

The former behaviours of the Problem Solver (PS) is shown in figure 4.10, and the new version is shown in figure 4.11. The newer version of the PS can have

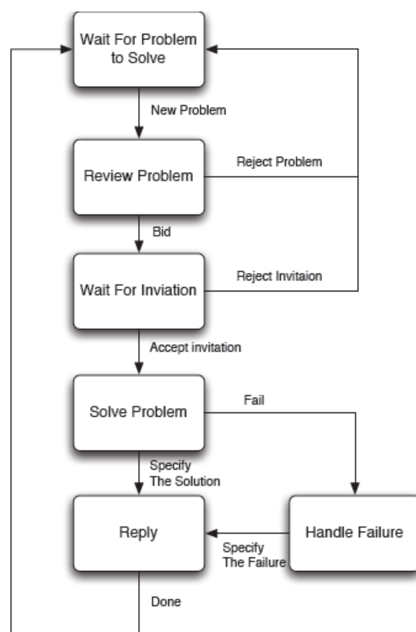

Figure 4.10: The **former version of the Problem Solver (PS)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of PS. Taken from [34].

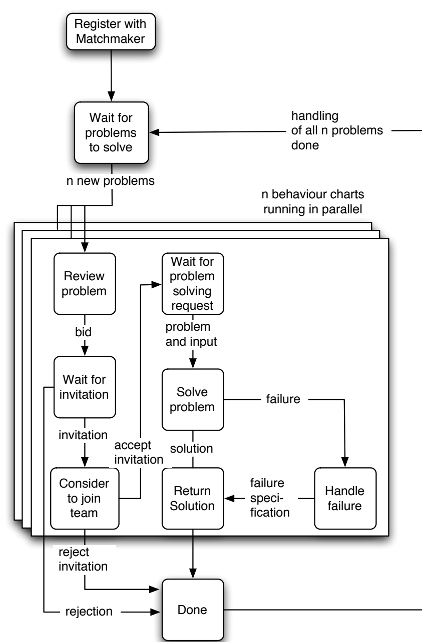

Figure 4.11: The **new version of Problem solver (PS)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR.

several capabilities (not a single one, like before), meaning that it can perform a variety of action types. It is also able to attend to several conversations at the same time, meaning that it can perform multiple actions simultaneously. Finally, it is extended with behavioural processes that manage the problem solving in the TEAM SPACE. The older version of PS is described in [34, 33], and the newer version is described next.

When the PS enters the CoPS environment it registers with the Matchmaker. A PS registers one or more capabilities, reflecting which action(s) it is capable of performing, or which problems it can solve. Then, it waits to receive problems to solve from TRs. When the PS has received some problems (requests to perform an action) it splits its behaviour into as many parts as the number of problems, and proceeds with parallel behaviours. The different problems are processed simultaneously, but in the following we look at the processing of one separate problem. All of the other problems are processed in the same way.

A problem is reviewed to see if it is interesting, and if the PS has the capabilities of solving it. If the PS decides to solve the problem (perform the action), it returns a bid with a price for the solution to the TR that requested a solution to that problem . If the TR accepts the bid of the PS, the PS gets an invitation (with a team ID) to join a problem solving team, if not the PS gets a rejection and is done processing the problem. When an invitation is received, the PS considers to join the team. If the invitation is rejected (the PS do not want to join the team) the PS is done, and if the invitation is accepted (the PS wants to join the team) the PS starts to wait for a problem solving request from the TS Agent to start solving the problem. A problem solving request from the TS Agent, contains information about the problem to be solved (action to execute), input information to this problem (knowledge needed to solve it), and a team ID indicating which problem solving team (and TMST) this problem belongs to. The PS uses the team ID and the information about the problem to verify that it actually is part of the team that the team ID identifies and that it has agreed to solve that problem. Then the PS solves the problem (executes the action), the solution is returned to the TS Agent, and the PS is done processing this problem. If a failure occurs while the PS tries to solve the problem, the failure is handled, and a failure specification is returned to the TS Agent as the "solution" of the problem. Then the PS is done processing this problem.

When all of the other problems also are readily processed and done, the PS starts waiting again for new problems to solve.

CoPS Task Responsible

The former behaviours of the CoPS Task Responsible (TR) is shown in figure 4.12, and the new version is shown in figure 4.13. The newer version of the TR is extended with behavioural processes that manage the problem solving in the TEAM SPACE. The older version of the TR is described in [34, 33], and the newer version is described next.

When the TR enters the CoPS environment it registers with the Matchmaker. It registers its capabilities of administrating the problem solving process. Then, it waits to receive problems (tasks) to solve. When a new problem arrives, the TR must review the problem to check if it is understood and accepted. When the problem is accepted, the TR has to get the decomposition (TMST) from the Decomposer (DEC). After the TMST is received from the DEC,

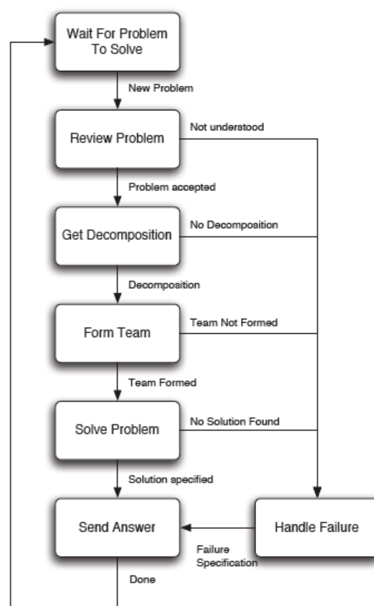

Figure 4.12: The **former version of Task Responsible (TR)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR. Taken from [34].

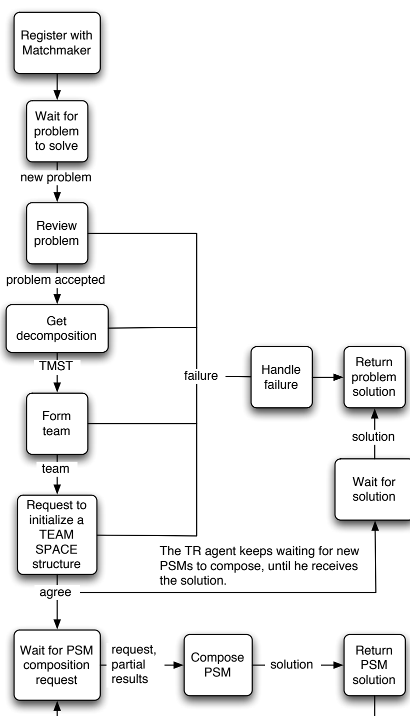

Figure 4.13: The **new version of Task Responsible (TR)** process state diagram. Internal process states are illustrated by boxes, while the possible transitions between them are illustrated by arrows. This process state diagram reflects the behaviours of TR.

the TR can form the problem solving team, based on this TMST (described in chapter 3). When a team is formed, the TR sends a request to initialize a TS Structure for the TS Agent. If the TS Agent agrees to do this, the TR has two processes running at the same time. In the first process, it waits for PSM composition requests from the TS Agent, and when such a request is received the TR composes the PSM and returns the solution for the TS Agent. In the second process, the TR waits for a solution (to the initial task/problem) from the TS Agent. As soon as a solution is received, the TR stops waiting for further PSM composition requests, and returns the problem solution for the requester of the problem. If any failures occur during the problem solving, the failure is handled and specified, and returned to the requester of the problem.

4.5 How the TEAM SPACE Architecture Meets the Functional Requirements

This section describes how the proposed architecture covers the functional requirements that were listed in chapter 3. Each of the requirements are repeated here, followed by description of how it is covered by the TEAM SPACE architecture proposed in this chapter:

1. *The TEAM SPACE must keep a structure for storing partial results from both the PSs and TR.*

   All partial results from the PSs and TR are kept by the TS Structure, in the Result Library. Partial results from the PSs' execution of actions and the TR's composition of PSMs, are sent to the TS Agent. The TS Agent calls the *update the Result Library* function in the TS structure with the results as an input. This function makes sure that the results are added as facts in the Result Library.

2. *The TEAM SPACE must be able to reason about the partial results and the current problem to be solved. So that it always knows which actions are ready to be executed, which partial results are ready to be integrated, and which goals are met. Said in another way, the TEAM SPACE, at any time, must keep the correct state of the problem solving process.*

   Reasoning in the TEAM SPACE architecture, is provided by the TS Structure Rule Base Container. The Rule Base Container is initialized when it instantiates a set of generic rule constructs by using the information in the TMST. The TMST has specific information about the problem that is being solved, and thus the generated rules are specialized for that problem. When the rules are executed, facts stored in the Result Library decide which rules that fire. The rules that fire, add new facts to the Result Library and Goal Stack. The Result Library may also be updated when TS Agent receives partial results from the PSs and TRs. The TS Problem Solving State depend on certain types of facts in the Result Library and Goal Stack. These facts include amongst others, actions that are ready to be executed, which partial results are ready to be integrated (PSMs ready to be composed) and which goals are met (achieved) (see figure 4.5).

The rules in the Rule Base Container "reason about" the partial results in the Result Library, updating the facts in the Result Library and Goal Stack. The state of the problem solving process (TS Problem Solving State) can at any time be extracted from the facts kept in the Result Library and Goal Stack.

3. *The TEAM SPACE must be able to communicate the correct information,efficiently, to the PSs and the TR (the problem solving team).*

   The TEAM SPACE is comprised of two main components: TS Agent and TS Structure. The TS Structure represents the shared memory or working space. And the TS Agent, is the interface between the TS Structure and the problem solving team. The TS Problem Solving State, represented by the TS Structure, at any time has information about what information that needs to be communicated to the PSs or TR part of a problem solving team. The TS Agents uses the information about the TEAM SPACE Problem Solving State to decide when to communicate information to the PSs and the TR, and thus the correct information is communicated efficiently.

4. *The TEAM SPACE must be able to handle the information received from the PSs and TR, and update the state of the problem solving process accordingly.*

   It is the TS Agent that first receives the information from the PSs and TR. Interactions between PSs and the TS Agent are realized with a FIPA-request protocol, and so is the interactions between TR and the TS Agent. In addition the messages sent in these interactions use the same communication language. This means that a TS Agent can understand the information received from the PSs and TR. When the TS Agent receives a message it translates this message into a form (objects representing actions, goals and PSMs) that the TS Structure understands, and asks the TS Structure to update the TS Problem Solving State. The TS Structure then tells the Result Library to add the new information. The Result Library translates the information into facts, using the generic templates and adds the fact to its list. As mentioned before, when new facts are added to the Result Library or Goal Stack, the TS Problem Solving State changes.

5. *The TEAM SPACE must handle concurrent problem solving, by allocating a separate working area for each problem solving team.*

   The TEAM SPACE architecture proposed in this chapter, has two approaches to handling concurrent problem solving in parallel TS Structures. These were described in section 4.3.1. The first approach involves a TS Agent having a list of TS Structures, handling all of them in parallel, using parallel behaviour. The second approach solves the concurrency by introducing several TS Agents into the CoPS environment, where each of them facilitates one problem solving team with a TS Structure.

6. *The TEAM SPACE must be able to convert the knowledge stored in the TMST, so that it could be used to initialize the working area for a problem solving team.*

   When the TS Agent receives a request to initialize a TS Structure from the TR, it creates a TS Structure component. Next, the TS Agent calls the *initialize* function on the TS Structure with the TMST with a TMST as an input. The TS Structure initializes itself by using the information in the TSMT considering the problem to be solved. The TS Structure is initialized through the initializations of its components.

   The Plan Library is initialized as the TS Structure provides it with the TMST. The Plan Library is than able to extract knowledge from the TMST, that is needed by the other TS Structure components. The Result Library is initialized by the TS Structure in two stages. First, the TS Structure tells the Result Library to define a set of generic templates describing how facts should be represented in the Result Library. The templates define concept from the TMST, and assists the Result Library in converting TMST-related knowledge into facts. And second, the TS Structure gets "the initial input to the problem solving process" from the Plan Library and tells the Result Library to add this information. The Result Library translates the information into a fact represented on the form of one of the templates. The Goal stack is initialized by the TS Structure telling it to define a template describing how facts about achieved goals should be represented in the Goal Stack. And finally the TS Structure tells the Rule Base Container to initialize, by generating rules. The rules are generated by instantiating generic rule-constructs with knowledge from the TMST, provided by the Plan Library.

   When the initialization is done: 1) The Plan Library is able to extract necessary information from the TMST, 2) Templates have been defined, that helps the TS Structure components to store knowledge related to the TMST, 3) Rules have been generated that "automates" the process of solving the problem (or task) composed by the TMST, and 4) The "initial input to the problem solving process", stored in the TMST has been translated into a fact, and stored in the Result Library.

The initialization of the TS Structure, that involves adapting the TS Structure to a specific TMST is the most extensive and complex function of the TS Structure. When the initialization is done, the problem solving process steps: *solving of subtasks* and *integration of partial solutions*, is "automated" by the TS Structure. The remaining work after the initialization involves, updating (executing rules, adding partial results) and extracting (checkin for actions and PSMs, getting the solution) information from the TS Problem Solving State.

Failure handling in the TEAM SPACE, discussed in subsection 4.3.2, is indirectly part of the functional requirements. Dynamics and re-planning supported by the TEAM SPACE, discussed in subsection 4.3.3, is not part of the functional requirements. The subsection is added to demonstrate how the TEAM SPACE architecture can be used for the purpose of team formation dynamics and re-planning of tasks in future extensions of the CoPS framework.

4.6 Summary

In this chapter a TEAM SPACE architecture has been proposed, which conceptualizes a shared memory structure, used by Task Responsible (TR) and Problem Solvers (PSs) during the problem solving process steps: *solving of subtasks* and *integration of partial solutions*. A summary of the TEAM SPACE architecture was provided in section 4.5. That section described how the proposed TEAM SPACE architecture covers all the functional requirements that were listed in chapter 3. In addition to the TEAM SPACE architecture, the extensions to the PS and TR part of the CoPS framework were described in this chapter.

The TEAM SPACE architecture and the extension to the CoPS framework are implemented in the CoPS framework prototype. Implementation details are provided by the next chapter. In chapter 7 we describe how the CoPS framework prototype was used to implement part of an application that realizes a real-domain problem in a health center scenario.

Chapter 5

Comparison of CoPS to Relevant Work

The TEAM SPACE architecture proposed for the problem solving in CoPS, make CoPS a blackboard-like architecture for cooperative problem solving. This solution differs from a traditional blackboard architecture, since CoPS is a multi-agent framework. The TEAM SPACE architecture represents a shared memory or workspace for the CoPS agents, using ideas from the blackboard paradigm. This part of CoPS, facilitates the collaborative problem solving. In this chapter we compare CoPS to other relevant work. Focus of the relevant work is collaborating modules using shared memories or workspaces.

In section 5.1 criteria for the comparison are described. In section 5.2 four different systems (architectures) that all involve collaborating modules using a shared repository are introduced. And finally, in section 5.3 a comparison is made between these four systems and the CoPS framework, using the chosen criteria.

5.1 Criteria for the Comparison

In chapter 2 multiagent systems and blackboard systems were described separately. Blackboard systems were the first attempt at integrating "collaborating" software modules. Work with blackboard systems has given rise to some ideas, like shared memories, that later have been used in the development of multiagent systems. Multiagent system research approaches the collaborating software paradigm from an agent-centric orientation. The goal of both blackboard - and multiagent systems is to achieve effective collaboration with a group of independent software entities [7]. That is also a goal in CoPS. CoPS intends to offer a FIPA - compliant framework for developing agents engaged in distributed collaborative problem solving.

The different kinds of relevant work introduced in the next section, all have something to do with collaborating groups of software entities, and they all use some kind of shared repository. First, a blackboard architecture is described. Here a set of experts implemented as knowledge sources in a blackboard system, collaborate to achieve a fixture design. Second, a blackboard based multiagent system is described. This system supports concurrent engineering projects. A

set of agencies (groups of intelligent agents) collaborate to assist the people working on a design project, in the project management process. Agencies also make sure that the project follows the product oriented process steps described on a blackboard. Results produced during the execution of a project are stored on the blackboard, in relation to their corresponding process steps. Third we introduce an architecture which uses a blackboard in combination with the publish/subscribe method. And finally, an architecture for an e-commerce system is presented. In this architecture agents collaborate to buy and sell goods on the behalf of their users. Buyer and seller agents find each other through a shared space that implements the publish/subscribe method.

In [7] Corkill lists some key challenges involved in creating effective collaborating software systems. These challenges are met differently by the many applications of such systems, and therefor serve as good criteria for comparing the CoPS architecture to the four systems introduced above. The criteria are:

1. **Representation** - getting software modules to understand one another.

2. **Awareness** - making modules aware when something relevant to them occurs.

3. **Investigation** - helping modules to quickly find information related to their current activities.

4. **Interaction** - creating modules that are able to use the concurrent work of others while working on a shared task.

5. **Integration** - combining results produced by other modules.

6. **Coordination** - getting modules to focus their activities on the right things at the right time.

CoPS can be seen as collaborating software. But, it is neither a full-fledged multiagent architecture nor a full-fledged classic blackboard architecture. CoPS is a multiagent architecture using a shared workspace - the TEAM SPACE, for collaborative problem solving. In the development of the TEAM SPACE, ideas from blackboard systems and the publish/subscribe model were used.

5.2 Relevant work - Systems Using Shared Repositories

In this section, four different systems are introduced. This work is relevant to what has been done in this master thesis, in different ways. First, all of the systems, like CoPS, are consisting of collaborating software modules. Second, they all use some form of shared repository. In addition, the systems in different ways use general architectures or methodologies that are adopted by, or has inspired the work with CoPS: The multiagent architecture, the blackboard architecture and the publish/subscribe method.

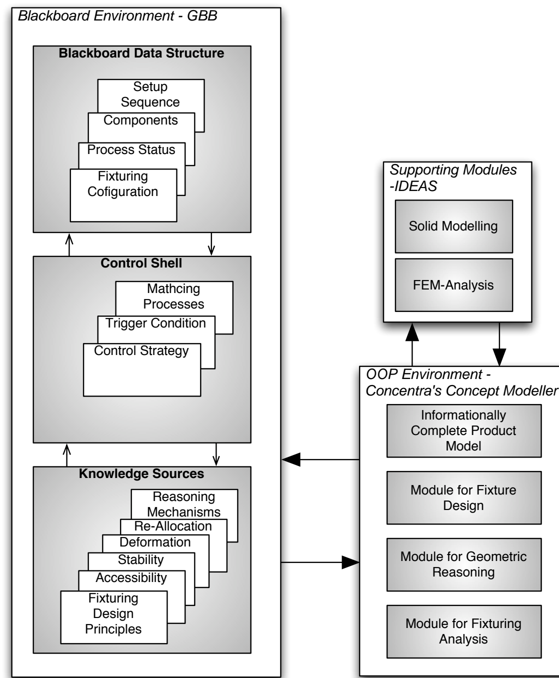

Figure 5.1: The architecture of the cooperative fixture design system, modified from [28].

5.2.1 Application of a Blackboard Framework to a Cooperative Fixture Design System

In [28] Roy and Liao propose an application of a blackboard framework to a cooperative fixture design system. Fixture design is a complicated process which needs knowledge about a number of design issues including work-piece configuration, manufacturing processes involved, machining environment, etc.

Introduction

The knowledge needed to perform a fixture design is divided between different designers (experts) working together, sharing knowledge, and performing a variety of design tasks. When performing a generic fixture design task, the designers use a variety of engineering-related models such as functional, equational, analytical and geometric models. The proposed cooperative fixture design system, supports communication between knowledge sources (experts), and it accommodates different modeling paradigms.

An architecture for cooperative problem solving consisting of a blackboard control system and several independently executing domain experts has been developed. The implementation of the architecture has been carried out in a generic blackboard framework, GBB [17, 9]. The *Blackboard Environment*, depicted in figure 5.1 consists of three major components: the *Knowledge Sources*

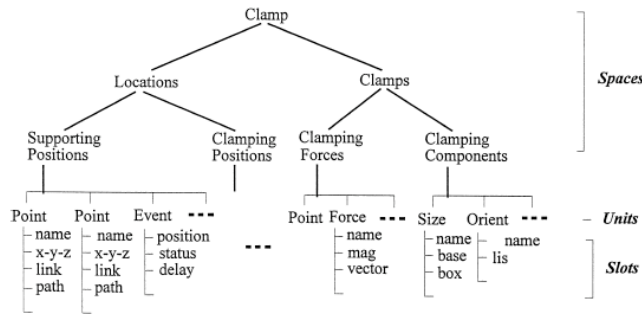

Figure 5.2: Hierarchical structure of the blackboard, from [28].

(KSs), the *Blackboard Data Structure* and the *Control Structure*. The blackboard environment also consults with other supporting modules. The blackboard system assists in the problem solving process, and the supporting modules provide guidance in the actual design activities. Components of the architecture belonging to the *Blackboard Environment*, are described next, more detailed information about the supporting modules can be found in [28].

The Knowledge Sources

Knowledge sources (KSs) keep the domain and control knowledge needed for problem solution. Each KS, can be written as procedures, collections of rules (IF-THEN), algorithms, reasoning methodologies, or design constraints as logic assertion. KSs are represented separately and independently as functional modules in the blackboard environment, and are the only contributers to the evolving problem solution in the problem solving process. The problem solution involves making a decision about a specific fixture design.

A number of KSs are developed to cover different general functions of the fixture design system, which are grouped into *Re-Allocation* and *Reasoning Mechanisms* (see 5.1). In addition several domain KSs are developed to cover the whole fixture design process, these are grouped into: *Fixture Design Heuristics* - KSs who know about fixture design principles, *Stability* - KSs who know how to make sure the work-piece (object for the fixturing design) keep stable during the fixturing process, *Accessibility* - KSs who know how to avoid interference and who know about how to satisfy geometric constraints, and finally *Deformation* - KSs who considers work-piece deformation during fixturing-setup and process operation.

The Blackboard Data Structure

The *Blackboard Data Structure*, provides a hierarchical organization of data. This data involves information and solutions considering a fixturing design which are generated during the problem solving process. Figure 5.2 illustrates an example of a hierarchical, blackboard structure created in the system. There are two basic components in the blackboard structure, *spaces* and *units*. Spaces divide the whole blackboard data structure into tree-like directories for keeping

objects called units. The spaces and units in a blackboard structure are like directories and files in a file system. The properties of units are determined by their various attributes, or *slots*. During the problem solving process, information (resident in slots) associated with objects (units) on one level serves as input to a set of KSs, which place new information on the same or other levels in turn. The objects and their properties define the vocabulary of the solution space.

The Control Shell

The KSs respond opportunistically to changes on the blackboard. The *Control Shell* (in figure 5.1) monitors the changes on the blackboard and decides what actions to take next. GBB, and thereby the fixturing design system, uses "event driven processing" as the interface between the blackboard and the control shell. An event is defined as any change in the blackboard due to new data or KS actions. Events are immediately made available to the control shell which responds by either activating a KS and putting it in a queue, or invoking execution of a KS activation.

The solution is built one step at a time. Any reasoning process(es) can be applied at one process step to achieve local solutions. As a result, the sequence of KS execution becomes dynamic and opportunistic rather than fixed and preprogrammed. Criteria are provided to determine when to terminate the process.

Implementation of the Cooperative Fixture Design System in the GBB Environment

How the architecture of the cooperative fixture design system (in figure 5.1) is implemented in the GBB environment, is shown in figure 5.3. The KSs are grouped into four main functional (domain) knowledge bases (KBs): *Design Heuristics, Stability, Accessibility* and *Deformation* (described before). All domain KSs are managed by a GBB administrator which controls the activation of and communication between the KSs. Hence the GBB administrator does the work of the *Control Shell*. The administrator performs two kinds of processes in the system: (i) activation of domain KBs according to the status of the current blackboard data and, (ii) modification of the current blackboard data according to the execution results of the KBs.

The administrator always consults with all domain KBs whenever there is any modification (for fixturing configurations) on the blackboard. This allows the system to consider appropriate fixturing configuration(s) according to all functional requirements expressed in individual domain KBs. Any modifications to an intermediate fixturing configuration (contributed by the domain KBs) are regularly posted on the blackboard by the administrator.

Each domain knowledge base has a basic structure including a pre-processor, a set of domain KSs and a re-allocation KS, and an interface mechanism. When a domain knowledge base is activated by the GBB administrator, the preprocessor determines which KSs should be activated. The activating conditions are based on the content of current blackboard data (data on the processed fixturing configuration). The interface mechanism deals with communication between the domain KBs and the GBB administrator.

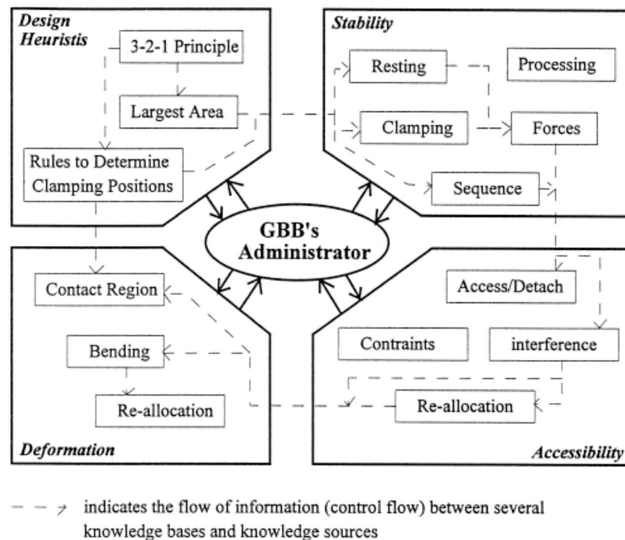

Figure 5.3: Implementation of the system architecture in the GBB environment, from [28].

The use of the System for a Fixture Design Process

The system starts out with a work-piece. The goal of the KSs is to collaboratively generate the clamping configurations (including clamping positions, an optimal clamp set-up sequence, and clamping forces) of this work-piece. The GBB administrator starts by activating an initial KS to retrieve all required information regarding work-piece geometry, available clamping areas, cutting tool paths, cutting forces, and clamping components. The GBB administrator then initiates the problem solving process by posting the initial fixturing configuration data on the blackboard. When the blackboard is initialized, the GBB administrator consults with all domain KBs to make them check the suitability and applicability of the preliminary fixturing configuration. Appropriate subset KSs are chosen by the pre-processors of their KBs, and are activated to add their modifications to the current configuration. The changes are communicated to the GBB Administrator through the KB's interface mechanism, and the GBB Administrator eventually posts the changes on the blackboard. The first level of searching paths is formed, as the fixturing configuration is adjusted according to the preferences and knowledge of the activated KSs. Next, all of the KBs are again consulted to check the suitability and applicability of the current fixturing configuration represented on the blackboard. The activated KSs provide new adjustments, and all of the KBs are consulted again.

The propagation of the problem solving process as it progresses through the intermediate stages of the blackboard, follows a tree-like searching path, like illustrated in figure 5.2. Based on the current blackboard data, the system generates the searching tree in a dynamic and opportunistic way. The system terminates the problem solving process when the KSs cannot suggest any other

modifications. The final status of the blackboard provides the information about the final clamping configuration.

5.2.2 A Blackboard-Based Multiagent System for Supporting Concurrent Engineering Projects

In [19] Kao, Su and Wang propose a framework of a blackboard-based multiagent system, called I2QFD (*Integrative Intelligent Quality Function Deployment*), to facilitate the communication and coordination of distributed design projects. To be more precise this framework enable timely sharing of information, support project planning and control, facilitate team collaboration and synchronize the design process.

Introduction

There is a dual view of the project's process: the *project management process*, which is about the description and the organization of work, and the *product-oriented process*, which is about the specification and the development of the product as a result of the project. The project management process is facilitated by a multiagent system, while the QFD (*Quality Function Deployment*)-embedded blackboard outlines and documents the product-oriented process. QFD is a comprehensive quality design method that seeks out customer needs, uncovers qualities that are important for the customer, translates these into design characteristics and actions, and finally builds and delivers a quality product [16].

The I2QFD Overall System

The overall I2QFD framework is illustrated in figure 5.4. As mentioned before, I2QFD is a framework for a blackboard-based multiagent system. The agents of the system is grouped into *IA modules* (or *Intelligent Agencies*). IA modules use the QFD-embedded blackboard, to coordinate the teamwork. In this way, the project management process and the product-oriented process of an engineering project are combined in the same system. The *IA modules* make sure that the project team members executes their tasks so that each stage of the project follows the QFD method. Team members use the blackboard, via the IA modules, to contribute ideas, submit works, respond to others' output and inspect current designs instantly from personal workstations. The blackboard has functionality to integrate knowledge, reason about it's content, support flexible control of the design process, and document the different tasks or stages of the engineering project.

QFD involves processes from several disciplines. Therefore, the global blackboard is parted into a number of distributed blackboards. Each of these blackboards are connected to the global *Knowledge-based System* (KBS) and to a local KBS. The local KBS is also accessible to the pertinent functional groups. This is not further discussed here.

The I2QFD Agency System

The agency system (agents) supports the project manager and facilitates the design team in complying with the QFD process. As presented in figure 5.4

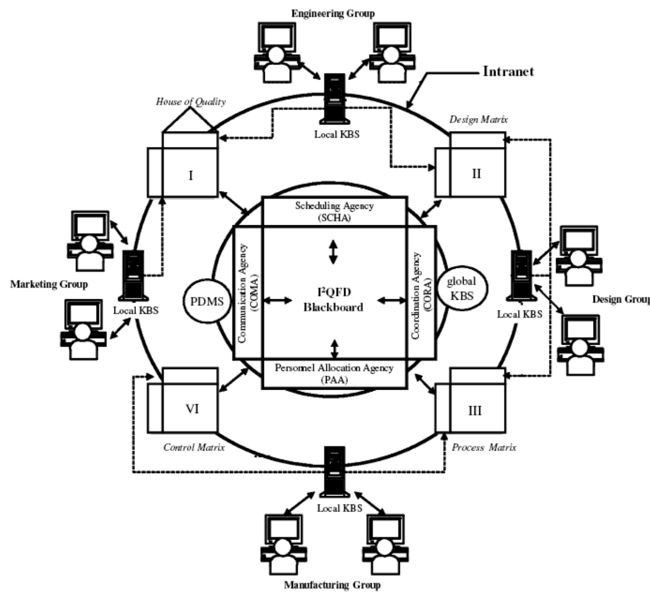

Figure 5.4: The blackboard-based I2QFD, from [19].

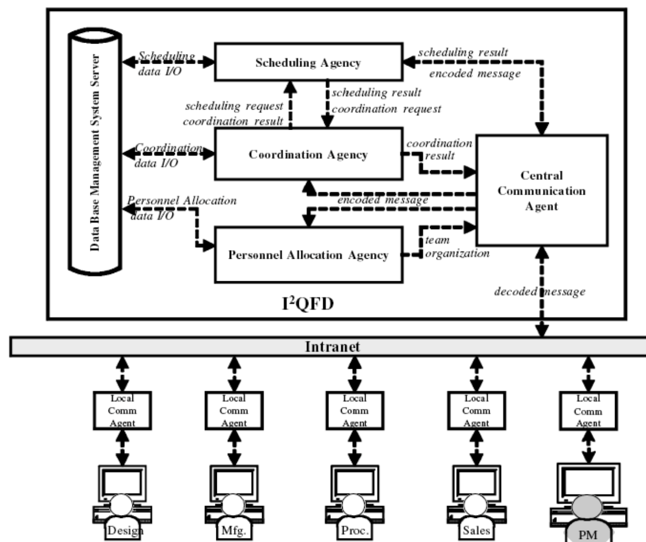

Figure 5.5: Agency system of I2QFD, from [19].

and figure 5.5, there are four IAs (*Intelligent Agencies*): the *Communication Agency*, *Personnel Allocation Agency*, *Scheduling Agency* and *Coordination Agency*. These work together while performing respective functions. *Knowledge Query and Manipulation Language* (KQML) is used as the communication protocol between IAs.

The *Communication Agency* consists of a central agent, which dispatches messages between local agents and other IAs, and a set of local agents, each of which serves an individual team member by transmitting messages between the member and the central agent. When a local agent receives a message in the KQML packet from the central agent, it will convert it into the specific format that is comprehensible to the local members.

The *Personnel Allocation Agency* assists the project manager in organizing the project team. At the beginning of the planning stage, the project manager defines a network of tasks part of the project. The agency uses this information to generate a relevant *Work Breakdown Structure* (WBS). The WBS represents the hierarchy and precedence of project tasks. Recruitment messages with descriptions of tasks are sent to the qualified experts. To respond to the recruitment, each interested expert sends a bid indicating discipline, skill level, committed available period and estimated time length fro processing the task of interest. Once there is a sufficient number of candidates for each task, the Personnel Allocation Agency will recommend the best project team that will complete the project within the shortest time.

The *Scheduling Agency* generates a feasible schedule for a project team that meets due date and budget constraints. During the project's initial stage, the Scheduling Agency estimates the project completion time of the alternative teams proposed by the Personnel Allocation Agency. Once the project starts to execute, the Scheduling Agency is capable of supporting the project manager to reschedule the project when unexpected events occur. Unexpected events might conflicts between tasks or the lack of critical resources.

Coordination involves managing interdependencies among activities. This is done to avoid redundant activities, provide sharing of information, and balance workload among agents. Specifically, the functions provided by the *Coordination Agency* include supporting the team members in performing their tasks in the right order, and solving conflicts to keep the design project on track.

The Use of I2QFD in the Project Process

I2QFD works as a supporting medium for the distributed project team, whereas the project manager and the team members are the true actors who should implement the project management process and product oriented process.

In a distributed, networked enterprise a project manager is responsible for a design project, which includes a set of tasks according to a defined WBS. To respond to the recruiting messages announced by the project manager, each expert may bid for more than one task. When sufficient bids are received, the project manager evaluates candidates with support from the Personnel Allocation Agency. The Personnel Agency makes a list of alternative team organizations. Then it sends a request to the Scheduling Agency to estimate project completion time with respect to each team organization. The schedule with the earliest completion time is chosen.

While the project is in progress, the design team carries out the assigned

tasks according to the preset schedule. During the project, the QFD-embedded blackboard evolves with information, as each team member is responsible for contributing useful information to the team. They may request and receive information for performing their tasks, from each other. The activities of an individual proceed in parallel with those of other members as well as various functions of the agency system. Information received by team members can be categorized as *request message* and *task assignment*. For instance, the Scheduling Agency may send a request to a team member for the progress report of an assigned task. In contrast, when working on the assigned task, there are two possible instances that require time-consuming activities. The first is that once a specific task is completed on schedule, its result will be sent to the Coordination Agency for further processing. Unless a design conflict is detected, the Coordination Agency will pass the result instantly to the next member(s) to start the follow-up processes. The second instance is that, when confronting a certain difficulty in completing the task on time, the member may request the project manager for assistance or additional information. With assistance from the Scheduling Agency and Coordination Agency, the project manager will identify the appropriate member(s) to assign additional task(s). Suppose the Coordination Agency detects design conflicts or a member failing to proceed with the task due to the preceding tasks performed by others, e.g. an unfit part's spec, the Coordination Agency will notify the source member(s) for modification. Once the present task is completed, the member becomes available for new assignment or may start to work on other projects.

5.2.3 A Blackboard Used for Collaborative Development of Interactive Robot

In [24] Matsusaka and Kobayashi propose an architecture for an environment, which enables the collaborative development of interactive robots. Collaboration between designers, engineers and other specialists is necessary in order to realize large-scale integrated systems.

Introduction

Most conventional approaches for the development of robots are based on the top-down design flow model; the chief engineer analyzes the problem and designs the global system framework before the module engineers develop the detailed design to satisfy the specifications. However, it is sometimes desirable that each component is designed and developed independently, but the behaviour of the combined system is often so complicated that it is impossible for the chief engineer to anticipate all of the problems. In such cases, the design process should be performed in parallel to assure flexibility. The module developers must analyze the problems from the viewpoint of their own specialties. This is called the bazaar-type development model. The aim of the study described in [24] was to develop an environment that supports this bazaar-type development model.

The experience of the module developer is denoted the Scope of Interest (SoI). Fields in which a module developer has sufficient knowledge to provide new information for the development of a module is denoted the Area of Profession (AoP). The act of disseminating this information is denoted commitment,

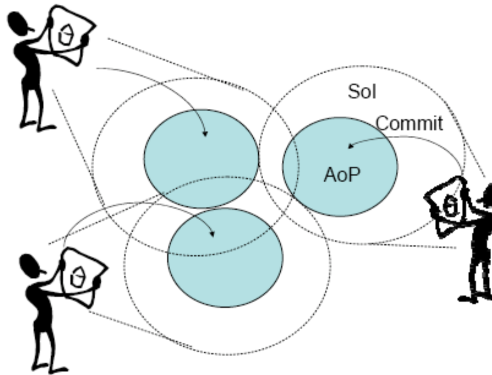

Figure 5.6: Basic composition and basic design process, from [24].

and the commitment of information requested from other developers is denoted collaboration. A schematic of this process is presented in figure 5.6. The shared workspace at the center of the system is called the Blackboard.

How the Blackboard is Used in a Development Process

The basic design process using the blackboard is as follows:

1. **Examine the blackboard.** A developer can gain an overview of all the information by looking at the blackboard.
2. **Select all useful information.** The developer selects all useful information within his or her SoI from the blackboard and uses it to implement the new module.
3. **Implement the individual module.** Implement the module that generates new information belonging to the developers AoP from sensor data or from reference to other information.
4. **Prepare the environment to execute the module.** Prepare sufficient computing power to execute the module in real-time.
5. **Connect to the system / commit the information.** Connect the module and begin to commit (add it to the blackboard) the new information generated by the module.
6. **Adjust the module.** Inspect the behaviour of the module when connected to the entire system and adjust if necessary.

The system Architecture

Information generated by any module must be open to all modules including new ones. To realize an open data-sharing framework, the blackboard model is adopted. The blackboard approach realizes equal access to data from any module in the system, hence the blackboard is used for storing and representing

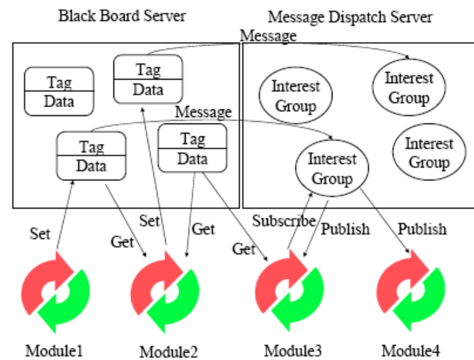

Figure 5.7: Combined blackboard structure with publish/subscribe model, taken from [24].

the data. Data on the blackboard is tagged to make it easy to access for the different modules. A publish/subscribe model is used for notifying the modules about events. Modules can view events by subscribing to the interested data (specified by their tags) in the the publish/subscribe framework.

The system uses a *data exhibition server* consisting of two servers, as depicted in figure 5.7. One of them provides a data exhibition service based on the blackboard model - *Blackboard Server*, and the other provides an event notification service based on the publish/subscribe model *Message Dispatch Server*. A module developer can gain an overview of -, select - and commit new information through the data exhibition service. A developer can use either the data exhibition or event notification service to access to the same piece of information. Thus, the developer can use the data as *state* or *event* information.

Data committed to the data exhibition server are defined as floating point numbers and text strings. In the case of text strings, the developer can define the data format following discussions with other developers. It is prohibited to commit data that requires contextual information to interpret the meanings. That means, modular outputs are limited to expressions of existing knowledge. Analysis of the causality of the data is performed in the modules.

Implementation of the System Architecture

The data exhibition server consists of two servers, a blackboard server and a message (dispatch) server, as illustrated in figure 5.7 and 5.8. The blackboard server responds to SET, GET and MONitor requests from the clients. Data is distinguished by the tags (text strings). Setting and getting requests are followed by tags to specify the area and data (for the setting request). The server responds to the request by notification of ACKnowledge (for setting request) or requested data (for getting request). A monitor request, which is prepared for reading the tag and data written to the blackboard, is followed by an index number and the server responds to the request by answering the tag and data written on the real address indicated by index number.

The message dispatch server responds to the SUBscribe and the POST re-

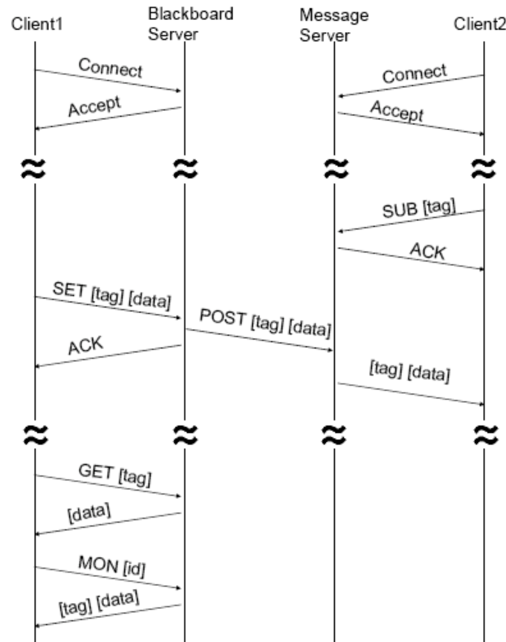

Figure 5.8: Transactions between the servers, from [24].

quest from the clients. The subscribing service provides a service based on regular expressions, and automatically adds client IDs to subscriber lists(*Interest Groups*) when a developer generates a new tag. The posting service will deliver the assigned message to the entire client list for the specified tag.

The blackboard server will send a post request when it receives a set request. Therefore, data can be accessed actively by sending a get request, or passively by sending a subscribe request.

5.2.4 MAPSEC: Mobile-Agent Based Publish/Subscribe Platform for Electronic Commerce

In [31, 30] Sahingoz and Erdogan presents MAPSEC, an architecture for e-commerce systems. They present a platform that uses the publish/subscribe mechanism for utilization of the system, and mobile agents as mediators between buyers and suppliers.

Introduction

Electronic commerce technology offers the opportunity to integrate and optimize the global production and distribution on supply chain. Software agents help to automate a variety of tasks including those involved in buying and selling products over the Internet. The electronic marketplace is dynamically changing, as any number of buyers and suppliers can be present at any time.

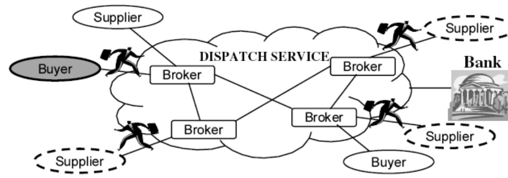

Figure 5.9: The architecture of MAPSEC, taken from [31].

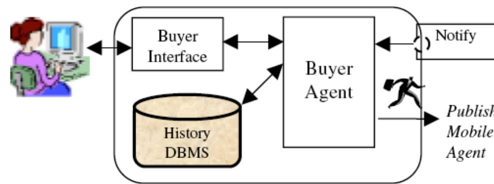

Figure 5.10: The MAPSEC buyer subsystem architecture, taken from [31].

The MAPSEC Architecture

The MAPSEC electronic commerce system involves three actors. *Buyers* are looking to purchase services. *Suppliers* or sellers offer the services and *Dispatch Service* facilitates communication between buyers and suppliers. The MAPSEC architecture, illustrated in figure 5.9, provides all the services which are essential to agent-based commercial services. This includes a communication infrastructure, a mechanism for storage, transfer of goods, banking and monetary transactions along with an economic mechanism for brokered buyer-supplier transactions.

To request a purchase order from the MAPSEC system, a buyer has to initialize a *buyer subsystem* on its machine. Figure 5.10 illustrates the buyer subsystem architecture. A human user interacts with the *Buyer Agent* via a *Buyer Interface* module. Progress of current transactions, and past transactions are stored in the *History DBMS*. Goods are categorized into generic names such as "car", "CD" and "book". When a Buyer Agent receives a purchase request from a user, it uses category information to prompt the user for search criteria to find the specific item to be purchased. Information required are name, maximum price, required quantity, and required delivery date. Then, it creates a *Mobile Agent* to search for product information and to perform goods or services acquisition in MAPSEC. The Buyer Agent specifies the criteria for the acquisition of the product, and dispatches the Mobile Agent to the potential suppliers. The mobile agent visits each supplier site, searches the product catalogs according to the buyer's criteria. Then, it returns to the buyer site with the best deal it finds and adds it to the History DBMS.

To subscribe to the system, a supplier has to initialize a *supplier subsystem* on its machine. Figure 5.11 illustrates the supplier subsystem. The *Supplier Interface* module facilitates the interaction between a user and *Supplier Agent*.

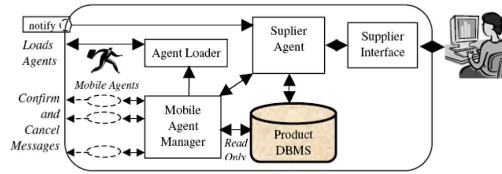

Figure 5.11: The MAPSEC supplier subsystem architecture, taken from [31].

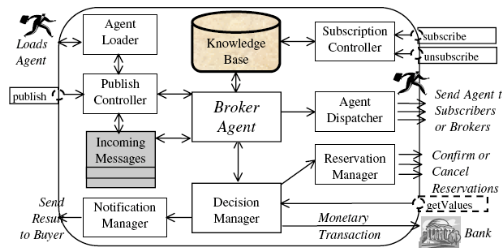

Figure 5.12: The inner structure of the MAPSEC Broker, taken from [31].

Supplier Agent processes purchase orders from buyer agents and decides how to execute transactions according to selling strategies specified by the user.

The *Products Database Management System* provides information about the products that may be sold by the Supplier Agent. Goods are added to the database either by manually adding a resource to the database or by purchasing goods from other agents. Each *Mobile Agent* on the supplier side searches the supplier database for the product it is interested in. It determines whether the required quantity is available to be offered or not. If so, the Supplier Agent gives an immediate quotation to the buyer's Mobile Agent.

Mediation between buyers and suppliers is done by the centralized *Dispatch Service*, consisting of several *Brokers* in a topology as shown in figure 5.9. The inner structure of a broker is given in figure 5.12. MAPSEC is basically an event-based system. A Broker implements the publish/subscribe paradigm in which purchase events are published and made available to the supplier components of the system, through notifications.

Both suppliers and buyers have to know the address (URL) of the *Broker Agent* that they will connect. When a buyer wants to buy a product, it creates a mobile agent with the necessary information and sends it by calling the *publish()* method of the broker. If a supplier wants to register (or unregister) to the broker, it calls the broker's *subscribe()* (or *unsubscribe()* method giving the necessary parameters. The parameters include products information, name, password, etc. *Knowledge Base* is a database, which keeps the information about the suppliers, buyers, brokers and products in the MAPSEC system.

Incoming agents are downloaded by the *Agent Loader*, and then they are dispatched to suppliers by the *Agent Dispatcher* when a request arrives from the Broker Agent. *Publish Controller* manages a queue of incoming messages,

which eventually are processed by the Broker Agent. A message is evaluated, by using the Knowledge Base, and a list of target suppliers and brokers are selected. Agents then send copies of the mobile agent to the selected brokers and suppliers, through the Agent Dispatcher. Suppliers and brokers return their results by calling the *getValue()* method of the *Decision Manager*. The Decision Manager compares the incoming results from the suppliers and selects the one that are closest to satisfy the constraints and sends a reply message to the brokers and suppliers through *Reservation Manager*. Decision Manager also sends its decision to the buyer, to be added to its History DBMS, by means of *Notification Manager*.

5.3 The Comparison to CoPS

In this section we compare the relevant work described in section 5.2 to the CoPS framework architecture that is proposed in chapter 3 and 4, using the criteria listed in section 5.1. These criteria describe challenges in collaborative software systems. Not all the relevant work is equally emphasized when discussing the different criteria. This is because we focus on the most important and interesting similarities and differences. In addition, some of the articles lack necessary information.

All of the systems described as relevant work, including the CoPS framework focus on collaborative software modules using a shared repository to solve different problems. The shared repository might include a shared memory or/and a shared workspace. The definitions of the software modules and the shared repositories are one of the differences between the systems.

In the *CoPS framework*, software modules are defined by the different *agents*: Task Responsible (TR), Problem Solver (PS), Decomposer (DEC), Matchmaker and Personal Assistant (PA). It is mainly the TRs and PSs that cooperate to solve a problem, the other agents assist in different stages of the problem solving process. The CoPS shared memory structure, used by the TRs and PSs in their cooperation, is defined by the *TEAM SPACE*. The TEAM SPACE itself is comprised of two components: the TS agent and the TS structure. *TS agent* controls the access to the *TS structure*, which conceptualizes the shared memory structure.

In the *fixture design system*, the software modules are realized by *knowledge sources* (KSs), and the these are divided into six functional groups representing their knowledge domain: reasoning mechanisms, re-allocation, deformation, stability, accessibility, and fixturing design principles. A *blackboard data structure* serves as the shared memory. A *control shell* controls the access to the blackboard data structure.

Software modules in the *blackboard based multiagent system* for project support, are like in CoPS , realized by *agents*. The agents are categorized into different intelligent agencies (IAs): Personnel Allocation Agency, Coordination Agency, Scheduling Agency, and Communication Agency. These agents collaborate to support a project manager and his team members in executing a project, by using a shared memory structure which is a *QFD embedded blackboard*.

The *system for collaborative development of an interactive robot*, can be seen as having composite modules. Here we suggest that a module is comprised of a *developer* (human) and the *software module* it develops. And both parts of

the module access the shared memory, which is a *blackboard*. The developers collaborate to develop an interactive robot by communicating information and results via the blackboard. And their software modules collaborate to simulate the interactive robot by committing their signal outputs to the blackboard and by taking as input the committed signals from other software modules. The software modules together represents the implementation of interactive robot developed so far.

The last system, *MAPSEC*, has two categories of software modules, realized by *agents* that are collaborating to trade goods between sellers and buyers: *Supplier* and *Buyer*. The shared memory or workspace could also be seen as a set of software modules. But here we choose to say that the *Dispatch Service*, consisting of a network of Brokers (agents), is a set of shared workspaces. Each *Broker* represents a single workspace.

5.3.1 Representation

The first challenge in collaborative software systems is *representation*, which deal with how one gets the software modules to understand each other, and thus how the software modules understand the information kept by the shared memory structure. Keywords considering representation in the systems to be compared are:

- Both direct and indirect exchange of information, only indirect exchange of information.
- The same knowledge representation in software modules and the shared memory, different knowledge representation in software modules and the shared memory.
- One software module to access the shared memory (translator), several software modules to access the shared memory.

The CoPS agents represent their knowledge using ontologies and they *exchange information directly* through messages, using the same message protocol, and the same communication language. The content of a message is a symbolic representation using the structure of the language and the vocabulary of an ontology. When an agent receives a message, it is only able to interpret the message content, if it knows about the ontology and the communication language that was used to encode it.

Indirect information exchange also occur between the TR and PSs in CoPS, that are part of the same problem solving team. The team communicates through the TEAM SPACE. The TS agent, part of the TEAM SPACE communicates with the PSs and TR part of a team, using messages (like described above). Imagine that PS1 has agreed to perform actionA and PS2 has agreed to perform actionB. PS2 needs the results from actionA before it can execute actionB. The TS agent extracts information from the TS structure, finds that actionA is ready to be executed, translates the information about the action into a message format that is understood by PS1, and sends it. PS1 executes the action and returns the result to the TS agent. Then, the TS agent gets the TS structure to add these results in the Result Library. Next time the TS agent

extracts information from the TS structure it finds that actionB is ready to be executed because the results from actionA is placed in the Result Library. Then, the TS agent communicates the results from actionA to PS2, and PS2 is able to execute its action. The results from actionA was communicated indirectly from PS1 to PS2 via the TEAM SPACE. The TS agent is the only software module accessing the shared memory structure - TS structure.

The TS structure does not use the same knowledge representation as the agents. Knowledge or facts stored in the TS structure are represented using pre-defined templates. These templates are only known by the TS structure and its components. When the TS agent extracts information from the TS structure, the TS structure translates the facts in the Result Library and Goal Stack into lists of actions or PSMs before the information is returned to the TS agent. Results from action executions and PSM compositions are also communicated from the TS agent to the TS structure in form of action- and PSM lists.

The blackboard based multiagent system meet the representation challenge in a similar way to CoPS. Agents communicates information directly using a common communication protocol (KQML), and they communicates information indirectly via the QFD embedded blackboard on behalf of the system users. A difference to CoPS is that this system allows several agents to access the shared memory structure.

In the fixture design architecture, the knowledge sources only exchange information indirectly via the blackboard data structure. In the implemented system, the GBB Administrator is the only module that can access the blackboard data structure, and it communicates information both ways between the knowledge sources and the blackboard data structure. The blackboard data structure is composed of directories (spaces) and objects (units). Objects' properties are resident in attributes (slots). During the problem solving, objects and their properties serve as input to a set of KSs which produce new information to be added to the blackboard data structure, also on the form of objects and properties. The objects and their properties define the vocabulary of the solution space, and must be known by KSs.

In the system for collaborative development of an interactive robot, all of the modules may access the blackboard. Information put on the blackboard must be tagged and represented on the form of a string, or an integer. The string-values should not require contextual information to interpret the meanings; their meanings are decided in beforehand by the developers. Developers communicate indirectly via the blackboard, and they understand what others have committed because of the earlier agreement on the meanings. Software modules created by the developers also communicate indirectly via the blackboard, by exchanging their sensor outputs. The meanings of the different tagged outputs were known by the developers when they created the software module, and thus the developer made the software module able to interpret them.

The tags used in this system have approximately the same function as the templates used in our TS structure. The meanings of the templates are also defined in beforehand, and they make it easier to extract specific knowledge.

5.3.2 Awareness

The second challenge in collaborative software systems is *awareness*. This challenge involves making modules aware when something relevant to them (an event) occurs. Keywords considering awareness in the systems to be compared are:

- Software modules are actively seeking information about relevant events or they are passively receiving information about relevant events.
- Who are notified about a particular event.
- Receiver and notifier of events.

In the CoPS framework agents make each other aware when something relevant to them happens by sending a message. When the TR receives a task solving request from a PA, the PA makes the TR aware that it should try to solve the task. When the TR sends a request for a task decomposition to the DEC, it makes the DEC aware that it should "do some work". Thus, agents in CoPS may be both receivers and notifiers about a particular event.

The TS agent makes the PSs and TR related to a TS structure component, aware when information relevant to their actions and PSMs are present in the Result Library knowledge base. When an action is ready to be executed or a PSM is ready to be composed this can be seen as an event. A particular event is only communicated to one of the PSs or the TR. The TS agent actively "seeks" for information about events in the TS structure. The other CoPS agents only passively receives information about events.

In the fixture design system, the KSs are all passively receiving information about relevant events, and they may not notify the other KSs about events. The KSs are made aware of relevant events on the blackboard data structure by the GBB Administrator. Events here are defined as change of data on the blackboard data structure. When an event occurs, all of the KSs are notified, and have the opportunity to evaluate the information related to that event, to see if it is able to act upon it.

In the blackboard based multiagent system, awareness is achieved in about the same way as in CoPS. Agents notify each other about happenings relevant to them, and changes in data on the blackboard are also handled by the relevant agents.

In the systems for collaborative development of an interactive robot, a developer can select either the data exhibition or event notification service to access the same information. By using the data exhibition service the developer actively get the information relevant to them, and by using the event notification service the developer is notified about changes to the data it is interested in by subscribing to certain types of information. When an event occurs on the blackboard (change of data), the label of this piece of data is used to find the developers that have subscribed for that information. All of the subscribers belonging to the specific label is notified about the event.

A human user of the MAPSEC system which wants to buy a product interacts with a Buyer Agent. When Buyer Agent receives a purchase request from the buyer, it passively receives information about a relevant event. Criteria for the product are specified. Then a Mobile Agent is created and sent by calling the *publish()* method of the Broker Agent. The purchase event is published and made available to the Supplier Agents of the system.

The supplier agent subscribes to certain purchase events by calling the *subscribe()* method of the Broker Agent giving the necessary parameters. The parameters describe the criteria for the product it offers. When a purchase event is published that matches the subscription criteria of a Broker Agent, the Broker Agent is made aware of this. Broker Agent call the *notify()* method on the Buyer Agent. The subscription information kept by the Broker Agent decides which Supplier Agents are notified about a purchase event.

This may be compared to the function of the TMST in CoPS. PSs offer to perform actions, and joins a team. To execute the action, the PS needs the input information of this action. Thus, when a PS offers to perform an action it also subscribes to the input information needed to execute it. During the problem solving in the TEAM SPACE, the TS agent notifies the PSs and TR about information that has arrived, and that they need to execute actions and compose PSMs. Information is published in the TS structure, via the TS agent by PSs or TR. Like in MAPSEC, "subscription information" stored in the TMST and kept by the TS structure is used by the TS Agent to decide which PSs or TR are notified about an event (actions ready to execute, PSMs ready to compose) in the TS structure. The difference is that published information is "modified" and does not directly serve as an event. One or several events are produced from the the published information (by executing the rules kept by the TS structure), and these events result in event notifications.

5.3.3 Investigation

The third challenge in collaborative software systems is *investigation*. Software modules need to find information related to their current activities (problem solving). How this information is found and by whom are important keywords here. The investigation challenge in the systems we are comparing seems to be closely related to the awareness challenge. This is probably because the events described before are related to change of information. How and by whom a software module is made aware of an event, and how and by whom a software module is provided with information related to their current activities, are almost the same.

CoPS agents get information related to their current activities enclosed in messages from other CoPS agents, or from the TS agent (extracting information from the TS structure). Information does not have to be searched for in the TS structure, since it generates the events; actions ready to execute and PSMs ready to compose, together with the information needed to respond to this event. It is originally the TSMT (relations between nodes, node input and outputs, I/O dependencies), that has the knowledge about what information is needed by whom during the problem solving process.

In the fixture design system, the GBB administrator extracts information from

the blackboard, which it gives to the different knowledge bases. This happens when an event occurs because of change in the blackboard data structure. The knowledge bases use this information to decide which KSs are to be activated. The activated KSs get the information they need from the knowledge base preprocessor.

In the blackboard based multiagent system the agents get information related to their current activities enclosed in messages from other agents, and from finding the information on the blackboard. We might say that the QFD - method is like the TMST when it is not instantiated for a problem (modified with a solution, after the team formation process). Both structures define how information should flow between the different phases a problem solving process. In CoPS the TMST is "specialized" for solving a specific problem with a given team of problem solver, when the team formation is done. And it then defines how the information should flow between team members during the execution of the problem solving. In the blackboard based multiagent system a Work Breakdown Structure (WBS) represents the hierarchy and precedence of project tasks. This WBS does not have several "solutions" like the TMST, but team members are recruited on the basis of the WBS. Since the WBS represents the precedence of tasks it also gives information about information dependencies between the agents performing the different tasks.

In the system for collaborative development of an interactive robot, software modules may find information themselves related to their current activities through the data exhibition service. Data is distinguished by tags, and are therefor easily found. The software modules may also be informed about information they are interesting in by subscribing for data with specific tags. Then it is the publish/subscribe service that "finds" and delivers the information.

5.3.4 Interaction

The fourth challenge in collaborative software systems is *interaction*. Interaction between the different software modules is the most important issue considering the collaborative work. Software modules should be able to use the concurrent work of others while working on a shared task. Keywords considering interaction in the systems to be compared are:

- Direct and indirect interaction.
- Serial (shared memory with single access) and concurrent interaction (shared memory with multiple access).

The work on the shared task of CoPS agents are handled in two stages. First, the agents interact directly until the task is decomposed into a TMST, and a team of PSs are formed. Second, the TR and PSs part of the problem solving team interact indirectly through the TEAM SPACE. The interactions through the TEAM SPACE follow a publish/subscribe kind of pattern (like described under the discussion of awareness).

The indirect interaction in CoPS may be both serial and concurrent, dependent on the sub-task relationships defined in the TMST. If several actions

and/or PSMs are ready to be processed, the TS agent notifies all of the relevant agents simultaneously. The agents process these actions and PSMs in parallel and return the results when they are done. All of the results are updated by the TS agent in the TS structure at the same time.

Sometimes several teams of problem solving agents also work in parallel. Then, there may occur interactions between the different teams as well. Imagine that the teams communicate with the same TS agent which keeps a separate TS structure for each of them. During the problem solving, the TS agent recognizes that two tasks existing in different TS structures are the same. Only one of these tasks need to be performed, as the result from this one is copied by the TS agent to the TS structure keeping the other task. This saves time effort and cost. The solving of several problems in parallel was not discussed by the relevant work presented in this chapter.

The blackboard based multiagent system also has both direct and indirect interaction between the agents. Agents work on the shared task of supporting a project manager an its team in performing an engineering project.

The fixture design system uses a more traditional blackboard approach and the KSs only interacts indirectly through the blackboard data structure. Here we may also have some kind of concurrent interaction since several KSs may be activated at the same time. They do not access the blackboard data structure themselves, but communicates their results to the GBB Administrator. The KSs work on the shared task of performing a fixture design.

In the system for collaborative development of an interactive robot the developers interact both directly and indirectly using the blackboard. The software modules only interact indirectly via the blackboard. The developers work on the shared task of developing an interactive robot. And the software modules works on the shared task of demonstrating the robot's composite behaviour.

Buyer Agents and Supplier Agents in MAPSEC only interact indirectly via a Broker Agent. Their shared task is to exchange goods between buyers and suppliers.

5.3.5 Integration

The fifth challenge in collaborative software systems is *integration*. When talking about integration here, we mean how results produced by the different modules are combined into an integrated result. Keywords considering integration in the systems to be compared are:

- Relationship management.
- Results integrated by software modules or by the shared memory structure.
- Opportunistic and planned problem solving. The use of a result integration model.
- Definition for when the solution to the initial problem is found.

In CoPS, the relationships among tasks are defined by the TMST and managed by the TS structure. The solution part of the TMST represents a plan for the problem solving, and can also be seen as a result integration model. The TS structure transforms the TMST solution into an executing plan represented by a set of rules. Then the TS agent can at any time extract information from the TS structure considering which actions or PSMs to perform next and what information is needed. The TS structure also makes sure to propagate output values upwards in the TMST. When the sub-node of a task is an action and the action is executed, the output of this action is set as the output of the task. When all of the subtasks of a PSM has an output-value. The PSM may be composed. This means to integrate the results/outputs from the PSMs subtasks. The composition of a PSM is always performed by the TR that administers the problem solving process. That PSM is the sub-node of a task, and the output from the composition of the PSM is set as the output of that task. When the initial task has an output - value, the problem solving is done. And the output of the initial task is the final solution.

The propagation of values, is not actually done on the TMST, but it is represented by adding new facts to the TS structure Result Library and Goal Stack.

In the blackboard based multiagent system, the WBS represents the hierarchy and precedence of project tasks, like the TMST in CoPS. While the project is in progress, the design team carries out the assigned tasks according to the preset schedule or plan. During the project, the QFD-embedded blackboard evolves with information, as each team member is responsible for contributing useful information to the team. The integration of results are done by the team members, using the system. For example, when a report is finished by one team member, this is forwarded to another team member (by the Coordination agency) for the follow-up process. Who is responsible for the follow up process, is defined by the WBS. The WBS can be seen as a result integration model. When all of the tasks are completed, the project (initial task) is done.

In the fixture design system, KSs solve a problem opportunistically. The problem solving process starts with an initial blackboard data and continues until the final blackboard data is reached when all the involved KSs unanimously agree (or disagree) about a design decision. The propagation of the decision making process as it progresses through the intermediate stages of the blackboard, follows a tree-like searching path. Based on the current blackboard data, the system generates the searching tree in a dynamic and opportunistic way. In each step of the searching process any KSs can be activated to contribute with new information. The fixture design system does not have a model for integrating results, as this model is very difficult to make because of the nature of opportunistic problem solving.

In the blackboard system for collaborative development of an interactive robot, the information on the blackboard is not integrated in any ways. Information is just added, in such a way that it will be available for other modules. When a new module is developed, it is integrated into the system, but it does not lead to changes in any of the other modules. But it might change the outcome of the whole system. This is module integration, and not integration of the data on the blackboard. Meaning that data is integrated in the modules (in how these

work together) and not on the blackboard. The blackboard only keeps the result from this integration. This system also uses opportunistic problem solving and does not have an result integration model. The "solution" is found when the different modules part of the robot, show the behaviour that the developers wanted.

5.3.6 Coordination

The sixth challenge in collaborative software systems is *coordination*. Introducing coordination mechanisms in a system of collaborating software modules is important when we want to anticipate or control the final outcome of the system. By coordinating the software modules, we get them to focus their activities on the right thing at the right time. In the systems we have been looking at there are two main differences; planned coordination, and opportunistic coordination (coordination decided by status of data kept by the shared memory).

In CoPS the agents decide for themselves when to engage in some activity. But when a TR has agreed to solve a problem, its focus is to come up with a solution that can be returned to the requester of the problem. The TR coordinates the other agents (DEC, PSs) during the team formation process. When a team is formed, the TR and PSs part of that team are coordinated by the TS agent. The TS agent follows the plan outlined by the TMST, which is automated by the TS structure.

In the blackboard based multiagent system, coordination done before a project team is formed is done by the project manager, Personnel Agency and Scheduling Agency. When a team is formed, the agents and project team members, are coordinated by the Coordination Agency, using the QFD embedded blackboard and the WBS.

In the fixture design system, all domain KBs are managed and coordinated by the GBB administrator. The KBs coordinates its KSs, by deciding which KSs to activate according to the incoming information from the GBB administrator. There is no plan (like the TMST or the WBS) used for the coordination in the fixture design system. Agents are only coordinated according to the status of the data kept by the blackboard data structure in combination with some KS-selection criteria used by the KBs.

The developers and software modules in the blackboard system for collaborative development of a robot, are only coordinated by the blackboard publish/subscribe service. Coordination here does not use a pre-defined plan either, it is solely decided by the status of the data kept by the shared memory.

In MAPSEC agents are coordinated by the Broker Agent.

5.4 Summary

In this chapter, our proposed CoPS framework with a shared memory structure, the TEAM SPACE, has been compared to relevant work. The relevant work

included here, involved other systems consisting of collaborative software modules and shared memory structures. These systems were compared to CoPS on the basis of six different criteria. Through this, it became more evident that CoPS adopts methodologies from the different paradigms: multiagent architectures, blackboard architectures and publish/subscribe models. During the comparison, our CoPS framework architecture was also described from other perspectives than were used in chapter 4.

Chapter 6

Implementation Details

In previous work with the CoPS framework a CoPS framework prototype was implemented. This work is described in [34, 33]. In this thesis we have focused on extending the CoPS framework by introducing a shared memory structure that is used to guide a problem solving team in solving their subproblems and sharing their results. To integrate this shared memory structure with the existing CoPS framework, the pre-existing parts of the CoPS framework also had to be modified. Chapter 4, described the TEAM SPACE architecture proposed for the shared memory structure. In addition, it explained how the Task Responsible (TR) and Problem Solver (PS) should be modified, to make it able for the existing CoPS framework to use the TEAM SPACE architecture. This chapter presents how the modifications and extensions to the CoPS framework is transfered and implemented in the CoPS framework prototype.

Section 6.1, has an overview of which parts of the CoPS architecture that are extended and which parts that are modified in the CoPS framework prototype. An introduction to the applied implementation tools, and to the pre-existing CoPS framework prototype is given in section 6.2. In section 6.3 we describe the implementation of the corrections to the CoPS framework, suggested in chapter 3. In section 6.4 we the describe the implementation of the extensions to the CoPS framework, also suggested in chapter 3. And finally in section 6.5 implementation details considering the TEAM SPACE architecture are presented.

This chapter is provided to give a detailed description of the CoPS framework prototype, and to show how the TEAM SPACE architecture constructs and the other extensions to the CoPS framework is translated into an implementation. The implemented CoPS framework prototype, described here, are used for a medical domain application in chapter 7, to show that the prototype works in the pre-described way.

6.1 Implementation Overview

What we have implemented is illustrated by figure 6.1. The pre-existing prototype of the CoPS framework focus on the problem solving process steps: *problem analyzis* and *team formation*. To realize these steps The Task Responsible (TR), Decomposer (DEC) and Problem Solver (PS) were implemented. The conver-

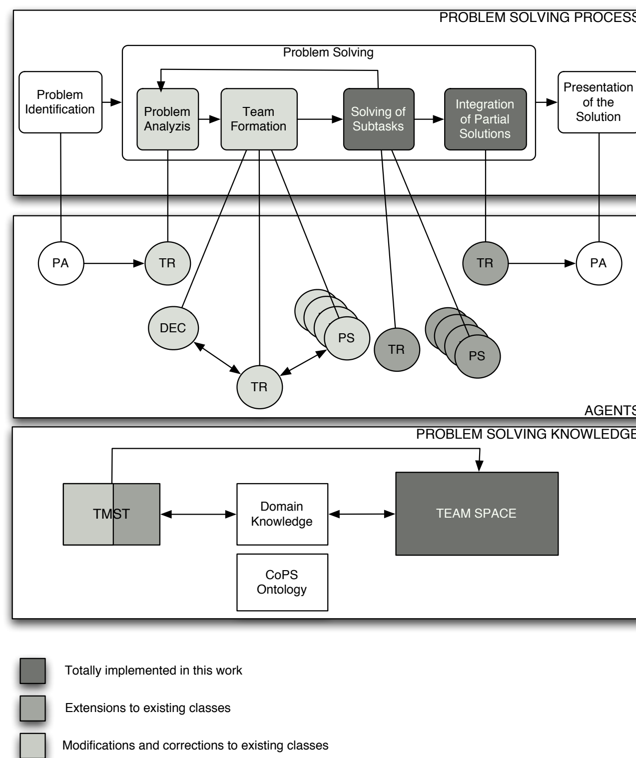

Figure 6.1: The three layered CoPS architecture, showing which parts are involved in our implementation. The agent abbreviations are: PA - Personal Assistant, TR - Task Responsible, DEC - Decomposer, PS - Problem Solver.

sations between these agents were implemented in addition to the TMST. The pre-existing prototype did not function optimally, so we have made some modifications and corrections to the TR, DEC, PS and TMST considering *problem analysis* and *team formation*. We have also implemented the *solving of subtasks* - and *integration of partial solutions* process steps of the problem solving. This required us to extend the pre-existing implementation of the TR, PS and TMST, and to implement the TEAM SPACE.

As one can see from figure 6.1, the implemented prototype of the CoPS framework is still not complete. Implementation of the remaining architectural components, most likely will infer further modification to the existing implementation.

6.2 Implementation Tools and The Pre-Existing CoPS Prototype

The pre-existing CoPS prototype is implemented in Java. *JADE* (Java Agent DEvelopment) is a Java based software framework which simplifies the implementation of multiagent systems through a middle-ware that complies with the FIPA specifications [2]. JADE was used to implement the CoPS agents and their conversations. *jCreek*, is the Java implementation of CreekL (implemented in Lisp). CreekL is a flexible frame-based knowledge representation language, which is modeled using semantic nets [15]. jCreek was used to implement the TMST. The implementation in this master thesis in addition uses *Jess*, which is a rule engine and scripting environment written in JAVA [32].

The three different Java-implementation tools are shortly introduced next. We also describe of how JADE and jCreek were used in the pre-existing CoPS prototype to realize the different parts of the CoPS architecture.

6.2.1 JADE and CoPS Agents

All necessary information needed to understand and use JADE can be found on the JADE-website [2]. Particularly useful documents are *JADE Programmer's Guide* [1] and *JADE Tutorial - Application-Defined Content Languages and Ontologies* [5].

The implemented part of the CoPS framework has four abstract agent classes, which are implemented using JADE. These classes are *CoPSDecomposer*, *CoPSTaskResponsible* and *CoPSProblemSolver* which are subclasses of *CoPSAgent*. As these agents are implemented using the JADE framework, they extend the JADE *Agent* class. A Personal Assistant (PA) does not necessarily need to be a CoPS agent, it may though. This class is not yet implemented. FIPA has standardized the use of a Directory Facillitator agent in multiagent system platforms, this agent will play the role of the Matchmaker. The entire class hierarchy is illustrated in figure 6.2, and the implemented classes are shaded in grey.

CoPS - agent classes implementing the JADE *Agent*, may use services provided by the JADE framework, like sending and receiving messages. To control how the agents react and behave, the JADE framework uses different classes of behaviour.

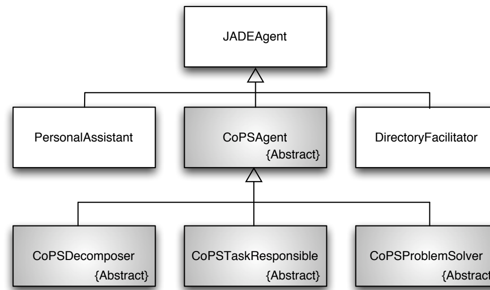

Figure 6.2: The class hierarchy of the CoPS agent implementation [34]. The shaded classes represent which agents that are actually implemented in the CoPS framework prototype.

JADE behaviours

There are several kinds of behaviours, and when using JADE an agent's behaviour is usually defined by implementing one or more of the behaviours provided by the framework. Both primitive and complex behaviours exist. Complex behaviours are comprised of a set of behaviours, while primitive behaviours are not.

During the startup of agents extending the JADE *Agent* class the *setup()* method is called. In the *setup()* method one or more behaviours are added to the implemented agents' set of current behaviours. The set of behaviours is arranged in a queue and behaviours are selected and executed in a non-preemptive round robin schedule. Behaviours are executed one at a time until the *action()* method of each behaviour is finished executing. Behaviours can also be added to the agent during execution of other behaviours.

Conversations between the agents are implemented using some of the complex JADE behaviours. JADE provides implementation of some of the interaction protocols specified by FIPA. The request-like protocol are implemented using *AchieveREInitiator* and *AchieveREResponder*. The Contract Net protocol, on the other hand, is implemented using *ContractNetInitiator* and *ContractnetResponder*. As defined by FIPA, initiators are the agents taking the initiative to start a conversation, while responders are agents responding to a request from the initiator. An extensive introduction to JADE behaviours and how these works is given in [1]. Next, we will give a short introductions to some behaviours that are used in the implementation of the CoPS prototype.

The OneShotBehaviour class

This abstract class models simple atomic behaviours that must be executed only once. A class that extends the *SimpleBehaviour* class implements the *action()* - method. After the behaviour has been added to the agent and chosen to be executed, the code in the *action* - method is run, and that is it [1].

The ParallelBehaviour class

This class implements an abstract class *CompositeBehaviour*. The *CompositeBehaviour* class are made up by composing a number of other behaviours(children). So the actual operations performed by executing this behaviour are not defined in the behaviour itself, but inside its children. The *ParallelBehaviour* class executes its sub-behaviours (children) concurrently and terminates when a particular condition on its sub-behaviours is met [1].

The FSMBehaviour class

This class, like the *ParallelBehaviour class*, implements the abstract class *CompositeBehaviour*. The *FSMBehaviour* class executes its children according to a Finite State Machine (FSM). Each child represents the activity to be performed within a state of the FSM and the user can define the transitions between the states of the FSM. One of the children can be registered as the starting state, meaning that the *FSMBehaviour* begins executing that associated behaviour. When a child corresponding to a state completes, its termination value (as returned by the *onEnd()* method) is used to select the transition to fire and a new state is reached. At next round, the child corresponding to the new state, will be executed. Some of the children of an *FSMBehaviour* can be registered as final states. The *FSMBehaviour* terminates after the completion of these children [1].

The AchieveREInitiator and AchieveReResponder classes

The *AchieveReInitiator* class is provided in JADE for aiding the implementation of the initiator of a request-like conversation. It defines a set of states (sub-behaviours) and the transitions between them. It is up to the programmer to define what happens in those states. Each of the states are, as in the *FSMBehaviour*, implemented using behaviours. In the JADE API the programmer is given access to some of these states through a set of methods. Programmers are also given the possibility to implement one or more of the states completely, by registering the behaviours that replace the default content (behaviour) of the given states.

Among other methods, the *AchieveReInitiator* class has methods for preparing and sending request messages (*prepareRequests()*), handling responses (*handleAgree(), handleRefuse(), handleNotUnderstood(), handleAllResponses()*), and handling result notifications (*handleInform(), handleFailure(), handleAllResultNotifications()*). These methods must be implemented by the class implementing the *AchieveREInitiator* class, or else nothing happens when they are called.

The *AchieveReResponder* class is implemented by agents that respond to requests in a conversation. As with *AchieveReInitiator*, there are defined a set of states (sub-behaviours), which may be reached trough methods or changed by states defined by the programmer.

Among other methods, the *AchieveReResponer* class has methods for responding to a request (*prepareResponse()* and sending the results (*prepareResultNotification()*). The response to a request can be an agree message, a refuse message, or a not-understood message. If the response is an agree message, the results are prepared. The message notifying the result can be an inform message

or a failure message. These methods must be implemented by the class implementing the *AchieveREResponder* class, or else nothing happens when they are called [1].

The ContractNetInitiator and ContractNetResponder classes

The *ContractNetInitiator* and *ContractNetResponder* classes are provided in JADE for aiding the implementation of the initiator and responder of a conversation following the Contract Net protocol. Like the *AchieveReInitiator* and *AchiveReResponder* classes, these classes define a set of states (sub-behaviours) and the transitions between them.

Among other methods, the *ContractNetInitiator* class has methods for preparing and sending the call-for-proposal message, requesting an action(*prepareCfps()*) and for handling the responses (*handlePropose(), handleRefuse(), handleNotUnderstood(), handleAllResponses()*). Handling the responses include making and sending proposal messages for the accepted proposals (bids) and reject-proposal messages for the rejected proposals (bids). The *ContractNetInitiator* class also has methods for handling the results from agents which proposals were accepted (*handleInform(), handleFailure(), handleAllResultNotifications()*). These methods must be implemented by the class implementing the *ContractNetInitiator* class, or else nothing happens when they are called.

Among other methods, the *ContractNetResponder* class has methods for preparing and sending a response to the call-for-proposal message (*prepareResponse()*), and a method for preparing the result of the requested action (*prepareResultNotification()*). The response to a call-for-proposal message can be a propose message, a refuse message, or a not-understood message. If the agent has made a proposal it waits for an accept-proposal message and when this one is arrived the actions is performed and a the result i returned as an inform message or as a failure message. These methods must be implemented by the class implementing the *ContractNetInitiator* class, or else nothing happens when they are called [1].

6.2.2 jCreek and TMST

The TMST is implemented using jcreek, which is the Java implementation of CreekL. CreekL is a flexible frame-based knowledge representation language, which is modeled using semantic nets. This knowledge representation language has two main categories of concepts: entities (objects) and relations between them. Thus, for representing a task it is modeled as an entity, and all of the slots (variables) of the this task are relations, which have other entities as the slot values. Reasoning on the structure happens in form of traversing the nodes.

Main Classes

The implementation of the TMST is done by defining new entity types. *Action*, *Task* and *ProblemSolvingMethod* classes are represented as frames and implemented specifying a type of their own. An actions in a TMST are implemented by instantiating an *Action* entity type class. The new instance of *Action* is a new entity type, which is related to *Action*. This relationship is defined by an instance-of relation. No relation type classes have been implemented though,

since new relations are defined at real-time. Two types of relationships are used: instance-of relation, defining sub- and superclass relations, and structural relation defining other relations between entities.

When the TR reasons about the PSs and their capabilities, the TR represents them in its knowledge model through the entity type *Executor*. These four entity types are the main classes used for implementing the *TMST*: *Action*, *Task*, *ProblemSolvingMethod* and *Executor*.

Slots for the different nodes in a TMST, were listed and described in chapter 3 - figure 3.3. A *ProblemSolvingMethod* entity class has slots defined by relations to one or several *Task* entity classes. And a *Task* entity class has relations to one or several *Action* entity classes or *ProblemSolvingMethod* entity classes. The other classes in the *TMST* represent the rest of the slots in these main classes, and are called supporting classes.

Supporting Classes

The supporting classes are all the classes which are not represented as nodes in a TMST such as input output, assumptions, and so on. These are used for representing additional information needed to implement the *TMST* - the information that is used for describing the tasks, PSMs, actions and executors. The different supporting classes are: *Assumptions*, *Input*, *Output*, *Value*, *Goal*, *Cost* and *ControlInformation*. Entity types created from these classes are related to other entity types with the structural relation.

The overall class TMST

In the implementation of the *TMST*, the knowledge model is a class called TMST which is a subclass of the jcreek *LocalKnowledgeModel*. Thus, it is a structure for storing entities, represented by the different main classes and supporting classes, and relations. Methods for receiving all entities of type *ProbleSolvingMethod*, *Action* and *Task* are provided. In addition there are methods for executing different operations on the TMST.

- Setting Initial Problem: A method for setting the root node of the *TMST* instance.
- I/O Dependency Analyzation: A method for analyzing the I/O dependencies among all subtasks of all the PSMs in the model.
- Solution Space Generation: A method for generating the solution space of a problem. All nodes part of the solution space is activated.
- Solution Generation: A method for finding the best solution of the solution space. All nodes part of the solution are tagged, these nodes are necessarily also activated.

6.2.3 Jess

All information needed to understand and use Jess can be found on the Jess-website [32]. Particularly useful information has been found in *Jess 6.1 Manual* [10], *The Zen of Jess 2* [25] and *Some Guidelines for Deciding Whether to Use a Rules Engine* [29].

Jess is an expert system shell and scripting language written in Java. It supports the development of rule-based expert systems which can be tightly coupled to code written in the Java language. Java functions can be called from the Jess scripting language and Jess can be embedded in a Java application. We have used that last opportunity - A Jess rule engine is embedded in our implementation of the TEAM SPACE (TS) structure. The rule engine is used in the Rule Base Container, the Goal Stack and the Result Library. We do not use pure Jess language scripts, but java code which manipulates Jess through its Java API. Jess supports a long list of functions and options (see [10]), but here we only describe the different Jess-functions and options used in our implementation.

Facts

A rule-base system maintains a collection of knowledge nuggets called facts. So does our TEAM SPACE structure, where facts are stored in the Result Library and the Goal Stack. Facts can be added to the knowledge base using the *assert* - function. The sentence *(assert (executed (action ”gettingPatientInfo”)(output ”(Patient :name Paul :info Information about Paul)”)))*, adds the fact *(executed (action ”gettingPatientInfo”)(output ”(Patient :name Paul :info Information about Paul)”))* to the knowledge base. You can remove an individual fact from the knowledge base using the *retract* - function. The sentence *(retract (executed (action ”gettingPatientInfo”)))*, removes the fact asserted above from the knowledge base.

The ”structure” of the fact that we added and removed from the knowledge base is decided by a template. A template is defined by using the *deftemplate*-function. The sentence *(deftemplate executed (slot action) (slot output))*, defines the template used by our fact from before. *executed* is the name of the concept or the object that the fact describes. Following this name are the slots or attributes of the concept. Our concept *executed* has the slots *action* and *output*. The types of the slot-values may also be defined, but it is not necessary. In our fact the slot-value of the *action*-slot is the string: ”gettingPatientInfo”. The slot-value of the *output*-slot is the string: ”(Patient :name Paul :info Information about Paul)”.

Rules

Rules can take actions based on the contents of one or more facts. A Jess rule is something like an ”if ... then” - statement in a procedural language, but it is not used in a procedural way. While ”if ... then” - statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their ”if” -parts (their *left-hand-sides*) are satisfied, given only that the rule engine is running. Rules are defined in Jess using the *defrule*-function. One of the rules in our system may look like this:

```
(defrule rule-34
        (executed (action ”gettingPatientInfo”) (output ?out))
        (test (neq ?out nil))
        ?fact <- (ready_to_execute (action ”gettingPatientInfo”))
=>
```

(assert (actionoutput (name "gettingPatientInfo_output") (value ?out)))
(retract ?fact))

The name of the rule is *rule-34*. The rule fires if the facts; *(executed (action "gettingPatientInfo")(output "(Patient :name Paul :info Information about Paul)"))* and *(ready_to_execute (action "gettingPatientInfo"))* exists in the knowledge base, and if the Jess-function *test* evaluates to true. *?out* is a variable, and *"(Patient :name Paul :info Information about Paul)"* is bounded to that variable when one checks if the rule should be fired. Clearly, *?out* is not equal to (*neq*) nil, and *test* evaluates to true. The fact : *(ready_to_execute (action "gettingPatientInfo"))* is bounded to the variable *?fact*.

Rules are run by the function *run*. When *rule-34* fires, the fact *(actionoutput (name "gettingPatientInfo_output") (value "(Patient :name Paul :info Information about Paul)"))* is added to the knowledge base, and the fact *(ready_to_execute (action "gettingPatientInfo"))* is removed from the knowledge base.

To use the introduced functions, manipulating facts and rules, from Java, a *jess.Rete*-class must be instantiated. This class is the rule engine itself, and it has its own knowledge base, agenda, rules, etc. If this class is manipulated by calling appropriate methods, one has a working rule-engine. For example, our fact, from above, may be added by calling the method *executeCommand((assert (executed (action "gettingPatientInfo")(output "(Patient :name Paul :info Information about Paul)"))))* on the *Rete*-instance. Other classes from the Jess-API are used to extract information from the knowledge base.

6.3 Corrections

All of the corrections to the CoPS framework, listed in chapter 3, have been implemented - except the one that involves a full realization of dynamic formation of problem solving teams. The correction that has not been done was too extensive to cover this time, and will be proposed as future work in chapter 8.

In addition to the proposed corrections, other necessary modification were applied due to problems that were encountered during the implementation. What has been done is only mentioned here. More detailed information is added to the java-doc in the proper classes. Pre-existing parts of the architecture that are affected by the corrections are marked by the lightest shade of grey in figure 6.1. The corrections are:

- Extensive debugging was done so that the CoPS agents behave like they should, according to their descriptions in [34]. The messages sent between the agents are now corrected, in such a way that the TR at any time can conclude correctly about the state of the problem solving process. And the TR runs until it's being properly terminated.
- The agent classes *CoPSAgent*, *CoPSDecomposer*, *CoPSTaskResponsible* and *CoPSProblemSolver* contained inner classes representing the behaviours. These inner classes are made as separate classes to make the code easier to follow.
- All of the conversation-protocols (implemented as agent behaviours) are completed, and every significant message is properly handled. Before only

parts of the protocols were implemented. This made the conversations incomplete, and it was difficult to follow the governing idea of the problem solving process.

- The recursive solution generation in the *TMST* class, did not reach the correct solution. One recursively propagated the costs upwards in the TMST by always choosing the cheapest alternative, and tagged the nodes that became part of the solution in the same recursion. This is logically impossible to do. We moved tagging of the solution to its own method that was run on the *TMST*, after the final (and cheapest) cost for each node was decided.

- The generation of the solution space (activation of nodes considering agent proposals), in the TMST class, did not give the correct outcome either, so it has been corrected as well.

- The *CoPSProblemSolver* class has been drastically changed; now a PS can have several capabilities, and attend to multiple conversations at a time; meaning a PS can join several problem solver teams simultaneously. The changes to the *CoPSProblemSolver* class are further described in the next section, 6.4.

All these corrections are proved to be working. The proofs are presented in chapter 7, giving results from the test-run of an example application.

6.4 CoPS Problem Solving Process Extensions

The suggested extensions to CoPS, like the corrections, were introduced in chapter 3. To realize the problem solving process steps, *solving of subtasks* and *integration of partial results*, some of the pre-existing classes of CoPS had to be extended. Agent classes that were modified are: *CoPSTaskResponsible* which implements the TR and *CoPSProblemSolver* which implements the PS. The problem solving knowledge - class that was modified is *TMST*. Pre-existing parts of the architecture that are affected by the extensions are marked by the medium shade of grey in figure 6.1.

The *problem solving knowledge* layer of our three-layered architecture that was introduced in chapter 3, and illustrated in figure 3.2 and 6.1, also has the components CoPS Ontology and TEAM SPACE, in addition to the TMST. The CoPS Ontology is not implemented as part of our work, and will be proposed for future work in chapter 8. The TEAM SPACE is implemented as the main extension to the CoPS framework, and implementation details are described in section 6.5.

In this section, we describe how the modified version of the pre-existing classes, function. Most of the classes have gotten a new instance variable of the type *FileWriter*. An instance of *FileWriter* is associated with a TR and follows the TR and all of the agents that it involves in its problem solving process. The FileWriter is used by the different classes involved in one common problem, to log the evolutionary steps of the problem solving process to a file. The file gets the name of the TR and a ".txt"-suffix. When we have several TRs working on different problems one file is created per TR. The *FileWriter* class is shown in

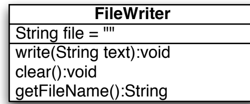

Figure 6.3: The FileWriter class. Used by most of the main classes in the CoPS framework prototype, to logs different happenings.

figure 6.3, and only has methods for writing to a file, clearing a file, and getting the name of the file.

6.4.1 The Modified CoPSTaskResponsible

Agents extending the *CoPSTaskResponsible* abstract class become the TR. As described before, the TR is responsible for seeing to that a requested problem is solved at run-time. TR analyzes the problem, gathers a team of PSs, administers the problem solving process, combine partial results of a PSM into an integrated solution, and finally returns a solution (or a failure specification) to the requester of the problem. Our implemented *CoPSTaskResponsible* is capable of doing all this. A modified UML class diagram of the *CoPSTaskResponsible* is illustrated in figure 6.4. The figure also indicates what classes are new, what classes are modified, and how the classes are extended or modified. As the *CoPSTaskResponsible* is a subclass of the *CoPSAgent*, it is also a subclass of the Jade *Agent* class.

The implementation of the TR and its behaviours is complex, and this is reflected in the number of implemented behaviour classes, and the long list of methods in the *CoPSTaskResponsible* class. The behaviour classes are not unaffected by each other, they interact and depend on each other in different ways. In the following we describe how the *CoPSTaskResponsible* class, and its behaviour classes realize the behaviour of the TR.

First, *setup()* in *CoPSTaskResponsible* is called. The *setup()* method is an empty placeholder for application specific code provided by the JADE *Agent* class. It is used for setting up the agent - prepare it for action in the agent environment. During the setup a message template is defined and the behaviours implemented by *DFRegisteringBehaviour* and *HandleProblemSolvingRequestBehaviour* is added to the agent (*CoPSTaskResponsible*). The message template is used for filtering the messages that should be handled by the *HandleProblemSolvingRequestBehaviour*.

The first behaviour to be executed by the agent is *DFRegisteringBehaviour*. This behaviour makes sure that the TR registers its capabilities with the Directory Facilitator (Matchmaker) implemented by the JADE framework. The next behaviour is the *HandleProblemSolvingRequestBehaviour*, which is not executed until the agent receives a message with a problem solving request that matches the template defined during the setup. The *HandleProblemSolvingRequestBehaviour* class extends the JADE *AchieveReResponder* behaviour class. When a problem solving request is received, the problem is analyzed by the abstract method *reviewProblem()* in *CoPSTaskResponsible*. If the problem is not ac-

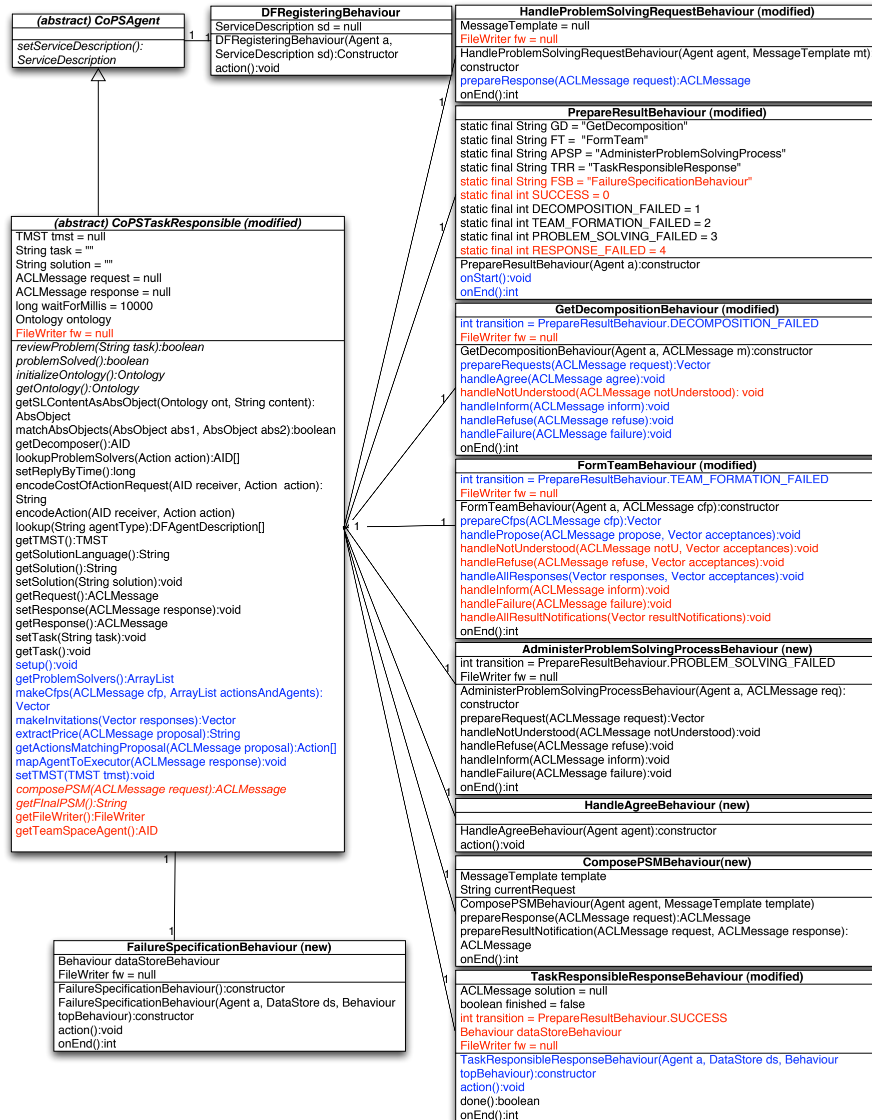

Figure 6.4: A modified UML diagram of the CoPSTaskResponsible agent and its behaviours. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes.

cepted, a refuse message is sent to the requester. If the problem is accepted, the *prepareResultNotification()* should be called. This is not the case here as this state of the *AchieveReResponder* behaviour is exchanged with a new behaviour, represented by the *PrepareResultBehaviour* class. This behaviour is added to the agent.

The *PrepareResultBehaviour* class extends the JADE *FSMBehaviour*. It defines a set of states represented by new behaviours, which are implemented by the classes: *GetDecompositionBehaviour*, *FormTeamBehaviour*, *AdministerProblemSolvingProcessBehaviour*, *TaskResponsibleResponseBehaviour* and *FailureSpecificationBehaviour*. The first state is implemented by the *GetDecompositionBehaviour* class.

GetDecompositionBehaviour

The *GetDecompositionBehaviour* class extends the JADE *AchieveREInitiator* class. This class realizes the TR's part of the conversation with the Decomposer (DEC) to get a TMST, representing the decomposition of a problem. The TMST is kept by an instance of the *TMST* class.

First, the Directory Facilitator (Matchmaker) is requested for all DEC agents in the system, and one of these is chosen. This is done through the method *getDecomposer()* in *CoPSTaskResponsible*. If an appropriate agent is found, a request message containing the initial problem as the content is sent to the DEC agent. If an inform message with a *TMST* instance is received from DEC, the *TMST* is stored in the *CoPSTaskResponsible* with the method *setTMST()*. Then, the transition returned from this behaviour is *SUCCESS*. If the TR receives a not-understood -, refuse-, or failure message from the DEC, the transition returned from this behaviour is *DECOMPOSITION_FAILED*.

FormTeamBehaviour

If the transition from the running of *GetDecompositionBehaviour* is *SUCCESS*, the state of the *PrepareResultBehaviour*, implemented by the *FormTeamBehaviour* class is reached. This class extends the JADE *ContractNetInitiator* class.

The *prepareCfps()* method of the *ContractNetInitiator* class is implemented to define the call-for-proposal messages that are sent to the proper PS agents in the system. The proper agents are found by extracting the actions (leaf -nodes) from the TMST, and requesting the Directory Facilitator for agents capable of performing those actions. This is realized by the *getProblemSolvers()* method in *CoPSTaskResponsible*, which returns a list of action-PS agent pairs. This list is used in *prepareCfps()* to make the call-for-proposal messages.

The implemented method *handleAllResponses()* is called when all the responses are received. For all of the received proposal-message the sender (PS) agent is mapped to the an *Executor* in the *TMST* by the *CoPSTaskResponsible* method *mapAgentToExecutor()*. Next, the solution space of the *TMST* is generated, and the best solution is found, by calling the *TMST* methods *generateSolutionSpace()* and *generateSolution()*. If a solution space is found, the *CoPSTaskResponsible* method *makeInvitations()* is called. The method returns a list of invitations for the agents part of the chosen solution in the *TMST*. Finally, the accept-proposal messages are sent for the PS agents as invitations

to join the team, and reject-proposal messages are sent for the PS agents that was not invited to join the team. If a solution in the *TMST* could not be found, all of the agents that sent a proposal receives a reject-proposal message, and the behaviour ends with the transition *TEAM_FORMATION_FAILED*.

Finally, the implemented method *handleAllResultNotifications()* goes through the results received from the agents that were invited to join the team. An inform message means that the PS accepts the invitation, a failure message means that the PS will not join the team. If a failure message is received, the current solution of the TMST is not fulfilled, and the transition from the behaviour is *PrepareResultBehaviour.TEAM_FORMATION_FAILED*. If no failure messages are received the transition from the behaviour is *PrepareResultBehaviour.SUCCESS* - a team of PS agents capable of solving the initial task through cooperation is successfully formed.

AdministerProblemSolvingProcessBehaviour

If the transition from the running of *GetDecompositionBehaviour* is *SUCCESS*, the state of the *PrepareResultBehaviour*, implemented by the *AdministerProblemSolvingProcessBehaviour* class is reached. This class extends the JADE *AchieveREInitiator* class.

This class implements the behaviour of the TR involving requesting the TEAM SPACE (TS) Agent to initialize and solve a problem in the TEAM SPACE (TS) Structure. The request sent by the behaviour contains the *TMST* instance kept by *CoPSTaskResponsible*. The *TMST* is gotten by calling the method *getTMST()*, and the TS Agent is found by calling the method *getTeamSpaceAgent()*, both in *CoPSTaskResponsible*. If a TS Agent sends a refuse message or a not-understood message the transition from this behaviour is *PROBLEM_SOLVING_FAILED*. The state of the *AchieveREInitiator*, handling an agree message is replaced by a behaviour implemented by the class *HandleAgreeBehaviour*.

When the TS Agent agrees to solve a problem in the TS Structure, the *HandleAgreeBehaviour* extending JADE *OneShotBehaviour* makes sure that a new behaviour implemented by the class *ComposePSMBehaviour* is added to *CoPSTaskResponsible*. The *ComposePSMBehaviour* extending JADE *AchieveREResponder* is executed when a request for composing a PSM is received from the TS Agent. If the content of the request is understood an agree message is sent, if not a not-understood message is sent. Then the result is prepared by calling the abstract method in *CoPSTaskResponsible* - *composePSM()*. In the implemented *onEnd()* method of *AdministerProblemSolvingProcessBehaviour*, a check is done to find out if the requested PSM is the final PSM (last PSM to compose in *TMST*). This is done by calling *getFinalPSM()* in *CoPSTaskResponsible*. If it was not the final PSM, the *composePSMBehaviour* is added again, and the TR agent waits for further PSMs to compose. If it was the final PSM, we are taken back to *AdministerProblemSolvingProcessBehaviour*. Now the TR agent waits for the TS Agent to return the final solution of the problem solving in the TEAM SPACE. If a failure message is received the behaviour ends with the transition *PROBLEM_SOLVING_FAILED*. If an inform message is received, containing the solution to the initial task, the behaviour ends with the transition *SUCCESS*.

TaskResponsibleResponseBehaviour

If the transition from the running of *AdministerProblemSolvingProcessBehaviour* is *SUCCESS*, the state of the *PrepareResultBehaviour* implemented by the *TaskResponsibleResponseBehaviour* is reached. This class extends the JADE *SimpleBehaviour*.

The purpose of this behaviour is to generate a message containing the solution to the initial problem. The message is generated in the implemented *action()* method, by calling *getSolution()* and *getSolutionLanguage* in *CoPSTaskResponsible*. The generated message is put back into the message - "system" of *HandleProblemSolvingRequestBehaviour*, which finally can return the result to the requester of the problem, as an inform message. When the result is returned, the behaviour implemented by *HandleProblemSolvingRequestBehaviour* is added to the agent again, and the TR starts again to wait for new problem solving request messages.

FailureSpecificationBehaviour

The *FailureSpecificationBehaviour* also implements a state of the *PrepareResultBehaviour*. This state is reached if a behaviour representing an other state in the *PrepareResultBehaviour*, ends with one of the following transitions:

- *DECOMPOSITION_FAILED*
- *TEAM_FORMATION_FAILED*
- *PROBLEM_SOLVING_FAILED*
- *RESPONSE_FAILED*

The purpose of this behaviour is to generate a message containing a failure specification. The message is generated in the implemented *action*, by using the information about the transition that led to this state. The generated message is put back into the message - "system" of *HandleProblemSolvingRequestBehaviour*, which finally can return the result to the requester of the problem, as a failure message.

6.4.2 The Modified CoPSProblemSolver

Agents extending the *CoPSProblemSolver* abstract class become PSs. As described before, the PSs are the workers in the system, and are capable of solving specific problems or performing actions. They may join the team, that is organized by the TR, by responding to call-for-proposal messages. When they are part of a team, they wait for and respond to problem solving requests from a TS Agent. The extended *CoPSProblemSolver* also implements the possibility of a PS joining several teams, or at least performing several actions, in each behavioural cycle.

As the *CoPSProblemSolver* is a subclass of the *CoPSAgent*, it is also a subclass of the JADE *Agent* class. A modified UML class diagram of the *CoPSProblemSolver* and its behaviour classes is illustrated in figure 6.5. The figure also indicates what classes are new, what classes are modified, and how the classes are extended or modified.

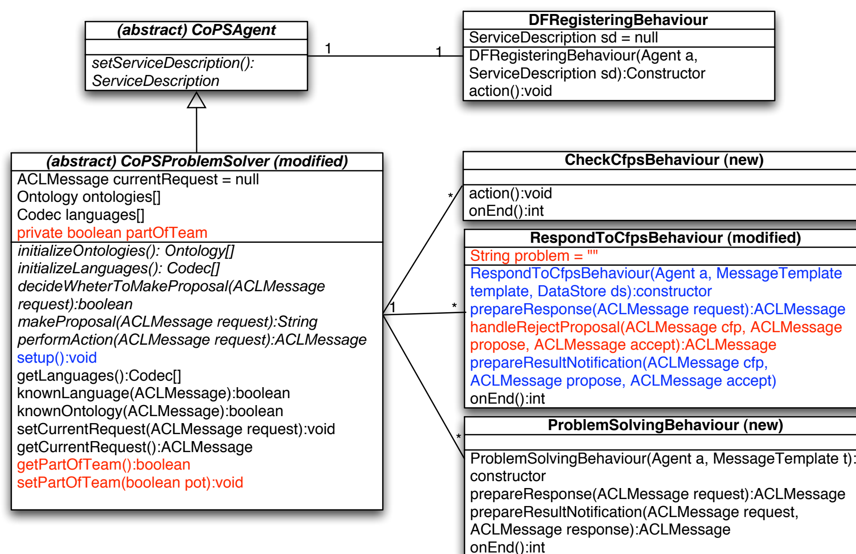

Figure 6.5: A modified UML diagram of the CoPSProblemSolver agent and its behaviours. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes.

First, *setup()* in *CoPSProblemSolver* is called. As mentioned before, the *setup()* method is an empty placeholder for application specific code provided by the JADE *Agent* class, used for setting up the agent. During the setup the agent's languages and ontologies are initialized by calling *initializeLanguages()* and *initializeOntologies()*, and two behaviours are added to the agent which are *DFRegisteringBehaviour* and *CheckCfpsBehaviour*.

The first behaviour to be executed by the agent is the *DFRegisteringBehaviour* - the PS agent registers its capabilities (what actions it can perform) with the Directory Facilitator (Matchmaker). The next behaviour is *CheckCfpsBehaviour* which extends the JADE *SimpleBehaviour* class. This behaviour gathers all the incoming call-for-proposal messages, and when there are no more incoming messages, a JADE *ParallelBehaviour* is added to *CoPSProblemSolver*. The *ParallelBehaviour* has one sub-behaviour for each of the call-for-proposal messages. The sub-behaviours are implemented by *RespondToCfpsBehaivour*. The *ParallelBehaviour* makes sure that the call-for-proposal-messages are handled in parallel by the *RespondToCfpsBehaviour* instances. *ParallelBehaviour* finishes when all of the *RespondToCfpsBehaviour* instances and their children (behaviours added to the agent from this behaviour) are done.

RespondToCfpsBehaviour extends the JADE *ContractNetResponder* class. A response to the call-for-proposal message received from a TR is prepared in *prepareResponse()*. The agent decides whether to make a proposal, by calling *decideWhetherToMakeProposal()*. The proposal is made by calling *makeProposal()*, which is an abstract method in *CoPSProblemSolver*. If a proposal is not made or the agent receives a reject-proposal message, this behaviour is done. If a proposal is made and sent, and the agent eventually receives an accept-proposal message, containing an invitation to join a team, *prepareResultNotification()* starts preparing the answer. The agent makes and sends an inform-message to tell that it joins the team and calls *setPartOfTeam(true)* in *CoPSProblemSolver*. And a new behaviour is added to the agent, implemented by the *ProblemSolvingBehaviour* class. If the agent decides not to join the team, it makes and sends a failure-message and the behaviour is done.

ProblemSolvingBehaviour extends the JADE *AchieveREResponder* class. This class handles problem solving request messages from a TS Agent. When a request message is received, it is handled by *prepareResponse()*. If the PS agent understands the content of the message, and is part of a team (*getPartOfTeam()*), it replies with an agree message, if not it replies with a refuse message (not part of a team) or a not-understood message (did not understand the content). If a refuse - or not-understood message was sent the behaviour is done. If an agree message was sent PS starts preparing the result in *prepareResultNotification()*. It performs the action by calling the abstract method *performAction()* in *CoPSProblemSolver*, and returns the results from the execution in an inform message. The behaviour is done.

When all the sub-behaviours of *ParallelBehaviour* are done, *CheckCfpsBehaviour* is added to the agent again, and it starts waiting for new call-for-proposal messages.

6.4.3 The Modified TMST

The *TMST* class keeps the model of a TMST, like described in section 6.2. The UML diagram of the *TMST* class is illustrated in figure 6.6. As one can

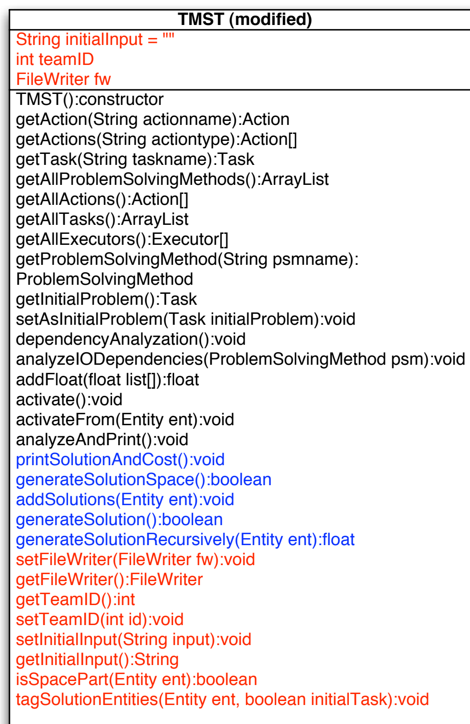

Figure 6.6: A modified UML diagram of the TMST main class. The former version is found in [34]. Red text means new variables/methods in modified classes, blue text means modified variables/methods in modified classes, black text means not modified variables/methods in old classes or variables/methods in new classes.

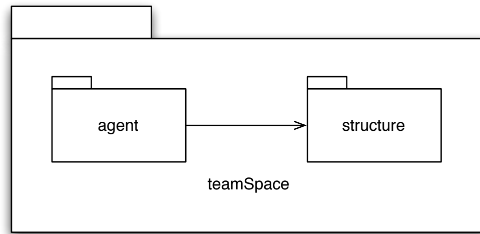

Figure 6.7: Our package structure for the TEAM SPACE implementation. *agent* contains TS Agent classes, and *structure* contains TS Structure classes.

see in the diagram, there has been done quite some modifications to this class, but most of these are due to the corrections listed in the previous section. Modifications considering "CoPS problem solving process extensions" involves the three instance variables *initialInput, teamID* and *fw*, and methods for getting and setting them.

The *initialInput* is a string, representing the actual input for the initial task of a *TMST* instance. This input might be *(Patient :name paul)* for the initial task *doCheckup*. The input related to the initial task, which is stored in *TMST* from before is *(Patient :name x)*, in this case.

Each instance of *TMST*, created by the Decomposer, keeps a decomposition of a specific problem that will be solved by a team of PSs. The *teamID* becomes the identification of this team, and it relates an instance of *TMST* to its team.

6.5 The TEAM SPACE

As described in chapter 4 the TEAM SPACE is comprised of a TEAM SPACE (TS) agent and a TEAM SPACE (TS) Structure. Classes that implements the TEAM SPACE are contained in the *teamSpace* package, classes that implements the TS Agent are contained in the *agent* package, and classes that implements the TS Structure are contained in the *structure* package. This is illustrated in figure 6.7. The figure also shows that classes from *agent* access classes from *structure*. The *TeamSpaceAgent* class (implementing the TS Agent) keeps instances of the *TeamSpace* class (implementing the TS Structure).

All the parts of the TEAM SPACE architecture that was proposed in chapter 4 have been implemented, except from the part covering dynamic team formation and re-planning of tasks. Our TEAM SPACE architecture covers the functional requirements listed in chapter 3. Dynamic team formation and re-planning of tasks are not part of these requirements. This indicates that the implementation of the TEAM SPACE also covers the functional requirements.

6.5.1 The TEAM SPACE agent

Main responsibilities of the TS Agent is to be TR's and PSs' interface to the TS Structure. A TS Agent initializes the TS structure on request from the TR, it notifies TRs and PSs about actions that need to be executed and PSMs that

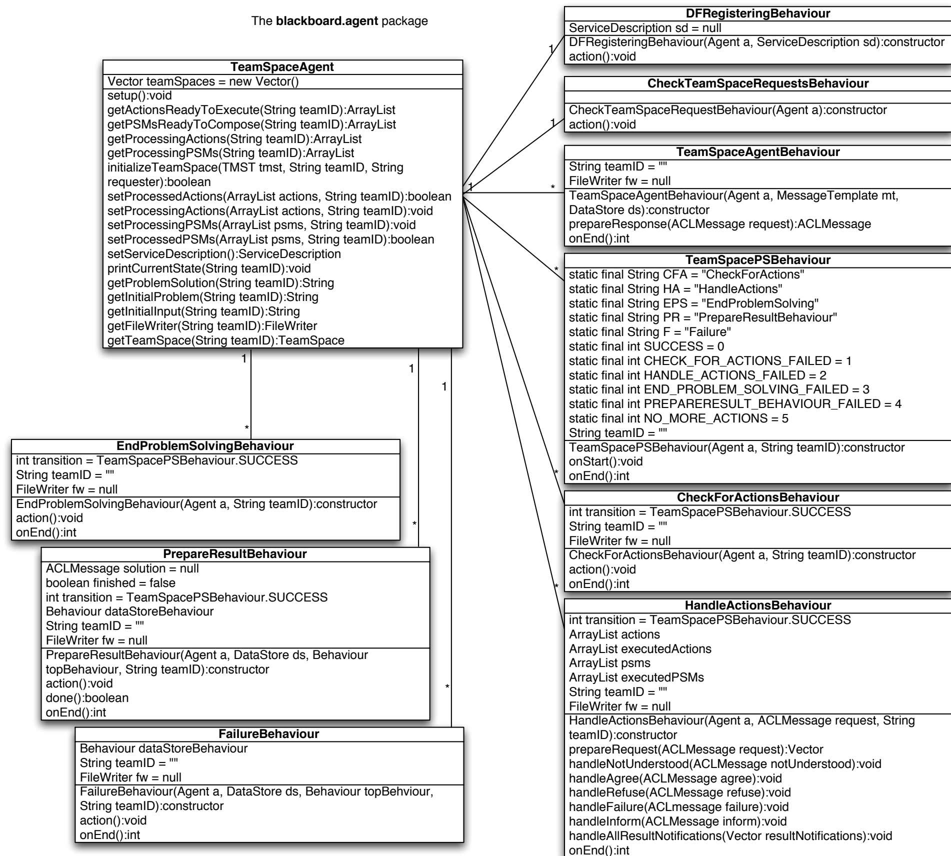

Figure 6.8: An UML class diagram showing the classes that implements the TS Agent. These classes are contained in the package: *teamSpace.agent*

need to be composed, it updates the TS structure with results from execution and composition, and it finally extracts the result to the initial problem and returns it to the TR. In addition, if there is only one TS Agent in the system, it might have to keep track of several TEAM SPACE structures at the same time and communicating with different problem solving teams.

The TS Agent is implemented by the *TeamSpaceAgent* class, which is not abstract and does not extend the *CoPSAgent* class like *CoPSTaskResponsible* and *CoPSProblemSolver* does. But, *TeamSpaceAgent* is a subclass of the JADE *Agent* class. A UML class diagram of the *TeamSpaceAgent* class and the classes implementing its behaviours is illustrated in figure 6.8. The *TeamSpaceAgent* class keeps a list of *TeamSpace* objects, and each of these objects is dedicated for solving a specific problem. *TeamSpace* objects are identified by the *teamID*, which is related to the team of PS agents that shall solve the problem. A *CoPSAgent* instance is the PS's and TR's interface to the *TeamSpace* objects representing the TS Structures. Since the *CoPSAgent* instance may be involved with many problem solving teams at the same time, the *teamID* must always me submitted when one of the methods dealing with a specific *TeamSpace* object is called. In the following, we describe how the *CoPSAgent* class and its behaviour classes realize the behaviour of the TS Agent.

First, *setup()* in *TeamSpaceAgent* is called. As mentioned before, the *setup()* method is an empty placeholder for application specific code provided by the JADE *Agent* class, used for setting up the agent. During the setup, two behaviours are added to *TeamSpaceAgent* which are implemented by *DFRegisteringBehaviour* and *CheckTeamSpaceRequestsBehaviour*.

The DFRegisteringBehaviour and CheckTeamSpaceRequestsBehaviour classes

The first behaviour to be executed by the *TeamSpaceAgent* is the *DFRegisteringBehaviour* - the *TeamSpaceAgent* registers its capabilities with the Directory Facilitator (Matchmaker). The next behaviour is *CheckTeamSpaceRequestsBehaviour* which extends the JADE *SimpleBehaviour* class. This behaviour gathers all incoming *TeamSpace* request messages. When there are no more incoming messages, a JADE *ParallelBehaviour* is added to *TeamSpaceAgent*. *ParallelBehaviour* has one sub-behaviour for each of the *TeamSpace* request messages. The sub-behaviours are implemented by the *TeamSpaceAgentBehaviour* class. *ParallelBehaviour* makes sure that the *TeamSpace* request messages are handled in parallel by the *TeamSpaceAgentBehaviour* instances, and it finishes when all of these *TeamSpaceAgentBehaviour* instances and their children (behaviours added to the agent from this behaviour) are done.

The TeamSpaceAgentBehaviour class

TeamSpaceAgentBehaviour extends the JADE *AchieveReResponder* class. A response to the *TeamSpace* request message received from the TR is prepared in *prepareResponse()*. The content of the request message is an instance of a *TMST*. Preparing the response involves trying to initialize a *TeamSpace* object by calling the method *initializeTeamSpace()* in *TeamSpaceAgent*, and submitting the *TMST*, a *teamID* and the name of the sender of the request (the TR). *initializeTeamSpace()* creates a new *TeamSpace* object and adds it to the list of

TS Structures kept by *TeamSpaceAgent*. Then the *TeamSpace* object is initialized by calling *initialize()* on the object. If the initialization ends successfully, *TeamSpaceAgent* responds with an agree message. If the content of the message was not understood, *TeamSpaceAgent* responds with a not-understood message. And if the *TeamSpace* could not be initialized textitTeamSpaceAgent responds with a refuse message. In the two last cases the behaviour is done. In the first case, where an agree message was sent, a new behaviour is added to handle the state of *TeamSpaceAgentBehaviour* that prepares the result notification. *TeamSpacePSBehaviour* implements the added behaviour.

The TeamSpacePSBehaviour class

The *TeamSpacePSBehaviour* class extends the JADE *FSMBehaviour*. It defines a set of states represented by different behaviours, which are implemented by the classes: *CheckForActionsBehaviour, HandleActionsBehaviour, EndProblemSolvingBehaviour, PrepareResultBehaviour* and *FailureBehaviour*. It also defines a set of transitions that decides what is the next state after the current state. The next sections describe the different states of this behaviour, and how the transition between states work. This behaviour is done (and thereby the *TeamSpaceAgentBehaviour* that started it), when one of the final states are reached. The final states are implemented by *PrepareResultBehaviour* and *FailureBehaviour*. The first state is implemented by the *CheckForActionsBehaviour* class.

The CheckForActionsBehaviour class

If the *TeamSpacePSBehaviour* is just started, or if the running of *HandleActionsBehaviour* ends with the transition *SUCCESS*, the state of the *TeamSpacePSBehaviour* that is implemented by *CheckForActionsBehaviour* is reached. This class extends the JADE *OneShotBehaviour* class. The *action()* method calls *getActionsReadyToExecute()* and *getPSMsReadyToCompose* in *TeamSpaceAgent* (submitting a teamID), which return a list of actions and a list of PSMs respectively. If the lists are empty, the behaviour is done and the transition returned is *NO_MORE_ACTIONS*. If the lists are not empty, the actions and PSMs that are ready to be processed are set in the correct *TeamSpace* by calling *setProcessingActions()* and *setProcessingPSMs()* in *TeamSpaceAgent*. Then, the behaviour is done and the returned transition is *SUCCESS*.

The HandleActionsBehaviour class

If the transition from *CheckForActionsBehaivour* is *SUCCESS*, the state of *TeamSpacePSBehaviour* implemented by *HandleActionsBehaviour* is reached. *HandleActionsBehaviour* extends the JADE *AchieveREInitiator* class and its purpose is to support a conversation with PSs and TRs, that will process the actions and PSMs found in the previous state.

Requests for the PSs and TRs are prepared in the method *prepareRequests()*. The actions and PSMs that are ready to be processed are gotten from the correct *TeamSpace* by calling *getProcessingActions()* and *getProcessingPSMs()* in *TeamSpaceAgent*, and stored in the lists *actions* and *psms*. An action object contains the address of a PS and information the PS needs to execute the action.

A PSM object contains the address of a TR, and information the TR needs (partial results, input) to compose that PSM. Request messages are made, and sent for all of the action and PSM objects.

If only agree messages are received as responses to the request messages, textitTeamSpaceAgent starts waiting for the inform messages. If a not-understood - or a refuse message is received, textitTeamSpaceAgent will not be able to complete the problem solving in the *TeamSpace*, and some action should be taken. For now, the behaviour ends with the transition *HANDLE_ACTIONS_FAILED*.

The inform messages received are processed by *handleInform()*. Inform messages keeps the output from an action or PSM, produced by a PS or TR. For each inform message, one extracts which action/psm it represents, this action/psm object is found in the *list actions/psms*, the object is manipulated by adding the output, and then it is put into the *executedActions/executedPSMs* list. When all the inform messages are received, *handleAllResultNotifications()* is called. The processed actions and PSMs are added to the correct *TeamSpace* by calling *setProcessedActions()* or *setProcessedPSMs()* in *TeamSpaceAgent*. If the actions and PSMs are successfully set, what is done in this state is written to the file kept by the *FileWriter*, and the behaviour ends with the transition *SUCCESS*. If an error occurs during the setting of actions and PSMs the behaviour ends with the transition *HANDLE_ACTIONS_FAILED*.

The EndProblemSolvingBehaviour class

If *CheckForActionsBehaviour* ends with the transition *NO_MORE_ACTION* the state of *TeamSpacePSBehaviour* implemented by *EndProblemSolvingBehaviour* is reached. *EndProblemSolvingBehaviour* extends the JADE *OneShotBehaviour*. In the *action()* method *getProblemSolution()* is called in *TeamSpaceAgent* to check if there is a solution to the initial problem. If there is a solution the behaviour ends with the transition *SUCCESS*, and if there is no the behaviour ends with the transition *END_PROBLEM_SOLVING_FAILED*.

The PrepareResultBehaviour class

If *EndProblemSolvingBehaviour* ends with the transition *SUCESS*, the state of *TeamSpacePSBehaviour* implemented by *PrepareResultBehaviour* is reached. *PrepareResultBehaviour* extends the JADE *OneShotBehaviour*. In the *action()* method a message containing the solution to the initial problem is generated, using the information from *getProblemSolution()* in *TeamSpaceAgent*. The message is put back into the message - "system" of *TeamSpaceAgentBehaviour*, which finally can return the result to the requester of the team space, as an inform message. When the result is returned, the *TeamSpacePSBehaviour* is done, since *PrepareResultBehaviour* is a final state.

The FailureBehaviour class

The *FailureBehaviour* also implements a final state of *TeamSpacePSBehaviour*. This state is reached if a behaviour representing an other state in *TeamSpacePSBehaviour* ends with one of the following transitions:

- *CHECK_FOR_ACTIONS_FAILED*

- *HANDLE_ACTIONS_FAILED*
- *END_PROBLEM_SOLVING_FAILED*

The purpose of this behaviour is to generate a message containing a failure specification. The message is generated in the implemented *action()* method, by using the information about the transition that led to this state. The generated message is put back into the message - "system" of *TeamSpaceAgentBehaviour*, which finally can return the result to the requester of *TeamSpace*, as a failure message. When the result is returned, the *TeamSpacePSBehaviour* is done, since *PrepareResultBehaviour* is a final state.

As described in the two final states, when the result is returned, the *TeamSpacePSBehaviour* is done, and thereby the *TeamSpaceAgentBehaviour* are done as well. When all of the other *TeamSpaceAgentBehaviour*s, that are children of *ParallelBehaviour*, added by *CheckTeamSpaceRequestsBehaviour*, are done also, the *ParallelBehaviour* is done, and thereby *CheckTeamSpaceRequestsBehaviour* is done. The *CheckTeamSpaceRequestsBehaviour* is added to the TS Agent again and it starts all over to wait for new *TeamSpace* request messages.

6.5.2 The TEAM SPACE structure

The TEAM SPACE (TS) structure architecture was described in chapter 4, section 4.1.3. The TS Structure is comprised of four different components: Plan Library, Result Library, Goal Stack and Rule Base Container. The TEAM SPACE (TS) Problem Solving State gives a view of information kept by the Result Library and the Goal Stack. The main responsibilities of the TS Structure is:

- ... to convert the knowledge in a *TMST* into rules that will guide the problem solving process steps: *solving of subtasks* and *integration of partial results.*
- ... to store results from PSs executing actions and TR composing a PSM's partial results.
- ... to reason about these results and using the rules to infer the correct state of the problem solving process.

The TS Structure is implemented by five classes: *TeamSpace, PlanLibrary, ResultLibrary, RuleBase* and *GoalStack*. The classes are illustrated by an UML diagram in figure 6.9. The *PlanLibrary* class implements the Plan Library in the TS Structure architecture, the *GoalStack* class implements the Goal Stack, the *ResultLibrary* class implements the Result Library and the *RuleBase* class implements the Rule Base Container. Finally, the *TeamSpace* class implements the TS Structure, containing references to the other components.

TEAM SPACE structure classes

The *TeamSpace* class realizes all of the functions provided by the TS Structure through manipulating the other classes, and thus implements a single interface to the TS Structure functionality. One instance of the *TeamSpace* structure

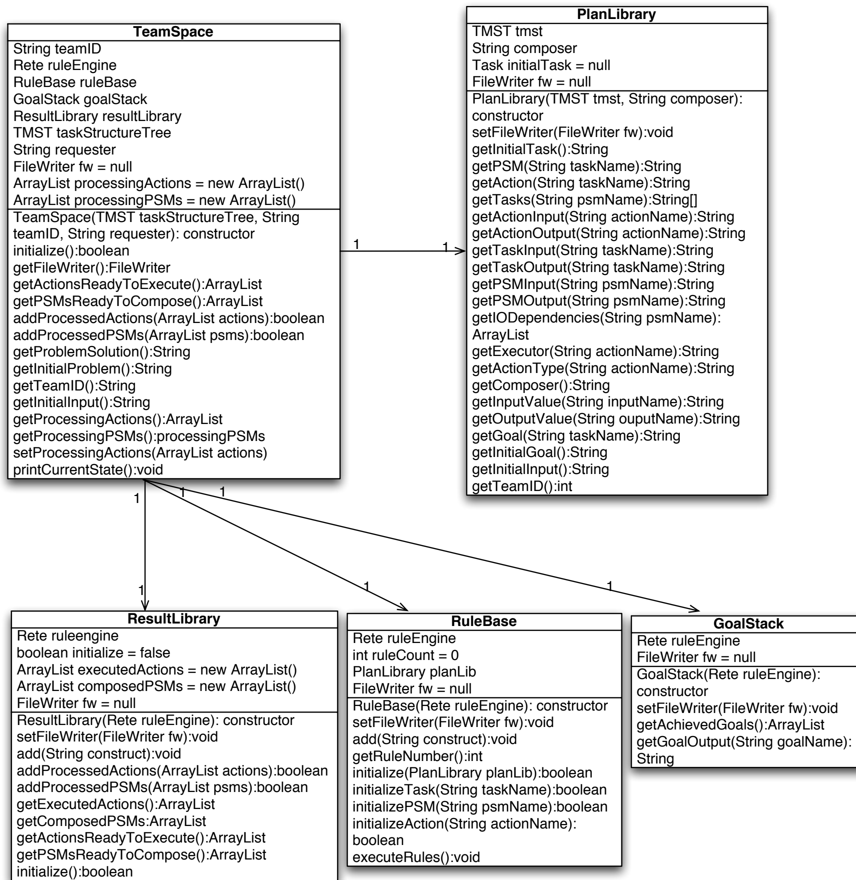

Figure 6.9: An UML class diagram showing the classes that implements the TS Structure. These classes are contained in the package: *teamSpace.structure*

is created for each problem, by calling the *TeamSpace* constructor with an instance of *TMST*, a teamID representing the team of problem solvers, and the name of the requester of the problem solving. The *TeamSpace* then instantiates one instance of each of the TS structure classes, and a Jess *Rete* class, which represents a Java interface to the Jess rule engine. The *Rete* class has its own knowledge base and rules.

The *PlanLibrary* class is instantiated by calling the constructor with an instance of a *TMST*, and the name of the composer of PSMs (TR). Basically, *PlanLibrary* serves as an interface to a *TMST*. It has different methods to extract and manipulate information from the *TMST*.

The *RuleBase* class is instantiated by calling the constructor with the *Rete* class instance created by *TeamSpace*. *RuleBase* serves as an interface to the rules in the Jess rule engine. The *initialize...()* methods in *RuleBase* converts information from a *TMST*, kept by the *PlanLibrary*, into rules describing the problem solving process. In addition, there are methods for adding and executing rules in the *Rete* instance. After a rule is created by one of the *initialize...()* methods, it is added to the *Rete* instance.

The *ResultLibrary* class is instantiated by calling the constructor with the same instance of the *Rete* class as was used in the *RuleBase* class. Only, this class serves as an interface to the facts stored in the knowledge base of the Jess rule engine. *ResultLibrary* has methods for adding facts to the Jess knowledge base, and for extracting certain types of facts from it.

The *GoalStack* class is also instantiated by calling the constructor with the *Rete* instance, and it is the interface to the part of the Jess knowledge base (kept by the *Rete* instance) that keeps facts about achieved goals. *GoalStack* has methods to extract information about achieved goals from the *Rete* instance.

TEAM SPACE structure main functionality

During the description of the TEAM SPACE architecture in chapter 4, we found four main functions that needed to be covered by the TS structure. The functions are: To initialize the TS structure for assisting in solving a specific problem, to check for actions and PSMs that are ready to be executed and composed (involves updating and extracting the problem solving state), to update the Result Library and to get the solution to the initial problem. All of these functions are implemented by the TS structure classes.

The TS Structure is initialized by creating a new instance of the *TeamSpace* class, which again instantiates one instance of each of the other TS Structure classes. Then *initialize()* is called on the *TeamSpace*, which again calls *initialize()* on its instance of *ResultLibrary* and on its instance of *RuleBase*. *initialize()* in *ResultLibrary* adds a set of Jess templates to the *Rete* (rule-engine) knowledge base, defining how the facts should be represented. And *initialize()* in *RuleBase* traverses the *TMST* by extracting information from *PlanLibrary*, and creates a set of Jess rules which it adds to the *Rete* (rule-engine) rules.

The check for actions and PSMs are done separately. *getActionsReadyToExecute()* in *TeamSpace* returns an ArrayList of actions that are ready to be executed. This is done by first calling *executeRules()* in *RuleBase*. The execution of rules may add new facts to the Jess knowledge base. Then, *getActionsReadyToExecute()* is called in *ResultLibrary*, which extracts facts from the knowledge base which are based on the template *(ready_to_execute (slot action)*

(slot action_type) (slot input) (slot executor)). getPSMsReadyToCompose() in *TeamSpace* returns an ArrayList of PSMs that are ready to be composed. Now, the rules are already executed. *getPSMsReadyToCompose* in the *ResultLibrary* is called, which extracts facts from the knowledge base which are based on the template *(ready_to_compose (slot psm) (slot input) (mulitslot partial_results) (slot executor)).*

When actions are executed and PSMs are composed, the *ResultLibrary* needs to be updated. Then *addProcessedActions()* and *addProcessedPSMs()* are called in the *TeamSpace* and these methods respectively call *addProcessedActions()* and *addProcessedPSMs()* in *ResultLibrary. addProcessedActions()* goes through the submitted list of processed actions and adds facts on this form; *(executed (action "an action name")(output "the action output"))*, to the Jess knowledge base. *addProcessedPSMs()* goes through the submitted list of composed PSMs and adds facts on this form; *(composed (psm "a psm name")(output "the psm output"))*, to the Jess knowledge base.

When there are no more actions and PSMs that is not processed, and there is a solution to the problem, the method *getProblemSolution()* returns the solution to the initial task (the output of the achieved goal connected to the initial task). This is done by calling *getGoalOutput()* in *GoalStack*, and submitting the initial goal which is gotten from calling *getInitialGoal()* in *PlanLibrary.*

The TS structure architecture has a part called the TS Problem Solving State. TS Problem Solving State is not a component, but rather a view of the facts stored in the knowledge base kept by the *Rete* instance. *printCurrentState()* in *TeamSpace* writes the current TS Problem Solving State once each problem solving cycle to the file represented by the *FileWriter*. A cycle lasts from the rules are run, to the next time they are run. Types of information part of the TS Problem Solving State and how they are found:

- **Achieved goals**. A list of achieved goals is returned from *getAchievedGoals()* in *GoalStack.* The method extracts knowledge from the knowledge base which is based on the template *(achieved (slot goal) (slot output)).*

- **Executed actions**. A list of executed actions is returned from *getExecutedActions()* in *ResultLibrary*, which stores the executed actions in a list.

- **Composed PSMs**. A list of composed PSMs is returned from *getComposedPSMs()* in *ResultLibrary*, which stores the composed PSMs in a list.

- **Actions ready to execute**. A list of actions ready to be executed is returned from *getActionsReadyToExecute()* in *ResultLibrary* (described above).

- **PSMs ready to compose**. A list of PSMs ready to be composed is returned from *getPSMsReadyToCompose()* in *ResultLibrary* (described above).

When we are talking about lists of goals, actions and PSMs we refer to objects which are instances of some supporting classes. These classes are shortly described next.

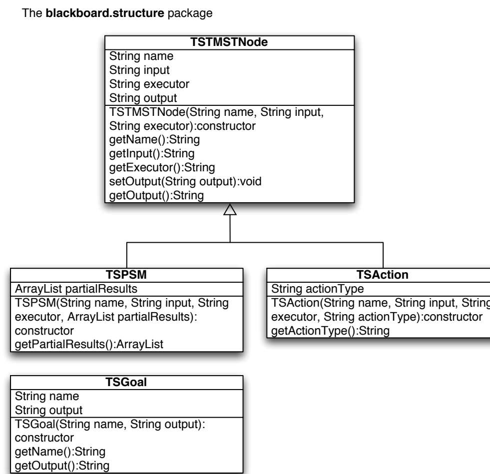

Figure 6.10: Classes representing some of the TMST-units. These are used by the other classes implementing the TS Structure and the TS Agent. These classes are contained in the package: *teamSpace.structure*

TEAM SPACE supporting classes

A UML class diagram of the supporting classes is illustrated in figure 6.10. The classes are *TSPSM* and *TSAction* which are subclasses of *TSTMSTNode*, and the class *TSGoal*. These classes represent PSMs, actions and goals and keep some of their attributes. This is done to make it easier for the other TEAM SPACE structure classes to handle and reason about these concepts outside the Jess rule-engine. When a CoPS ontology is properly implemented, the ontology should be used to represent the PSMs, actions and goals. The supporting classes may also be seen as manageable substitutes for the nodes part of a TMST.

6.6 Summary

This chapter has given a detailed description of how the corrections and extensions to the CoPS framework, described in chapter 3, have been implemented by modifying and extending the pre-existing CoPS framework Prototype. The main extension to the prototype was the shared memory structure, the TEAM SPACE. All of the components of the TEAM SPACE architecture, that was proposed in chapter 4, has been implemented.

In chapter 7, the implemented CoPS framework prototype is used for implementing part of a medical domain check-up system. There we get to prove that our implementation covers the functionality proposed for the CoPS corrections and extensions, that are described earlier, in conceptual terms.

Chapter 7

Experimentation and Results

The CoPS framework prototype is an API, that may be used to let programmers easily implement their applications of cooperative distributed problem solving agents. The prototype that is a the result of the work with this master thesis, is described in the previous chapters.

In this chapter, we use the CoPS framework prototype to implement an application from a medical domain. The application is concerned with performing a checkup at a health center. In section 7.1 we describe the steps needed to be taken when implementing an application using the CoPS framework prototype. Section 7.2 presents the different implementation steps of our checkup example application. The test-run of our application is described in section 7.3. And finally, the results from the test-run are listed in section 7.4.

This chapter lay the grounds for making conclusions and describing the achieved results of the work with this master thesis, in chapter 8.

7.1 How to Implement an Application Using the CoPS Framework

The CoPS framework has three main parts: CoPS agents, TMST and TEAM SPACE. Both CoPS agents and the TMST needs to be implemented or specialized by the application developer using the CoPS framework prototype. The TEAM SPACE is generic and may be used in any application domain.

To implement an application with cooperating agents using our CoPS framework prototype, a minimum set of tasks have to be done [34]:

- The agents playing the necessary roles (Task Responsible - TR, Problem Solver - PS, Decomposer - DEC) must be extended, and their abstract methods must be implemented.

- A JADE ontology describing the domain of the problem being solved must be implemented.

- A TMST based on the implemented JADE ontology must be implemented.

PSs need ontologies to understand the problems they are set to solve. To encode these problems, the TR also need to understand how to encode and decode actions, thus agents need a common understanding of actions. Ontologies are used to share understanding of domains. It is imperative that the TMST used as basis for the cooperation must be based on the same ontology that agents use to understand the domain. This ensures that the tasks, actions and PSMs described in the TMST are solvable by the agents. Implementing agents in the CoPS framework is done by extending the different agent classes. As the abstract CoPS agents contains a set of abstract methods, these methods must be implemented. Thus, implementing a CoPS agent means to:

- Extend the agent class corresponding to the agent role the agent is supposed to play.
- Implement the abstract method of the extended class.

JADE ontologies should be implemented as explained by Caire in [5]. A TMST must have a task as the root node, and actions as leaf nodes to be a valid TMST. All of the implemented nodes of the TMST must have unique names. The input, output and assumptions must be encoded using the implemented JADE ontology.

7.2 The Checkup Example Application

In our 2005-project [33], a complex scenario, involving a checkup at a health center was modeled, based on the CoPS framework. A small part of the model was implemented using the CoPS framework prototype. In the work with this master thesis, we have implemented a greater part of of the modeled checkup scenario.

This section describes the implemented part of the checkup scenario, and shows how it is realized by applying the CoPS API. First, an ontology was implemented. Second, the TMST depicted by figure 7.1 was implemented. The implemented TMST decomposes the problem of doing a checkup on a patient, and it has the initial task *Do checkup*. And finally, the different agents were implemented.

7.2.1 Implementation of the Checkup Ontology

The ontology serves as a vocabulary for representing knowledge in the system, describing the domain. It is known and used by the agents when they represent their services and when they interpret the messages they receive. In addition it is used in the implementation of the TMST. The TMST consists of several frames representing the different nodes of the tree. These frames have slots for input, output and goals that will need to be understood by the agents. As the agents know about the ontology, and these slots are defined using it, the agents can achieve a correct interpretation of the slot - information.

The implemented ontology is illustrated in figure 7.2. How JADE ontologies should be implemented, is described in [5]. JADE ontologies distinguish between *concepts*, *predicates* and *agent actions*. Concepts represent entities that exist in the world and may have a complex structure. Predicates are expressions that say

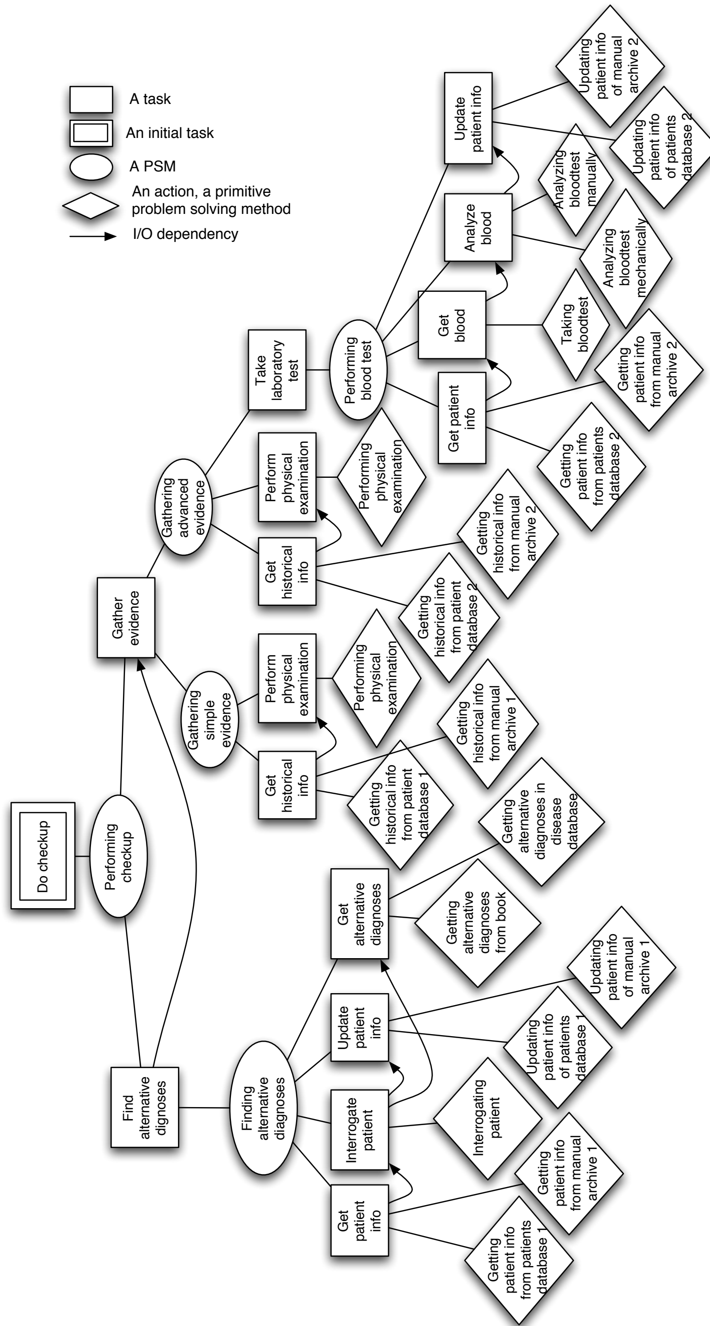

Figure 7.1: The implemented part of the Checkup TMST. The initial task *Do checkup* is decomposed into a hierarchy of PSMs, actions, and tasks. The Checkup TMST has a total of 20 different actions.

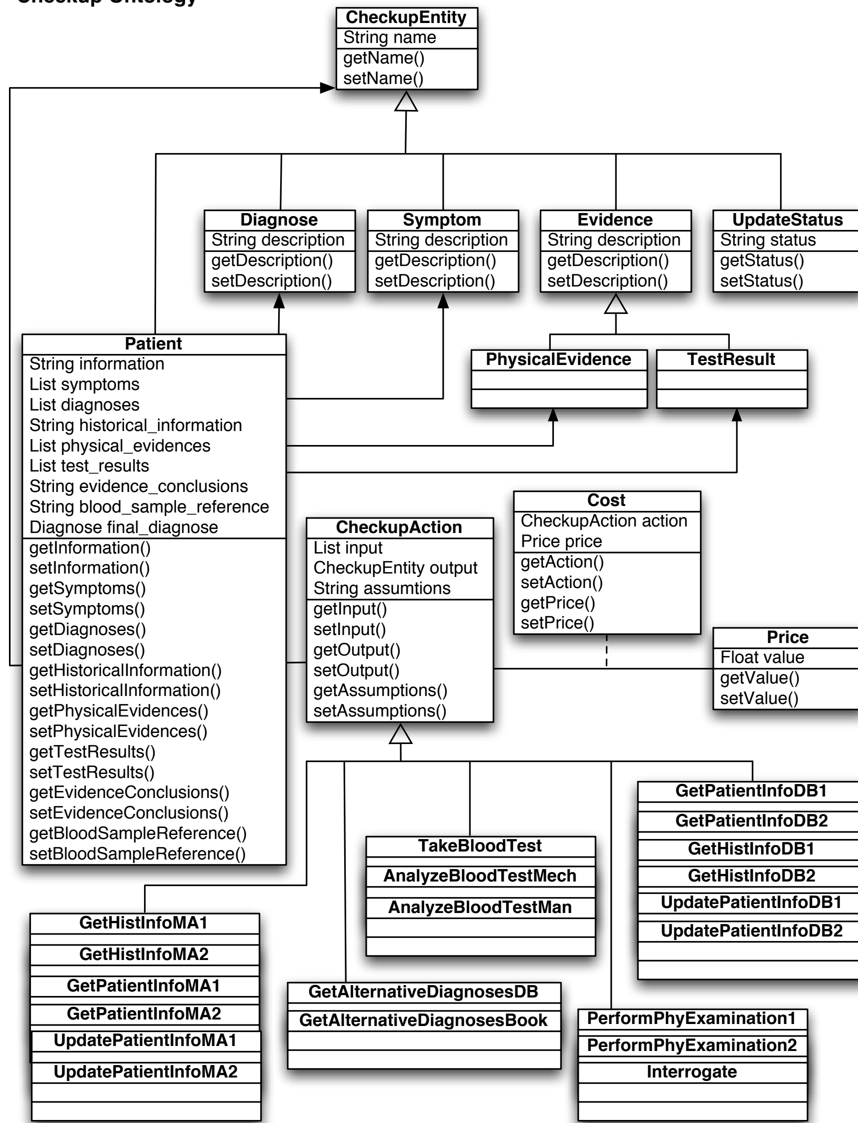

Figure 7.2: The class diagram of the checkup ontology. The ontology describes concepts used by agents when solving a checkup problem.

something about the world and thus may be either true or false. Agent actions are a special type of concepts that indicate actions that may be performed by agents.

In the *Checkup Ontology*, illustrated in figure 7.2 the *concepts* are the *CheckupEntity*, its subclasses and *Price*. The subclasses of the *CheckupEntity* are *Patient*, *Diagnose*, *Symptom*, *UdateStatus* and *Evidence*. The *Patient* concept is used to represent the patient that are in for the checkup. It has a list of properties which are updated and used by the agents while they solve the *Do Checkup* task on a specific patient. Some of the *Patient* properties are represented using the *CheckupEntity* subclasses. The *UpdateStatus* - concept is used to let the agents know if an update process, i.e. update of patient information, succeeded.

The *agent actions* in the ontology are subclasses of *CheckupAction*. Each of the 20 agent actions correspond to one of the action - nodes of the TMST, depicted in figure 7.1. *ChekupAction* specifies a set of attributes or properties: input, output and assumptions. Input may be a list of *CheckupEntity*, output must be one *CheckupEntity*, and the assumptions must be stored in a string. Assumptions are not used in this implementation though.

The only predicate part of the ontology is *Cost*. *Cost* ties a *Price* to a *CheckupAction*. Because *Cost* is a predicate, it is possible for a TR to ask the PSs about what the cost (*Cost*) of performing an action (*CheckupAction*) is. The agent answers with specifying the price (*Price*). Since JADE supports encoding and decoding objects to and from the textual FIPA-SL language, as long as the domain knowledge has been implemented as a JADE ontology, the agents are able to interpret the messages using the ontology. A query for what the price of getting alternative diagnoses from a database is encoded in FIPA-SL as follows:

```
((=
    (iota ?X
      (Cost
        (GetAlternativeDiagnosesDB
          :output
            (Patient
              :name paul
              :diagnoses
                (Diagnose
                  :description diagnose1)
                (Diagnose
                  :description diagnose2)) ?X))
    (Price
      :value 2.940009593963623
      :valuta GBP)))
```

The TR asks the receiver of this message whether there exists a price - X for getting alternative diagnoses from a database, where the output will be an instance of *Patient*. The patient has the name Paul and two alternative diagnoses. The *Cost* predicate has the form *Cost(CheckupAction, Price)*. The *CheckupAction* is *GetAlternativeDiagnosesDB*, the Price is ?X, meaning it is unknown. The TR therefor expects to get the value of X in return.

7.2.2 Implementation of the Checkup TMST

The Checkup TMST is implemented using the CoPS framework that again is based on jCreek. The different elements of the Checkup TMST, being tasks, actions and PSMs are all frames, represented by classes in the CoPS framework. The slots of these frames are also represented by classes in the CoPS framework. These classes are used for building the Checkup TMST.

The Checkup TMST is illustrated in figure 7.1. The implementation of the TMST, is done in the *CheckUpTMST* class. The different tasks, PSMs and actions are implemented by instantiating the corresponding classes in the CoPS framework, and they are all associated with an instance of the *TMST* class. The slots of the different frames, or the attributes of the nodes in the Checkup TMST are also defined here. The input, output, goal and (action) type slots are coded using the Checkup Ontology, because the information has to be understood by the TR and the PSs.

The initial task *Do Checkup* has the following values for the name, input, output, goal and problem solving method slots:

- name: *doCheckup*
- input: *(Patient :name x)*
- output: *(Patient :name x :finalDiagnose y)*
- goal: *(Patient :name x :finalDiagnose y")*
- problem solving method (PSM): *performingCheckup*

The input of the initial task is the patient's name represented by the Checkup Ontology's *Patient* concept, and the goal of this task is to associate a final diagnose to that patient. The output of the task is the same as the goal. To achieve this task, the PSM *performingCheckup* is used. This PSM has these values for the name, input, output and task slots:

- name: *performingCheckup*
- input: *(Patient :name x)*
- output: *(Patient :name x :finalDiagnose y)*
- tasks: *findAlternativeDiagnoses, gatherEvidence*

The *performingCheckup* PSM produces its output from composing the outputs from the tasks *findAlternativeDiagnoses* and *gatherEvidence*. The *findAlternativeDiagnoses* task has these values for the name, input, output, goal and problem solving method slots:

- name: *findAlternativeDiagnoses*
- input: *(Patient :name x)*
- output: *(Patient :name x :diagnoses (Diagnose :description y))*
- goal: *(Patient :name x :diagnoses (Diagnose :description y))*
- problem solving method (PSM): *findingAlternativeDiagnoses*

To achieve this task the PSM *findingAlternativeDiagnoses* is used. The other subtask of the PSM *performingCheckup* - *gatherEvidence* has these values for the name, input, output, goal and problem solving method slots:

- name: *gatherEvidence*
- input: *(Patient :name x :diagnoses (Diagnose :description y))*
- output: *(Patient :name x :diagnoses (Diagnose :description y) :evidence_conclusions z)*
- goal: *(Patient :name x :diagnoses (Diagnose :description y) :evidence_conclusions z)*
- problem solving method (PSM): *gatheringSimpleEvidence*, *gatheringAdvancedEvidence*

To achieve this task one of the PSMs *gatheringSimpleEvidence* and *gatheringAdvancedEvidence* may be used. All of the other tasks and PSMs part of the Checkup TMST, depicted in figure 7.1 are defined in similar ways. When defining nodes in the Checkup TMST it is important that the slot - values that relate to the Checkup Ontology is correctly stated, or else the agents using the information in the Checkup TMST will not be able to interpret the content of the slots. In addition, one has to be aware of the restrictions put on the format of a TMST by the TEAM SPACE architecture. These restrictions were:

- The output of a task should have the same definition as the output of its subnodes (actions or PSMs).
- The input of a task should have the same definition as the input of its subnodes (actions or PSMs).
- The input of a PSM should have the same definition as at least one of the inputs of its subtasks.

Actions in the Checkup TMST are also defined by slots with values represented on the form of the Checkup Ontology. The action *gettingPatientInfoDB1* has these values for the name, input, output and type slots:

- name: *gettingPatientInfoDB1*
- input: *(Patient :name x)*
- output: *(Patient :name x :information y)*
- type: *GetPatientInfoDB1*

The string defining this action-type is the name of a *CheckupAction* subclass in our Checkup Ontology. Each of the action nodes in the Checkup TMST refer to one of the *CheckupAction* subclasses in the Checkup Ontology like this. PSs define their capabilities also using the subclasses of *CheckupAction* in the Checkup Ontology. And when a PS has the capability *GetPatientInfoDB1*, it means that it is able to perform the action *gettingPatientInfoDB1*. The PS will only perform the action if it knows the input value *(Patient :name x)*, where x is substituted with a real name, say Paul. After performing the action an output

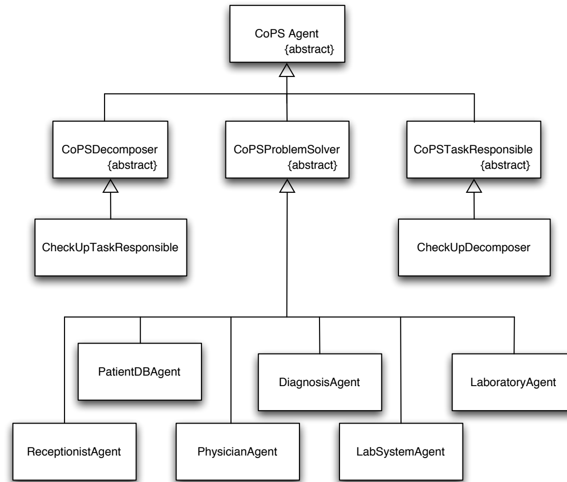

Figure 7.3: Class diagram for the agents implemented to solve the *Do checkup* task.

on the form *(Patient :name x :information y)* is produced: *(Patient :name Paul :information "Paul had a check last month where...")*.

The TMST is built by executing the *CheckUpTMST* class. During this execution, dependencies of the subtasks belonging to each PSM in the Checkup TMST are analyzed. The result of this analysis is stored in a slot of the PSM as control information. As can be seen in figure 7.1, the subtasks of the PSM described above *performingCheckup* has an I/O dependency. During the dependency analysis, the input and output values of subtasks of a PSM are compared. If the input of one task matches the output of another task, this is interpreted as an I/O dependency. *findAlternativeDiagnoses* has the output value *(Patient :name x :diagnoses (Diagnose :description y))* which is exactly the same as the input value of *gatherEvidence*. The I/O dependency between these tasks is stored in the PSM *performingCheckup* in this format:

<gatherEvidenceInput>(gatherEvidence) : -
<findAlternativeDiagnosesOutput>(findAlternativeDiagnoses)

The control information says that the task *findAlternativeDiagnoses* must be performed before the task *gatherEvidence*, as the input of *gatherEvidence* is the same as the output of *findAlternativeDiagnoses*.

7.2.3 Implementation of the Checkup Agents

Eight agents were implemented to solve the *Do checkup* (*doCheckup*) task. These are *CheckUpTaskResponsible*, *CheckUpDecomposer*, *ReceptionistAgent*, *PatientDBAgent*, *PhysicianAgent*, *DiagnosisAgent*, *LabSystemAgent* and *LaboratoryAgent*. The class diagram illustrating the agent hierarchy is pictured in figure 7.3. The *CheckUpTaskResponsible* is a subclass of *CoPSTaskResponsible*,

the *CheckUpDecomposer* is a subclass of *CoPSDecomposer*, and the six other implemented agents are subclasses of *CoPSProblemSolver*.

The Task Responsible (TR) - *CheckUpTaskResponsible*

Even though the *CheckUpTaskResponsible* is the one doing most of the work, this class contains little code. This is because the work done by the *CheckUpTaskResponsible* is more generic than the other agents. Most of the necessary functions are are implemented by the superclass *CoPSTaskResponsible*. The *CheckUpTaskResponsible* uses the Checkup Ontology to encode and decode messages associated with the *team formation* problem solving process step. And it uses the Checkup TMST to form the team of PSs and to initialize the TS structure.

During the problem solving process steps assisted by the TEAM SPACE; *solving of subtasks* and *integration of partial solutions*, the *CheckUpTaskResponsible* has to compose all of the PSMs that are tagged as part of the Checkup TMST solution. This is managed by implementing the abstract method *composePSM()* in *CoPSTaskResponsible*. By implementing that method, we make sure that the *CheckUpTaskResponsible* "knows" how to compose the partial results from the subtasks of a PSM. When a PSM's subtasks are achieved, a TS Agent informs *CheckUpTaskResponsible*, by sending a requests to compose the PSM. *CoPSTaskResponsible* handles the request in *composePSM()*, where it also creates the results that are returned in an inform message. The request message content should be encoded in FIPA-SL by using a CoPS Ontology. Since the CoPS ontology is still not implemented, the content is a text-string on this format:

<PSM-name>-<partial-result-1><partial-result-2>...<partial-result-3>

The content of the answer/inform message is formed as a text-string on this format:

<PSM-name>-<PSM-output>

If the *CheckUpTaskResponsible* was requested to compose the PSM *performingCheckup* described above, the request message from the TS Agent could look like this:

performingCheckup -
(Patient :name Paul :diagnoses (Diagnose :description "A diagnose for Paul"))
(Patient :name Paul :diagnoses (Diagnose :description "A diagnose for Paul")
:evidence_conclusions "Conclusions from the evidences")

Partial results in the message follow the definition of the output slots in the PSM *performingCheckup*'s subtasks. The inform message returned to a TS Agent could look like this:

performingCheckup -
(Patient :name x :finalDiagnose y)

The output value in the message follows the definition of the PSM *performingCheckup* output. Thus, the strings are still composed of concepts from our Checkup Ontology.

The Decomposer (DEC) - *CheckUpDecomposer*

The *CheckUpDecomposer* does not make the decomposition from the ground up. In the current version of the CoPS framework, it is only a mere container of the Checkup TMST. When the *CheckUpTMST* class is executed, the Checkup

```
((=
    (iota ?X
      (Cost
        (GetAlternativeDiagnosesDB
          :output
            (Patient
              :name paul
              :diagnoses
                (Diagnose
                  :description diagnose1)
                (Diagnose
                  :description diagnose2)) ?X))
    (Price
      :value 2.940009593963623
      :valuta GBP)))
```

Figure 7.4: A proposal message encoded in FIPA-SL, using the Checkup Ontology.

TMST definition is stored in a file. Every time *CheckUpDecomposer* receives a request message, that file is used to instantiate an object of the *TMST* class. *TMST* is used to manipulate and extract information from the Checkup TMST. *CheckUpDecomposer* adds a unique *teamID* to *TMST* before it is serialized and sent to the agent who requested it.

The Problem Solvers (PSs)

All of the implemented PSs extends *CoPSProblemSolver* and implement its abstract methods. In *setServiceDescription()*, the different PSs use the Checkup Ontology to describe their services, that they register with the Matchmaker. Services of the problem solving agents have two properties: The action-type, defining which action it can perform, and the output, defining the results of the action. The action-type is defined by one of the *CheckupAction* subclasses in the Checkup Ontology. The output is a string defining a *CheckupEntity*. The different implemented PSs and their services (action-types) are:

- *ReceptionistAgent*: *GetPatientInfoMA1, GetPatientInfoMA2, UpdatePatientInfoMA1, UpdatePatientInfoMA2, GetHistInfoMA1, GetHistInfoMA2*
- *PatientDBAgent*: *GetPatientInfoDB1, GetPatientInfoDB2, UpdatePatientInfoDB1, UpdatePatientInfoDB2, GetHistInfoDB1, GetHistInfoDB2*
- *PhysicianAgent*: *Interrogate, GetAlternativeDiagnosesBook, PerformPhyExamination1, PerformPhyExamination2*
- *DiagnosisAgent*: *GetAlternativeDiagnosesDB*
- *LabSystemAgent*: *AnalyzeBloodtestMech*
- *LaboratoryAgent*: *TakeBloodtest, AnalyzeBloodtestMan*

In *makeProposal()* the Checkup Ontology is used for decoding the call-for-proposal messages from *CheckUpTaskResponsible*. When a PS receives a call-for-proposal message, the method is used to consider if the agent should make a proposal or not. This is done by checking if one of its capabilities matches the requested action. If a PS decides to return a proposal message, the price is computed at random. The proposal that may be the response to the call-for-proposal message described in subsection 7.2.1, is encoded in FIPA-SL like described by figure 7.4. As a PS finds out that it has the capability of performing the action with the action-type *GetAlternativeDiagnosesDB* with the right output, it generates a price for performing the action. The price is decoded in the message as a Checkup Ontology *Price* concept.

A PS handles requests for executing actions received from a TS Agent, by implementing the abstract method *performAction()*. This method works in the same way as *composePSM()* in *CheckupTaskResponsible*. The content of the request message to be handled has the format:

<action-type>-<action-input>

And the content of the response message to this request has the format:

<action-type>-<action-output>

If a PS agent was requested to execute the action *gettingPatientInfoDB1*, the content of the request message could be:

GetPatientInfoDB1-
(Patient :name Paul)

And the content of the response message could be:

GetPatientInfoDB1-
(Patient :name Paul :information "Paul had a check...")

When the *performAction()* method is called, the PS agent has already agreed to solve the *GetPatientInfoDB1* message during the *team formation* process step.

7.3 Test-Run of the Checkup Example Application

How to run the implemented checkup example application, is described in appendix B. In the first subsection, the described test-run involves solving a single problem, and in the second subsection the described test-run involves solving two problems in parallel.

During our test-runs, where the agents solve the problem decomposed by figure 7.1, the process of the problem solving process is written to a txt-file with the same name as the instantiated *CheckupTaskResponsible*. In addition, we can follow the flow of messages between the agents using the JADE GUI (graphical user interface). The txt-file and the GUI are used to present run-time descriptions of our example application. Our description follows the different steps of the problem solving process as described in chapter 3.

7.3.1 Solving a single problem

Here we describe the test-run of the example code solving a single problem. We instantiate or start one of each of the implemented agents in addition to a *TeamSpaceAgent* and a *DefaultAgent*, by naming them. The default agent is the one sending a request for the *CheckupTaskResponsible* to solve a task. If

a solution is found or a failure occurs during the problem solving process, the *CheckupTaskResponsible* notifies the default agent by sending an inform message or a failure message. The agents run in the JADE Environment. Names of the different instances of the agent classes are as follows:

ReceptionistAgent - RE
PatientDBAgent - PDB
PhysicianAgent - PHY
DiagnosisAgent - DI
LabSystemAgent - LSA
LaboratoryAgent - LA
CheckupTaskResponsible - TR
CheckupDecomposer - DEC
TeamSpaceAgent - TSAgent
DefaultAgent - DEF

Remember that the PSs have registered their capabilities in form of action-types and outputs, as described in subsection 7.2.3.

The problem solving is started by *DEF* which sends a request message for *TR*. The terminal window from where the code is run has this output, which describes the problem being solved and the final outcome of the problem solving:

```
TR received a task solving request from DEF, output from the different
steps of the problem solvin process are written to TR.txt stored where this program is run from.
TR received the task: (Task :name doCheckup :input (Patient :name paul))
TR@Kari.local:1099/JADE returns the solution for the requester of the problem, DEF
  Content of the solution-message: "(Patient :name paul :finalDiagnose diagnose)"
```

The intervening steps where the problem is solved in accordance to the problem solving process steps, described in chapter 3, are logged to the file TR.txt. These steps are described next. The txt-file that was generated during this specific test-run is enclosed in the zip-file, following this thesis, as described in appendix A.

Problem Analyzis

TR reviews the task *(Task :name doCheckup :input (Patient :name paul))*, and finds that it cannot solve it alone.

Team Formation

TR needs a decomposition of the task (*TMST*). And sends a **request** for *DEC* with the content: *(Task :name doCheckup :input (Patient :name paul))*. *DEC* understands the task and sends an **agree** message for *TR*. Then *DEC* creates a *TMST* object, sets *TMST.teamID = 1* and returns it for *TR* in an **inform** message. *TR* extracts the *TMST* from the message, sets *TMST.initialInput = (Patient :name paul))*, and saves the *TMST* object in an instance variable.

Next, *TR* requests the *Matchmaker* for agents capable of performing the actions in the *TMST*. One **request** is sent per action and one **inform** message

for each of the requests are received in return. Action (action-type) - agent tuples that are a result from this conversation is illustrated in figure 7.5.

```
gettingPatientInfoDB1 (GetPatientInfoDB1) - PDB
gettingPatientInfoMA1 (GetPatientInfoMA1) - RE
interrogatingPatient (Interrogate) - PHY
updatingPatientInfoDB1 (UpdatePatientInfoDB1) - PDB
updatingPatientInfoMA1 (UpdatePatientInfoMA1) - RE
gettingAlternativeDiagnosesBook (GetAlternativeDiagnosesBook) - PHY
gettingAlternativeDiagnosesDiseaseDB (GetAlternativeDiagnosesDB) - DI
gettingHistInfoDB1 (GetHistInfoDB1) - PDB
gettingHistInfoMA1 (GetHistInfoMA1) - RE
performingPhysicalExamination1 (PerformPhyExamination1) - PHY
gettingHistInfoDB2 (GetHistInfoDB2) - PDB
gettingHistInfoMA2 (GetHistInfoMA2) - RE
performingPhysicalExamination2 (PerformPhyExamination2) - PHY
gettingPatientInfoDB2 (GetPatientInfoDB2) - PDB
gettingPatientInfoMA2 (GetPatientInfoMA2) - RE
takingBloodtest (TakeBloodtest) - LA
analyzingBloodtestMech (AnalyzeBloodtestMech) - LSA
analyzingBloodtestMan (AnalyzeBloodtestMan) - LA
updatingPatientInfoDB2 (UpdatePatientInfoDB2) - PDB
updatingPatientInfoMA2 (UpdatePatientInfoMA2) - RE
```

Figure 7.5: Action (action - type) - agent tuples that are a returned by the *Matchmaker* (DF).

Now, the FIPA Contract Net Protocol is initialized by the *TR* and one **call-for-proposal** message is made and sent for each of the action - agent tuples in figure 7.5. The call-for-proposal messages look like the one described in section 7.2.1. The agents that are capable of performing several actions, receives one call-for-proposal message for each of these actions. The PSs respond with **one proposal** message for each of the received call-for-proposal messages, where they offer a price for performing their actions. The propose messages look like the one described in section 7.2.3.

All the proposals are handled by *TR*. Names of the agents sending proposals are used to create *Executors* (one of the TMST main classes - described in chapter 6). *Executors* are mapped to *Actions* in the *TMST*. Each *Executor* - *Action* relation has a *Cost* - value. The information needed to do this is found in the proposals. The result from this process, based on the *TR's* received proposal in this test-run is illustrated in figure 7.6. The first line in figure 7.6 says that the agent *DI* offers to perform the action *gettingAlternativeDiagnosesDiseaseDB* for the cost of approximately *10.5*. The result will be different each time the code is run, since the PSs generate their price for performing an action randomly.

Next, the solution space is activated in the *TMST*. All *Actions* connected to an *Executor* are activated, all *Tasks* connected to an activated *Action* or *PSM* are activated. And, all *PSMs* having all of their sub-*Tasks* activated are activated. If the initial *Task* gets activated, the problem has at least one solution. In this example all of the nodes in the *TMST* are activated, including the initial task *doCheckup*.

After the solution space is activated, *TR* tries to find the cheapest solution according to the *Executor* - *Action* costs. One starts at the bottom level (*Actions*) and propagates the costs upwards in the *TMST*. For all of the *Tasks* one

```
Executor: DI - Action: gettingAlternativeDiagnosesDiseaseDB - Cost: 10.5400915
Executor: LSA - Action: analyzingBloodtestMech - Cost: 0.76991844
Executor: LA - Action: analyzingBloodtestMan - Cost: 8.804422
Executor: LA - Action: takingBloodtest - Cost: 4.135359
Executor: PHY - Action: performingPhysicalExamination2 - Cost: 9.996419
Executor: PHY - Action: performingPhysicalExamination1 - Cost: 6.4883585
Executor: PHY - Action: gettingAlternativeDiagnosesBook - Cost: 11.443243
Executor: PDB - Action: updatingPatientInfoDB2 - Cost: 9.371593
Executor: PHY - Action: interrogatingPatient - Cost: 1.3894434
Executor: PDB - Action: gettingPatientInfoDB2 - Cost: 6.4522524
Executor: RE - Action: updatingPatientInfoMA2 - Cost: 8.703579
Executor: PDB - Action: gettingHistInfoDB2 - Cost: 5.038865
Executor: RE - Action: gettingPatientInfoMA2 - Cost: 3.9942203
Executor: RE - Action: gettingHistInfoMA2 - Cost: 0.88038605
Executor: RE - Action: gettingHistInfoMA1 - Cost: 4.392433
Executor: RE - Action: updatingPatientInfoMA1 - Cost: 7.2296653
Executor: PDB - Action: gettingHistInfoDB1 - Cost: 5.1099377
Executor: RE - Action: gettingPatientInfoMA1 - Cost: 5.383788
Executor: PDB - Action: updatingPatientInfoDB1 - Cost: 6.2010794
Executor: PDB - Action: gettingPatientInfoDB1 - Cost: 2.5902364
```

Figure 7.6: Result from the *TR's* handling of proposals. *Executors* (*TMST* node) are named after PSs, and related to a *TMST Action* and a *Cost*.

```
Task: doCheckup  Cost: 31.601643
PSM: performingCheckup Cost: 31.601643
Task: findAlternativeDiagnoses  Cost: 20.720852
PSM: findingAlternativeDiagnoses Cost: 20.720852
Task: getPatientInfo1  Cost: 2.5902364
Action: gettingPatientInfoDB1 Cost: 2.5902364
Task: interrogatePatient  Cost: 1.3894434
Action: interrogatingPatient Cost: 1.3894434
Task: updatePatientInfo1  Cost: 6.2010794
Action: updatingPatientInfoDB1 Cost: 6.2010794
Task: getAlternativeDiagnoses  Cost: 10.5400915
Action: gettingAlternativeDiagnosesDiseaseDB Cost: 10.5400915
Task: gatherEvidence  Cost: 10.880792
PSM: gatheringSimpleEvidence Cost: 10.880792
Task: getHistInfo1  Cost: 4.392433
Action: gettingHistInfoMA1 Cost: 4.392433
Task: performPhysicalExamination1  Cost: 6.4883585
Action: performingPhysicalExamination1 Cost: 6.4883585
TR did find the cheepest solution
```

Figure 7.7: Result from the solution generation. The figure lists the different nodes part of the solution and their final costs. The cost of the initial *Task* is approximately *31.6*.

```
These agents receive invitations (accept-proposals), from TR, to join the team:
  Agent: PDB  Invitation: (action (:name GetPatientInfoDB1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name Interrogate) (:teamID 1))
  Agent: PDB  Invitation: (action (:name UpdatePatientInfoDB1) (:teamID 1))
  Agent: DI  Invitation: (action (:name GetAlternativeDiagnosesDB) (:teamID 1))
  Agent: RE  Invitation: (action (:name GetHistInfoMA1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name PerformPhyExamination1) (:teamID 1))

These agents receives reject-proposals, saying they don't get to join the team:
  Agent: LSA  Action in proposal: AnalyzeBloodtestMech
  Agent: LA  Action in proposal: AnalyzeBloodtestMan
  Agent: LA  Action in proposal: TakeBloodtest
  Agent: PHY  Action in proposal: PerformPhyExamination2
  Agent: PHY  Action in proposal: GetAlternativeDiagnosesBook
  Agent: PDB  Action in proposal: UpdatePatientInfoDB2
  Agent: PDB  Action in proposal: GetPatientInfoDB2
  Agent: RE  Action in proposal: UpdatePatientInfoMA2
  Agent: PDB  Action in proposal: GetHistInfoDB2
  Agent: RE  Action in proposal: GetPatientInfoMA2
  Agent: RE  Action in proposal: GetHistInfoMA2
  Agent: RE  Action in proposal: UpdatePatientInfoMA1
  Agent: PDB  Action in proposal: GetHistInfoDB1
  Agent: RE  Action in proposal: GetPatientInfoMA1
```

Figure 7.8: A list of the agents receiving accept-proposal and refuse-proposal messages, after the solution of the *TMST* is generated.

always choose the cheapest alternative of solving it (choose one of the sub-*PSMs* or sub-*Actions*). All chosen nodes are tagged as part of the chosen solution. The nodes part of the solution and their costs are illustrated in figure 7.7. The total cost of the chosen solution is approximately *31.6*.

When the solution is generated, *TR* makes invitations (**accept-proposals**) to join the team for the agents that made a prosal and is mapped (through an *Executor*) to the *Actions* that are part of the chosen solution. The other agents that made a proposal receive a **refuse-proposal** message. An agent gets one inivtation(accept-proposal) or refuse-proposal for each of the prosals it has made. Who gets to join the team and who are refused is illustrated in figure 7.8.

TR receives **inform** messages from all of the agents that received an invitation (accept-proposal). And *TR* has successfully formed a team.

Solving of Subtasks and Integration of Partial Results

The problem solving process steps; *solving of subtasks* and *integration of partial results*, are performed using the TEAM SPACE. These steps are described using a message sequence diagram copied from the JADE GUI at run-time. The diagram is depicted in figure 7.9. The different messages are explained by the following numbered list. The numbers correspond to the numbers aligned with the messages in the figure.

1. *TR* requests the *Matchmaker* (DF), for an agent offering the service *TEAM_SPACE_COORDINATOR*.

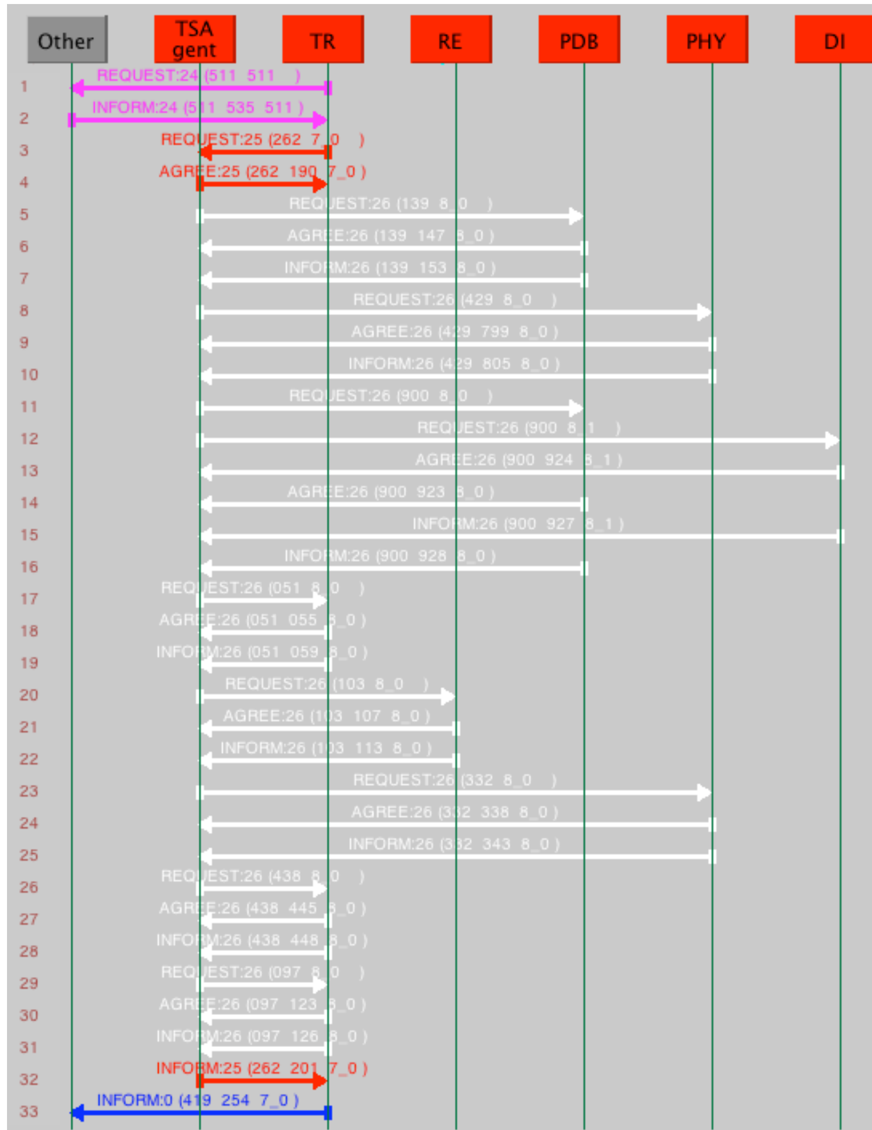

Figure 7.9: Agent interactions in the problem solving process steps: *solving of subtasks* and *integration of partial results*.

```
##############  CURRENT STATE OF THE TEAM SPACE ################

######################  Achieved goals  ########################
    No achieved goals yet...

#####################  Executed actions  #######################
    No executed actions yet...

######################  Composed PSMs  #########################
    No composed PSMs yet...

################## Actions ready to execute ####################
    Action: gettingPatientInfoDB1
    Input: "(Patient :name paul)"
    Executor: PDB@Kari.local:1099/JADE

##################  PSMs ready to compose  #####################
    No PSMs to compose in the current state...
```

Figure 7.10: The first problem solving state, generated after executing the rules of *RuleBase* for the first time.

2. *Matchmaker* returns the address of *TSAgent*

3. *TR* requests the *TSAgent* to initialize a *TS Structure*. Attached to this request is the *TMST*, which now contains information about the chosen solution.

4. *TSAgent* sends an agree message to tell the *TR* that initialization of the *TS Structure* went well. The *TS Structure* is represented by an object of the *TeamSpace* class. Templates have been defined and rules have been generated like described in chapter 4. Defined rules and templates are logged to TR.txt, and it involves to much information to include here. The *TeamSpace* object attribute *teamID* is set to 1. And an initial fact has been added to the *ResultLibrary*: *(taskinput (name doCheckup_input1) (value "(Patient :name paul)"))*.

   Now the *TSAgent* may initiate the solving of subtasks (execution of actions), and integration of partial results (composition of PSMs). First, the *TSAgent* manipulates the *TeamSpace* object so that the rules are executed, and thus the problem solving state is updated. The problem solving state at this point is illustrated in figure 7.10.

5. *TSAgent* requests *PDB* to execute *gettingPatientInfoDB1* with the input *(Patient :name paul)*. The message has the formate as described in section 7.2.3. That is also the case for the other messages exchanged between *TSAgent* and PSs, and between *TSAgent* and *TR*.

6. *PDB* sends an agree message. It agrees to execute the action.

7. *PDB* sends an inform message containing the output: *(Patient :name paul :information paul_information)*.

```
##############  CURRENT STATE OF THE TEAM SPACE ################

######################  Achieved goals  ########################
    Goal: getPatientInfo1_goal
    Output: "(Patient :name paul :information paul_information)"

#####################  Executed actions  #######################
    Action: gettingPatientInfoDB1
    Output: (Patient :name paul :information paul_information)

######################  Composed PSMs  #########################
    No composed PSMs yet...

################## Actions ready to execute ####################
    Action: interrogatingPatient
    Input: "(Patient :name paul :information paul_information)"
    Executor: PHY@Kari.local:1099/JADE

##################  PSMs ready to compose  #####################
    No PSMs to compose in the current state...
```

Figure 7.11: The second problem solving state, generated after executing the rules of *RuleBase* for the second time.

Each time the *TSAgent* receives solutions to executed actions or composed PSMs, it makes sure that the problem solving state kept by the *TeamSpace* object is updated. This involves adding new information (results) in *ResultLibrary*, and executing the rules in *RuleBase*. Rules that fire also add new information in *ResultLibrary*. The problem solving state at this point is illustrated in figure 7.11.

8. *TSAgent* requests *PHY* to execute *interrogatingPatient* with the input *(Patient :name paul :information paul_information)*.

9. *PHY* sends an agree message. It agrees to execute the action.

10. *PHY* sends an inform message containing the output: *(Patient :name paul :information paul_information :symptoms (Symptom :description symptom1))*.

    The problem solving state is updated (description of the whole problem solving state at this point can be found in the file TR.txt, this is also the case for the proceeding evolution of the problem solving states). Actions ready to execute: *updatingPatientInfoDB1*, *gettingAlternativeDiagnoses-DiseaseDB*.

11. *TSAgent* requests *PDB* to execute *updatingPatientInfoDB1* with the input *(Patient :name paul :information paul_information :symptoms (Symptom :description symptom1))*.

12. *TSAgent* requests *DI* to execute *gettingAlternativeDiagnosesDiseaseDB*

with the input *(Patient :name paul :information paul_information :symptoms (Symptom :description symptom1))*.

13. *DI* sends an agree message. It agrees to execute the action.
14. *PDB* sends an agree message. It agrees to execute the action.
15. *DI* sends an inform message containing the output: *(Patient :name paul :diagnoses (Diagnose :description diagnose1))*.
16. *PDB* sends an inform message containing the output: *(UpdateStatus :status ok)*.

    The problem solving state is updated. PSM ready to compose: *findingAlternativeDiagnoses*.

17. *TSAgent* requests *TR* to compose *findingAlternativeDiagnoses*, the partial results are: *(Patient :name paul :information paul_information), (Patient :name paul :information paul_information :symptoms (Symptom :description symptom1)), (UpdateStatus :status ok), (Patient :name paul :diagnoses (Diagnose :description diagnose1))*.
18. *TR* sends an agree message. It agrees to compose the PSM.
19. *TR* sends an inform message containing the output: *(Patient :name paul :diagnoses (Diagnose :description diagnose1))*.

    The problem solving state is updated. Action ready to execute: *gettingHistInfoMA1*.

20. *TSAgent* requests *RE* to execute *gettingHistInfoMA1* with the input *(Patient :name paul :diagnoses (Diagnose :description diagnose1))*.
21. *RE* sends an agree message. It agrees to execute the action.
22. *RE* sends an inform message containing the output: *(Patient :name paul :historical_information paul_historical_information :diagnoses (Diagnose :description diagnose1))*.

    The problem solving sate is updated. Action ready to execute: *performingPhysicalExamination1*.

23. *TSAgent* requests *PHY* to execute *performingPhysicalExamination1* with the input *(Patient :name paul :historical_information paul_historical_information :diagnoses (Diagnose :description diagnose1))*.
24. *PHY* sends an agree message. It agrees to execute the action.
25. *PHY* sends an inform message containing the output: *(Patient :name paul :historical_information paul_historical_information :diagnoses (Diagnose :description diagnose1) :physical_evidences (PhysicalEvidence :description evidence1))*.

    The problem solving state is updated. PSM ready to compose: *gatheringSimpleEvidence*.

26. *TSAgent* requests *TR* to compose *gatheringSimpleEvidence*, the partial results are: *(Patient :name paul :historical_information paul_historical_information :diagnoses (Diagnose :description diagnose1)), (Patient :name paul :historical_information paul_historical_information :diagnoses (Diagnose :description diagnose1) :physical_evidences (PhysicalEvidence :description evidence1)).*

27. *TR* sends an agree message. It agrees to compose the PSM.

28. *TR* sends an inform message containing the output: *(Patient :name paul :diagnoses (Diagnose :description diagnose1) :evidence_conclusions summary).*

    The problem solving state is updated. PSM ready to compose: *performingCheckup*.

29. *TSAgent* requests *TR* to compose *performingCheckup*, the partial results are: *(Patient :name paul :diagnoses (Diagnose :description diagnose1)), (Patient :name paul :diagnoses (Diagnose :description diagnose1) :evidence_conclusions summary).*

30. *TR* sends an agree message. It agrees to compose the PSM.

31. *TR* sends an inform message containing the output: *(Patient :name paul :finalDiagnose diagnose).*

    The problem solving state is updated. There are no actions ready to execute and no PSMs ready to compose. The problem solving state tells that the goal *doCheckup_goal* is achieved, and has the output *(Patient :name paul :finalDiagnose diagnose)*. That goal belongs to the initial task *doCheckup*, and this means that the initial problem is solved.

32. *TSAgent* returns the solution to the initial task *doCheckup*, *(Patient :name paul :finalDiagnose diagnose)*, for *TR* in an inform message. The inform message completes the conversation initiated by the third message in figure 7.9.

33. *TR* returns an inform message for *DEF* with the solution of the initial task *doCheckup*, *(Patient :name paul :finalDiagnose diagnose)*. This inform message completes the first conversation that was initiated, where *DEF* requested *TR* to solve the task *doCheckup*.

The last message described above completes the problem solving process. And the *TR* is now free to handle further requests for solving tasks.

7.3.2 Solving of two problems in parallel

Here we describe the test-run of the example code solving two problems in parallel. As described in chapter 4, the TEAM SPACE has two different ways of solving concurrent problem solving teams. If only one TS Agent is running in the environment, this agent may handle several teams by creating a TS structure for each of them. If there are several TS Agents running in the environment, these agents share handling the requests of initializing a TS structure from TRs.

There is always one TR and a group of PSs in a problem solving team. The PSs can join several teams at the same time. A TR is only part of one team at a time.

Test-run with One TS Agent

We instantiate/start one of each of the implemented agents, except there are two instances of *CheckUpTaskResponsible*. In addition a *TeamSpaceAgent* and a*DefaultAgent* are instantiated. Names of the different instances of the agent classes are as follows: Names of the different instances of the agent classes are as follows:

ReceptionistAgent - RE
PatientDBAgent - PDB
PhysicianAgent - PHY
DiagnosisAgent - DI
LabSystemAgent - LSA
LaboratoryAgent - LA
CheckupTaskResponsible - TR1
CheckupTaskResponsible - TR2
CheckupDecomposer - DEC
TeamSpaceAgent - TSAgent
DefaultAgent - DEF

The problem solving is started by the *DEF* which sends a request message for *TR1* and *TR2*. The request messages specify two different problems to be solved. The terminal window from where the code is run has this output, which describes the problems being solved and the final outcome:

```
TR2 received a task solving request from DEF, output from the different
steps of the problem solvin process are written to TR2.txt stored where this program is run from.
TR2 received the task: (Task :name doCheckup :input (Patient :name maria))

TR1 received a task solving request from DEF, output from the different
steps of the problem solvin process are written to TR1.txt stored where this program is run from.
TR1 received the task: (Task :name doCheckup :input (Patient :name paul))
TR2@dhcp-102-247.idi.ntnu.no:1099/JADE returns the solution for the requester of the problem, DEF
  Content of the solution-message: "(Patient :name maria :finalDiagnose diagnose)"
TR1@dhcp-102-247.idi.ntnu.no:1099/JADE returns the solution for the requester of the problem, DEF
  Content of the solution-message: "(Patient :name paul :finalDiagnose diagnose)"
```

The intervening steps where the problems are solved, follow the description given in subsection 7.3.1. The work related to *TR1* is written to the file TR1.txt, and the work related to *TR2* is written to the file TR2.txt. The txt-files from this specific test run are found in the enclosed zip-file, like described in appendix A. Figure 7.12 and 7.13 shows which agents get to join the teams of respectively *TR1* and *TR2*. These figures also show that *TR1* and *TR2* have achieved different solutions in their *TMST*s, and that an agent (like the *PBD* and *PHY*) can join several problem solving teams. For the team of *TR1 teamID = 2*, and for the team of *TR2 teamID = 1*.

Both *TR1* and *TR2* contacts *TSAgent* for initializing a *TeamSpace* where they can solve their problems. Segments from the files TR1.txt and TR2.txt illustrated in respectively figure 7.14 and 7.15 show that both *TR1* and *TR2*

```
These agents receive invitations (accept-proposals), from TR1, to join the team:
  Agent: PDB  Invitation: (action (:name GetPatientInfoDB1) (:teamID 2))
  Agent: PHY  Invitation: (action (:name Interrogate) (:teamID 2))
  Agent: PDB  Invitation: (action (:name UpdatePatientInfoDB1) (:teamID 2))
  Agent: DI  Invitation: (action (:name GetAlternativeDiagnosesDB) (:teamID 2))
  Agent: RE  Invitation: (action (:name GetHistInfoMA1) (:teamID 2))
  Agent: PHY  Invitation: (action (:name PerformPhyExamination1) (:teamID 2))
```

Figure 7.12: A list of the agents receiving accept-proposal messages from *TR1*, after the solution of the *TMST* is generated.

```
These agents receive invitations (accept-proposals), from TR2, to join the team:
  Agent: RE  Invitation: (action (:name GetPatientInfoMA1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name Interrogate) (:teamID 1))
  Agent: PDB  Invitation: (action (:name UpdatePatientInfoDB1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name GetAlternativeDiagnosesBook) (:teamID 1))
  Agent: RE  Invitation: (action (:name GetHistInfoMA1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name PerformPhyExamination1) (:teamID 1))
```

Figure 7.13: A list of the agents receiving accept-proposal messages from *TR2*, after the solution of the *TMST* is generated.

contacts *TSAgent* to solve two different problems. During the execution of actions and composition of PSMs, the *TSAgent* runs a parallel behaviour with two sets of sub-behaviours, where each of them handles one of the problems. The two initialized *TeamSpace* classes are separated by the *teamID* instance variable.

Test-run with Two TS Agents

We instantiate/start one of each of the implemented agents, except there are two instances of *CheckUpTaskResponsible*. In addition two *TeamSpaceAgent* instances and an instance of *DefaultAgent* are made. Names of the different instances of the agent classes are as follows:
ReceptionistAgent - RE

```
TR1 requests (a TEAM SPACE agent) TSAgent to initialize
the TEAM SPACE, by sending the TMST, now containing information about the chosen solution

###############################################################
    Initial task/problem: doCheckup
    Input: (Patient :name paul)
    TeamID: 2
###############################################################
```

Figure 7.14: A segment from TR1.txt showing that *TR1* requests *TSAgent* to initialize a TS structure, and specialize it for a task with a certain input and teamID.

```
TR2 requests (a TEAM SPACE agent) TSAgent to initialize
the TEAM SPACE, by sending the TMST, now containing information about the chosen solution

###############################################################
    Initial task/problem: doCheckup
    Input: (Patient :name maria)
    TeamID: 1
###############################################################
```

Figure 7.15: A segment from TR2.txt showing that *TR2* requests *TSAgent* to initialize a TS structure, and specialize it for a task with a certain input and teamID.

PatientDBAgent - PDB
PhysicianAgent - PHY
DiagnosisAgent - DI
LabSystemAgent - LSA
LaboratoryAgent - LA
CheckupTaskResponsible - TR1
CheckupTaskResponsible - TR2
CheckupDecomposer - DEC
TeamSpaceAgent - TSAgent1
TeamSpaceAgent - TSAgent2
DefaultAgent - DEF

The problem solving is again started by the *DEF* which sends a request message for *TR1* and *TR2*. The request messages specify the two different problems to be solved; the same problems as was used in the test-run with one TS Agent. The terminal from where the code is run has this output, which describes the problems being solved and the final outcome:

```
TR1 received a task solving request from DEF, output from the different
steps of the problem solvin process are written to TR1.txt stored where this program is run from.
TR1 received the task: (Task :name doCheckup :input (Patient :name paul))

TR2 received a task solving request from DEF, output from the different
steps of the problem solvin process are written to TR2.txt stored where this program is run from.
TR2 received the task: (Task :name doCheckup :input (Patient :name maria))
TR2@dhcp-102-247.idi.ntnu.no:1099/JADE returns the solution for the requester of the problem, DEF
  Content of the solution-message: "(Patient :name maria :finalDiagnose diagnose)"
TR1@dhcp-102-247.idi.ntnu.no:1099/JADE returns the solution for the requester of the problem, DEF
  Content of the solution-message: "(Patient :name paul :finalDiagnose diagnose)"
```

The intervening steps where the problems are solved, follow the description given in subsection 7.3.1. The work related to *TR1* is written to the file TR1.txt, and the work related to *TR2* is written to the file TR2.txt. The txt-files from this specific test run are found in the enclosed zip-file, like described in appendix A. Figure 7.16 and 7.17 shows which agents get to join the teams of respectively *TR1* and *TR2*. In the test-run with one TS Agent, all of the agents joining the two teams communicated with the same *TSAgent* during the problem solving process. In this example, the agents in the team of *TR1* communicate with

```
These agents receive invitations (accept-proposals), from TR1, to join the team:
  Agent: PDB  Invitation: (action (:name GetPatientInfoDB1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name Interrogate) (:teamID 1))
  Agent: RE  Invitation: (action (:name UpdatePatientInfoMA1) (:teamID 1))
  Agent: DI  Invitation: (action (:name GetAlternativeDiagnosesDB) (:teamID 1))
  Agent: PDB  Invitation: (action (:name GetHistInfoDB1) (:teamID 1))
  Agent: PHY  Invitation: (action (:name PerformPhyExamination1) (:teamID 1))
```

Figure 7.16: A list of the agents receiving accept-proposal messages from *TR1*, after the solution of the *TMST* is generated.

```
These agents receive invitations (accept-proposals), from TR2, to join the team:
  Agent: RE  Invitation: (action (:name GetPatientInfoMA1) (:teamID 2))
  Agent: PHY  Invitation: (action (:name Interrogate) (:teamID 2))
  Agent: PDB  Invitation: (action (:name UpdatePatientInfoDB1) (:teamID 2))
  Agent: PHY  Invitation: (action (:name GetAlternativeDiagnosesBook) (:teamID 2))
  Agent: PDB  Invitation: (action (:name GetHistInfoDB1) (:teamID 2))
  Agent: PHY  Invitation: (action (:name PerformPhyExamination1) (:teamID 2))
```

Figure 7.17: A list of the agents receiving accept-proposal messages from *TR2*, after the solution of the *TMST* is generated.

TSAgent1, and the agents in the team of *TR2* communicate with *TSAgent2*.

Segments from the files TR1.txt and TR2.txt illustrated in respectively figure 7.14 and 7.15, show that *TR1* contacts *TSAgent1* and *TR2* contacts *TSAgent2* to solve two different problems.

7.4 Test-Run Results

In this section we summarize the results from the test-run of our checkup example application. In addition to the test-run scenarios described in this chapter, we have used other scenarios to test the corrections and extensions to the CoPS framework prototype. The results are based on what is implemented. Corrections and extensions to the CoPS framework architecture that have been

```
TR1 requests (a TEAM SPACE agent) TSAgent1 to initialize
the TEAM SPACE, by sending the TMST, now containing information about the chosen solution
################################################################
    Initial task/problem: doCheckup
    Input: (Patient :name paul)
    TeamID: 1
################################################################
```

Figure 7.18: A segment from TR1.txt showing that *TR1* requests *TSAgent1* to initialize a TS structure, and specialize it for a task with a certain input and teamID.

```
TR2 requests (a TEAM SPACE agent) TSAgent2 to initialize
the TEAM SPACE, by sending the TMST, now containing information about the chosen solution
##############################################################
    Initial task/problem: doCheckup
    Input: (Patient :name maria)
    TeamID: 2
##############################################################
```

Figure 7.19: A segment from TR2.txt showing that *TR2* requests *TSAgent2* to initialize a TS structure, and specialize it for a task with a certain input and teamID.

mentioned earlier, but are not implemented, are proposed for future work in chapter 8.

7.4.1 Corrections

The corrections to the CoPS framework prototype were proposed in chapter 3 - subsection 3.5.1. Which of the corrections that have been implemented and details about the implementation were described in chapter 6. During the implementation, new faults were detected and corrected, these are also described in chapter 6. Results from implemented corrections that affect the code at run-time are:

- The messages sent between the agents are now corrected, and the CoPS agents behave as described in chapter 3. This can be verified by looking at the message sequence diagram, which can be set up in the JADE GUI at run-time (like the one in figure 7.9). Task Responsible (TR) can at any time conclude correctly about the state of the problem solving process, and it runs until it is being properly terminated. This has been tested by running examples where there are not enough agents to form a problem solving team, where an agent fails to perform its action, etc. No matter what, the TR informs the requester of the problem about the outcome in either an inform message (containing the result) or in a failure message (containing a failure specification).

- All of the conversation-protocols are complete, as all necessary messages are being sent. Meaning that our CoPS framework prototype is FIPA compliant. All conversations conform to either the *FIPA Request Protocol* or the *FIPA Contract Net Protocol*, depicted in chapter 3 - figure 3.5 and 3.6. This can be verified by looking at the message sequence diagram, which can be set up in the JADE GUI at run-time

- The generation of the solution space, work as it is supposed to. This can be verified by looking in the TR.txt files generated during a test-run. A conceptual description of the solution space generation is provided in chapter 3 - section 3.2.

- The generation of the solution, work as it is supposed to. This can be verified by looking in the TR.txt files generated during a test-run, or at

the test-run scenarios described in this chapter. A conceptual description of the solution generation is also provided in chapter 3 - section 3.2.

- A Problem Solver (PS) can have several capabilities. It can also attend to multiple conversations at the same time. A PS handles several conversations with the TR during the team formation. A PS can join several problem solving teams simultaneously. This can be verified by looking at the previous sections of this chapter.

7.4.2 Extensions Pre-Existing CoPS components

Extensions to the pre-existing CoPS framework components were proposed in chapter 3 - subsection 3.5.2 and modeled in chapter 4 - section 4.4. Which extensions are implemented and their implementation details are described in chapter 6. Results from implemented extensions are:

- The Task Responsible (TR) is able to use the TEAM SPACE. TR communicates with the TS Agent to initialize the TS structure. TR participates in the problem solving by composing PSMs, also through communication with the TS Agent. When the TS Agent has found the solution to the initial problem (task) in the TS structure, this is returned to TR, and TR forwards the solution to the requester of the initial problem (task). This can be verified by looking at the test-run scenarios described in this chapter.

- The Problem Solver (PS) is able to use the TEAM SPACE. PSs part of a team participates in the problem solving by executing actions, through communication with the TS Agent. This can be verified by looking at the test-run scenarios described in this chapter.

- The TMST stores the initial input for the problem to be solved and a team ID. This can also be verified by looking at the test-run scenarios described in this chapter.

- Through the previous items of this list we conclude that the problem solving steps *solving of subtasks* and *integration of partial solutions* are successfully realized in the CoPS framework prototype.

7.4.3 The Main Extension to CoPS: TEAM SPACE

The functional requirements of the TEAM SPACE architecture were described in chapter 3 - subsection 3.5.2. In chapter 4 an architecture was proposed that fulfilled the functional requirements. All parts of the architecture fulfilling the requirements have been implemented, and the implementation details are described in chapter 6. The test-run scenarios described in this chapter may be used to verify that all implemented parts of the TEAM SPACE work according to the requirements. The results are:

- The TEAM SPACE stores partial results from both PSs and TR.

- The TEAM SPACE does at any time keep the correct state of the problem solving process.

- The TEAM SPACE communicates the correct information, efficiently, to the PSs and the TR (the problem solving team).
- The TEAM SPACE handles the information received from the PSs and TR, and updates the state of the problem solving process accordingly.
- The TEAM SPACE handles concurrent problem solving, by allocating a separate working area (TS structure) for each problem solving team.
- The TEAM SPACE converts the knowledge stored in the TMST, when initializing the TS structure for a problem solving team.

Results from the test-run show that all of the implemented propositions for corrections and extensions to the CoPS framework are working the way we wanted. This is probably because of the incremental development approach we used. First, all the corrections to the CoPS framework prototype was implemented and properly tested. Next, we performed cycles of modeling extensions, implementing them as part of the CoPS framework prototype and testing them (on our checkup example). The tests were evaluated, and then we started from the beginning working with the models (architecture) of the extensions again.

But, there still remains a lot of work with the CoPS framework. Some of the proposed corrections and extensions to the CoPS framework have not been implemented, due to lack of time and the chosen focus of this master thesis. Some of the architectural CoPS components are not developed at all. And most of the components that have been developed need to be improved. We continue this discussion in the next chapter, 8.

7.5 Summary

In this chapter we have used the CoPS framework prototype, to implement an application from a medical domain; the checkup example application. This involved implementing a Checkup Ontology, a Checkup TMST, and a set of agents that were specialized to solve the checkup problem by extending the abstract CoPS agent classes. Several test-run scenarios of our checkup example application were described. The descriptions were used as a basis for discussing the results of our implemented prototype. The conclusion from this discussion is that all implemented corrections and extensions to the CoPS framework prototype work as we planned for.

Chapter 8

Conclusions and Future Work

In this master thesis we have been working with a FIPA compliant framework architecture for cooperative distributed problem solving agents. A shared memory structure has been proposed, to coordinate a team of agents working on a shared task.

This chapter summarizes the work done through discussion and conclusions, in section 8.1. Section 8.2, lists some proposition for future work with the CoPS framework architecture and prototype.

8.1 Discussion and Conclusions

In this section, we first give a short introduction to the current CoPS framework. Second, we summarize the accomplished objectives. Third, we give some arguments for our TEAM SPACE approach. And finally, we mention some critical points of the CoPS architecture.

8.1.1 The Current CoPS Framework

A problem is decomposed into subtasks as a TMST. The TMST represents several ways of solving the initial task. A Task Responsible (TR) extracts the leave nodes of the TMST, which are actions. Then, TR requests capable Problem Solvers (PSs) if they are willing to perform those actions. When the TR knows which PSs are offering to perform the actions, it is able to generate a solution space. The solution space reflects the current configuration of the system. Each of the PSs have proposed a cost for performing an action. These cost values are used to generate the chosen solution of the TMST. Nodes part of the chosen solution are tagged. PSs connected to tagged action - nodes are invited to join the TR's problem solving team.

The chosen solution of a TMST, works as a plan for the TR and PSs part of a problem solving team. A shared memory structure, the TEAM SPACE, is applied by the TR and PSs during the problem solving. In the TEAM SPACE, the plan represented by the TMST is automated by generating a set of rules. The rules cover the dependencies between nodes in the TMST solution, and thus

between PSs and TR. In addition the TEAM SPACE keeps knowledge about agents' partial results. The rules "reason about" this knowledge, and uses it to generate a correct state of the problem solving process.

TEAM SPACE notifies PSs when their actions are ready to be executed, and they return their partial results back to the TEAM SPACE. TR is responsible for combining partial results. TEAM SPACE notifies TR about partial results when they have been produced by the PSs. This communication pattern resembles a publish/subscribe model. When the TR has combined the last set of partial results, these are returned to the TEAM SPACE. Then, the TEAM SPACE is able to recognize that the solution to the initial problem is met, and the solution is returned to the TR.

The TEAM SPACE consists of two main components; TS Agents and the TS Structures. A TS Structure keeps the rules and knowledge related to a specific problem solving team. And the TS Agent acts as the control function of the TEAM SPACE. Responsibilities of the TS Agent are:

- ... to make sure that a TS structure is instantiated and specialized for a problem solving team, based on the solution of their TMST. Specialization involves generating rules and adding initial knowledge, and is performed by the different TS Structure components.
- ... to notify agents about actions that are ready to be performed and partial results that are ready to be composed.
- ... to update the TS Structure with partial results from agents.
- ... and finally to inform the TR when there is a solution to the initial problem.

As the CoPS framework is extended with the TEAM SPACE, it resembles a blackboard architecture. The TS Structure could be compared to the blackboard, the TS Agent could be compared to the control component, and the CoPS agents could be compared to knowledge sources.

Our proposed CoPS framework architecture is a multiagent framework for cooperative problem solving, lending methodologies from the blackboard and publish/subscribe paradigms.

8.1.2 Accomplished Objectives

In chapter 1, we introduced the objectives of this master thesis. Here we present the results from those objectives:

- *Correct problems in the pre-existing CoPS framework.*

  Corrections to the pre-existing CoPS framework were proposed in chapter 3. All of these corrections have been implemented in the CoPS framework prototype - except the one that involves a full realization of dynamic formation of problem solving teams. During the implementation, additional problems were discovered. These problems were also corrected. Implementation details are covered by chapter 6. The implemented corrections were proved to be working by the test-run of our checkup example application, described in chapter 7

- *Modify the pre-existing CoPS framework, in such a way that a shared memory structure could be integrated.*

  Extensions to the CoPS framework were proposed in chapter 3. All of the proposed extensions to the pre-existing CoPS framwework, except the CoPS Ontology, have been realized. These involve extensions to the agents PS and TR, and to the TMST. Modifications to the PS and TR architectures were described in chapter 4. The modifications to their architectures were realized in the CoPS framework prototype. Extensions to the TMST were also realized. Implementation details are covered by chapter 6. The implemented extensions were proved to be working by the test-run of our checkup example application, described in chapter 7.

- *Model an architecture for a shared memory structure and integrate it with the pre-existing CoPS framework.*

  The main extension to the CoPS framework was the TEAM SPACE, which conceptualized a shared memory structure. Functional requirements for the TEAM SPACE were listed in chapter 3. The TEAM SPACE architecture was thoroughly explained in chapter 4. Our proposed architecture covered all of the functional requirements from chapter 3. All components of the TEAM SPACE architecture, that were realizing the functional requirements, were also realized in the CoPS framework prototype. Implementation details are covered by chapter 6. The implemented TEAM SPACE was proved to be working in accordance to the functional requirements by the test-run of our checkup example application, described in chapter 7.

Based on this, we may state that the outputs of our project are: 1) A modified CoPS framework architecture with an integrated shared memory structure - TEAM SPACE. 2) A CoPS framework prototype (API), realizing the architecture. 3) And a checkup example application, implemented using the CoPS framework API.

8.1.3 Arguments for Our TEAM SPACE Approach

We chose to conceptualize the TEAM SPACE into the two main components TS Agent and TS Structure. A TEAM SPACE may consist of several TS Agents and TS Structures. A TS Structure is related to one problem solving team. And a TS Agent manages the TS Structure on behalf of the agents part of that problem solving team, and uses it to coordinate the agents and their actions. A TS Agent may handle several problem solving teams in parallel. Then, TS Agent keeps one TS Structure for each of the teams. These are the arguments for our approach:

- The approach is flexible. If there are few available TS Agents in the system, they may take on more work. The TS Agents have one TS Structure for each of the problem solving teams. When it is needed it can handle several TS Structures simultaneously. When the system is overloaded with problem solving teams, one could add yet more TS Agents. Then, handling of the teams' TS structures are divided between the TS Agents.

- We have realized a pure message interface between agents part of a problem solving team and the TS Agent. This is clean and elegant. All functionality related to the shared memory structure is encapsulated within the TEAM SPACE.

- Since it is only the TS Agent that accesses a TS Structure:

  - There is no need for TS Structure access control. Since only one TS Agent may access a particular TS Structure.

  - The TS Agent may filter the information from TRs and PSs, in order to recognize erroneous information at an early stage. For example, when a PS is not able to execute an action, or when one of the agents do not respond to the request messages sent by the TS Agent.

  - The TS Agent may recognize if the received information could be used by agents part of another problem solving team. This can be done, because the TS Agent has an overall view of several TS Structure components (belonging to the teams it is coordinating).

  - PS and TR does not need any information about the shared memory structure. PS only needs to know how to handle the messages received from the TS Agent. TR also needs to know how it should contact a TS Agent to make it initialize a TS Structure for its team.

- Everywhere there is a TS Agent, there is a TEAM SPACE. Thus, our TEAM SPACE approach is ideal for distributed problem solving.

The TS Structure needed some mechanism for executing the plan represented by the TMST, and reason about partial results. We chose to use rules for this purpose. Because the knowledge about task dependencies in the TMST could easily be converted into rules. This is also true for the dependencies between the different types of nodes in the TMST. The dependencies could be defined in IF .... THEN - sentences. And at the same time, these rules may "reason about" partial results, which are also related to the TMST.

8.1.4 A Critical View of CoPS

In the current CoPS framework architecture and API, there are several critical points. We mention some of them here:

- When the TR uses the composed TMST to form a team of PSs, it sends call-for-proposal messages to all available agents capable of all actions part of the TMST. If the TR would choose one solution of the TMST and try form a team capable of this solution. If it was not possible to form a team, another solution should be tried. The number of sent messages could be decreased, but it still depends on the domain, number of agents in the system, etc. We have not tested which is the best solution in our hospital domain.

- In the implemented API, the best solution of the TMST is decided by its cost only. This may not always be the best heuristic.

- A lot of message are exchanged between TS Agent and the agents part of a problem solving team, during the problem solving. If the PSs and TRs would be able to access the TS Structure, the number of messages could be decreased. But, this advantage should be weighed out against the advantages of the current approach.

In addition, the current architecture and API has some critical points due to parts of the architecture that are not yet modeled. These are not mentioned here, but proposed for future work in the next section.

8.2 Future Work

At this point, the CoPS framework architecture is still not complete. And the implemented API is still just a prototype. Propositions for future work with CoPS are parted into two different categories. The first category covers the work proposed for this master thesis that was not accomplished:

- Dynamic formation of problem solving teams are not fully realized. Reconfiguration of the team is not possible if agents happen to change their minds about joining the team, are disconnected, or fail in any other way. By now, team formation is dynamic in the sense that it is formed according to the state of the system at the exact time it is needed. But, the CoPS framework architecture proposes that the team may change in real-time, depending on external changes to the environment. The TEAM SPACE could be used to assist in this process.

- Modeling and implementing a CoPS Ontology. The ontology should cover concepts about the problem solving in CoPS. An implementation of the CoPS Ontology would lead to modifications to other parts of CoPS as well.

- Handling failures properly if they occur during the team formation or during the problem solving in the TEAM SPACE. In the current version, failures are just specified and a failure message are sent to the relevant receiver.

- The TEAM SPACE architecture implied some limitations to the structure of a TMST (described in chapter 4). Remove these limitations.

The second category of future work, deals with work related to architectural components, and work to improve the current architecture:

- Implement the Personal Assistant (PA).

- Do something about the critical points of the CoPS framework, mentioned in the previous section.

- Solve the problem of unique name constraint in TMST implementations. In the implemented TMST, the nodes must have unique names. This is bothersome when we have several action - nodes in the TMST that represent the same action.

- Use the CoPS Framework API to implement applications from different domains.
- Extend PS and TR in such a way that they may actually reason about the problems.
- Modify the TS Agent in such a way that it would recognize similar tasks in its parallel TS Structures.
- Evaluate new possibilities for use of the TEAM SPACE, like giving the agents an opportunity to request information from the shared TS Structure.

8.3 Summary

This chapter has provided a discussion and conclusions about our work with this master thesis. We have, among other things, described our accomplished objectives. These objectives were introduced in chapter 1, and further elaborated in chapter 3. The proposed work for this master thesis presented in chapter 3, that has not been done, was described as future work.

Appendix A

Content of the Enclosed Zip-File

The delivery of this master thesis is enclosed with a zip-file, *master_deliverance.zip*. In this appendix we list the content of this file. The top-level folder has 3 folders: *javadoc*, *source_code* and *checkup_example*.

A.1 The *javadoc* folder

This folder contains the javadoc for the implementation of the CoPS framework prototype. The submitted javadoc include both the implementation done in this work, implementation done in earlier work with CoPS and the implementation of our checkup example application. The javadoc is not complete for the earlier parts of the CoPS framework prototype. But, most of the classes that were implemented during our work here, is commented. The documentation is parted into three folders: *teamSpace*, *cops* and *tmst*. These are also the names of the top-level packages of our implementation.

A.2 The *source_code* folder

This folder contains the source code of the implemented java-classes. The submitted java classes include both the implementation done in this work, implementation done in earlier work with CoPS and the implementation of our checkup example application. The classes are enclosed in three different folders: *teamSpace*, *cops* and *tmst*. The classes in each of the folders are contained in the packages of the same name as the folders.

A.3 The *checkup_example* folder

This folder contains the files needed to run the checkup example application that was described in chapter 7. The files produced from the test-run scenarios described in chapter 7 - section 7.3, are contained in the folders *thesis-test-run-parallel* and *thesis-test-run-single*. For each of the scenarios there are a text file containing the log(s) of the TR(s), and a *jadegui* - document, which

shows the sequence diagram copied from the JADE GUI at run-time. Jar-files, including the jar-files representing the CoPS framework prototype and the checkup example application are contained in the *lib* folder. Appendix B describes how to run the checkup example application, and how to view the *jadegui* files from the checkup test-run scenarios.

Appendix B

How to Run the Checkup Example Application

The code of the CoPS framework prototype, and the code of our example application is contained in the zipped file enclosed with this master thesis, as described in Appendix A.

B.1 Running the Checkup Example Application in UNIX

The Checkup Example Application consists of two main parts: The checkup TMST and the checkup agents. The checkup agents (the application) could be run with or without the Jade GUI.

- To run the TMST, move to the *checkup_example* folder and write *./tmst_eksekver* in the terminal window. One might have to set the permission to execute the bash file, if access is denied. Then, write this in the terminal window: *chmod 755 tmst_eksekver*.
- To run the checkup application, where the agents solve a single problem, move to the *checkup_example* folder and write *./cops_single_problem* in the terminal window. One might have to set the permission to execute the bash file, if access is denied. Then, write this in the terminal window: *chmod 755 cops_single_problem*.
- To run the checkup application, where the agents solve two problems in parallel using one TS Agent, move to the *checkup_example* folder and write *./cops_two_problems1* in the terminal window. One might have to set the permission to execute the bash file, if access is denied. Then, write this in the terminal window: *chmod 755 cops_two_problems1*.
- To run the checkup application, where the agents solve two problems in parallel using two TS Agents, move to the *checkup_example* folder and write *./cops_two_problems2* in the terminal window. One might have to set the permission to execute the bash file, if access is denied. Then, write this in the terminal window: *chmod 755 cops_two_problems2*.

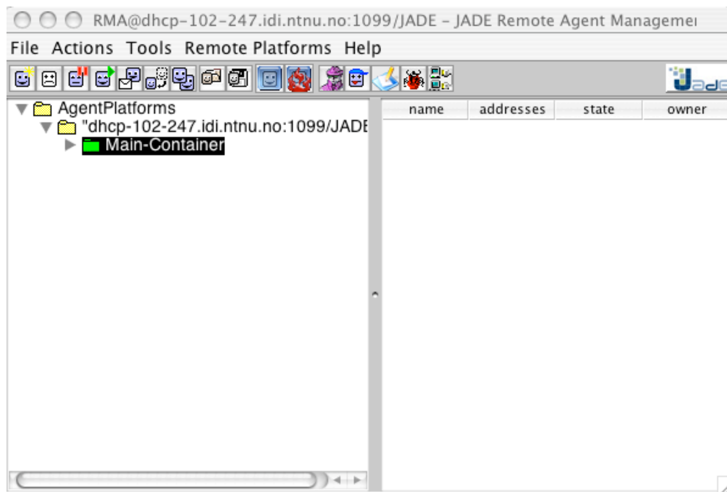

Figure B.1: The starting window of the Jade GUI.

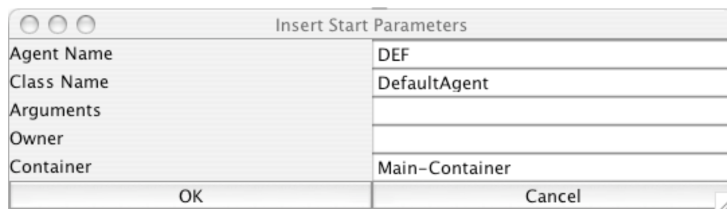

Figure B.2: This illustrations shows how to fill out the parameters, when starting a new agent in the Jade GUI.

To run the different checkup application scenarios with a Jade GUI, the first procedure is the same as described above, only extend the *./cops_...* commands with *_gui*. One might also have to set access rights on these files as well. Then, do as described above. The the Jade GUI illustrated in figure B.1 appears at the screen. By using the GUI, one can watch agent interactions during the problem solving process steps. To achieve this follow these instructions:

1. Select *Tools / Start Sniffer* from the menu. A window for a sniffer agent opens.

2. The sniffer agent window has two parts. Open the catalogues on the right side until one see a list of the different agents in the system. Right click the agents and chose *Do sniff this agent(s)*, for the agents one want to observe.

3. Then one have to start a DefaultAgent (that we have implemented). The DefaultAgent starts the problem solving process by sending request message(s) for the TR(s). To start the agent go back to the window illustrated in figure B.1. Select the folder *Main - Container* in the folder list. Then, select *Actions / Start New Agent* from the menu. And fill out the pop-up form like depicted in figure B.2.

4. Go back to the sniffer agent window and watch the interactions between the agents one have chosen to "sniff". The arrows represent messages. To inspect the content of a message, double-click the arrow.

To look at the *jadegui* documents described in appendix A, one first move to *checkup_example* folder an write *./jade_gui* in the terminal window. The Jade GUI appears on the screen. Then, start the sniffer agent like described above. In the window of the sniffer agent, select *Action / Open Snapshot File*, and open one of the *jadegui* files, in subfolders of *checkup_example*: *thesis-test-run-single* or *thesis-test-run-parallel*.

B.2 Running the Checkup Example Application in Windows

To run one of the examples described above in Windows, one should copy the content of the batch file and paste it in a Windows command (terminal) window. Then, replace all / with \ and all : with ; (not the ones after the agent-names). Besides this, one follows the instructions described in the previous section.

Bibliography

[1] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa. JADE PROGRAMMER'S GUIDE, 2005. http://jade.tilab.com/doc/programmersguide.pdf.

[2] Fabio Bellifemine, Giovanni Caire, Giovanni Rimassa, Agostino Poggi, Tiziana Trucco, Elisabetta Cortese, Filippo Quarta, Giosu Vitaglione, Nicolas Lhuillier, and Jrme Picault. Java Agent DEvelopment Framework (JADE). http://jade.cselt.it/index.html.

[3] M. Brian Blake, David H. Fado, and Gregory A. Mack. A publish and subscribe collaboration architecture for web-based information. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1164–1165, New York, NY, USA, 2005. ACM Press.

[4] Jan Bosh. *Design and Use of Software Architectures - Adopting and developing a product-line approach.* Addison-Wesley, 2000.

[5] Giovanni Caire. JADE TUTORIAL Application-Defined Content Languages and Ontologies, 2004. http://sharon.cselt.it/projects/jade/doc/CLOntoSupport.pdf.

[6] Daniel D. Corkill. Blackboard Systems. *AI Expert*, 6(9):40–47, 1991.

[7] Daniel D. Corkill. Collaborating Software: Blackboard and Multiagent Systems and the Future, 2003. Proceedings of the International Lisp Conference.

[8] G. Edwin and M. T. Cox. COMAS: Coordination in Multiagent Systems, 2001. Edwin, G., Comas: coordination in multiagent systems, Master's thesis, Wright State University, Dayton, OH, 2001.

[9] Robert Engelmore and Tony Morgan, editors. *Blackboard Systems.* ADDISON-WESLEY PUBLISHING COMPANY, 1988.

[10] Ernest J. Friedman-Hill. Jess the Rule Engine for the Java Platform (Jess 6.1 Manual), 2005. http://www.jessrules.com/jess/docs/61/.

[11] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[12] R. Flores-Mendez. Towards the Standardization of Multi Agent Systems Architectures: An Overview, 1999. ACM Crossroads - Special Issue on Intelligence Agents, Vol. 5 (4), ACM Press, Summer, 1999.

[13] IEEE Foundation for Intelligent Physical Agents. The foundation for intelligent physical agents, 2005. http://www.fipa.org.

[14] Darren Foster, Carolyn McGregor, and Samir El-Masri. A Survey of Agent-Based Intelligent Support Systems Support Clinical Management and Research. In *Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.

[15] Frode Srmo. jCreek. Webpage: http://dionysus.idi.ntnu.no/newcreek/jCreek-ProgrammersGuide.doc.

[16] QFD Institute. Quality Function Deployment (QFD), 2006. http://www.qfdi.org/.

[17] V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors. *Blackboard Architectures and Applications*. ACADEMIC PRESS, INC., 1989.

[18] N. R. Jennings. Coordination techniques for distributed artificial intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 187–210. John Wiley & Sons, 1996.

[19] Hsing-Pei Kao, Eric Su, and Brian Wang. I2QFD: a Blackboard-Based Multiagent System for Supporting Concurrent Engineering Projects. *International Journal of Production Research*, 40(5):1235–1267, 2002.

[20] David Keil and Dina Goldin. Modeling Indirect Interaction in Open Computational Systems, 2003. In Proceedings of the Twelfth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE03).

[21] Elizabeth A. Kendall. Agent Roles And Role Models : New Abstractions For Intelligent Agent System Analysis And Design, 1999. ECOOP'99, AOP Workshop, Lisbon, 14.06.99.

[22] Rick Kazman Len Bass, Paul Clements. *Software Architecture in Practice - Second Edition*. Addison-Wesley, 2003.

[23] LCC Lexico Publishing Group. Dictionary.com. www.dictionary.com.

[24] Yosuke Matsusaka and Tetsunori Kobbayashi. System Software for Collaborative Development of Interactive Robot, 2001. In Proceedings of the IEEE.

[25] Jason Morris. The Zen of Jess 2, 2005. http://www.jessrules.com/jess/zen.shtml.

[26] Pinar Öztürk. Lecture Notes, TDT4280 Distributed Artificial Intelligence and Intelligent Agents. Lectured at NTNU, Norway, 2004.

[27] H. Van Dyke Parunak. Practical and Industrial Applications of Agent-Based Systems, 1998. Parunak (1998). Practical and Industrial Applications of Agent-Based Systems. http://www.cs.umbc.edu/agents/.

[28] Utpal Roy and Jianmin Liao. Application of a Blackboard Framework to a Cooperative Fixture Design System. *Computers in Industry*, 37(1):67–81, 1998.

[29] George Rudolph. Some Guidelines for Deciding Whether to Use a Rules Engine, 2003. http://www.jessrules.com/jess/guidelines.shtml.

[30] Ozgur Koray Sahingoz and Nadia Erdogan. A Two-Leveled Mobile Agent System for Electronic Commerce, 2003. "In the journal of Aearonautics and Space Technologies Institute (ASTIN)".

[31] Ozgur Koray Sahingoz and Nadia Erdogan. MAPSEC: Mobile-Agent Based Publish/Subscribe Platform for Electronic Commerce, 2003. In Computer and Information Sciences - ISCIS 2003, p. 348-355.

[32] Sandia National Laboratories. Jess the Rule Engine for the Java Platform. Webpage: http://herzberg.ca.sandia.gov.

[33] Kari Rssland (supervisor: Pinar Öztürk). Application of CoPS Multiagent Framework in a Medical Domain, 2005. TDT4745 Knowledge Systems, depth study Autumn 2005.

[34] Odd Erik Gundersen (supervisor: Pinar Öztürk). A MultiAgent Framework for Collaborative Problem Solving, 2004. A master thesis submitted to the Norwegian University of Science and Technology, Department of Computer and Information Science, Trondheim, Norway, July 30.

[35] Katia P. Sycara and Dajun Zeng. Coordination of Multiple Intelligent Software Agents. *International Journal of Cooperative Information Systems*, 5(2):181–212, 1996.

[36] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Massachusetts Institute of Technology, 1999.

[37] Michael Wooldridge. *An Introduction to: Multiagent Systems*. John Wiley & Sons Ltd, 1996.