Improving Performance of Blackboard Systems

Emilio AGUSTIN MOLINA

ARTIFICIAL INTELLIGENCE LIBRARY
UNIVERSITY OF EDINBURGH
80 South Bridge
Edinburgh EH1 1HN

Ph.D.

University of Edinburgh

1995

Declaration

I declare that this thesis has been composed by myself and that the work described is my own

Emilio AGUSTIN MOLINA

[illegible] INTELLIG[illegible]
[illegible]ERSITY OF [illegible]
8[illegible] South [illegible]
Edinburgh EH1 1[illegible]N

To my wife, Maria Blanca, who suffered this thesis more than I did,
and to Ana Cristina, my new born baby girl.

Acknowledgements

I would like to acknowledge all the people that gave me their support without which this thesis would have been impossible.

I would like to thank my supervisors, Robert Rae who welcomed me to Edinburgh and guided me all these years, Karl Millington who helped me at the beginning of my studies, Tim Smithers whose comments and observations really put me on the right tracks, and Bob Fisher who gave me the last push to get the thesis finished. Without your comments, suggestions and advise this thesis would have been impossible.

I would like to thank all my friends in Edinburgh who made my life a lot more interesting. I would like to thank John Beaven, who made me feel at home VERY quickly. I would like to thank everybody at the Artificial Intelligence Applications Institute. Thanks to Ragiv Trehan and Maria Cabral, Luis Castillo y Gloria Quintanilla, Jussi Stader, Tim Duncan, Andy Bowles, Flavio Soares, Rob Scott, Amaia Bernaras, Marisa Barja and Tore and all the others I am surely forgetting for their friendship.

I would like to specially thank René Bañares, Arantza Aldea and Rocio Aguilar for their hospitality and help in these last moments of my PhD studies.

I would also like to thank my examiners, Iain Craig and Richard Baldock for their comments during the viva.

Finally I would like to thank María Blanca Ibáñez, my wife without your love and support this thesis would not be here.

Abstract

In this thesis, we deal with blackboard system performance issues. We show that blackboard system performance can be improved using parallel processing strategies and a novel blackboard architecture.

We study traditional blackboard architectures using a novel performance framework. This is a useful tool for directing system optimisation efforts. We present the analysis of four blackboard systems present in the literature.

Besides localised optimisation efforts, one of the most promising approaches for improving blackboard system performance is the use of parallel processing techniques. However, traditional blackboard architectures present both data and control contention when implemented in parallel.

In this thesis we present a novel blackboard architecture, the *Active Blackboard Architecture* (ABB). We based ABB on a novel variation of the traditional "Blackboard and Experts" metaphor, called *"Blackboard, Experts and Desks"*. This new metaphor introduces a new element, the desks, used by the experts to perform their work.

The ABB architecture is based on an active blackboard, capable of processing on its own, and a decentralised control model. This avoids control contention and bottlenecks. We describe this architecture using the Z specification language, and implemented and evaluated in the EPCC Meiko Computing Surface, a multi-transputer distributed memory parallel machine.

The ABB Parallel prototype is an object oriented implementation of the ABB model that overcomes both data and control bottlenecks by having a distributed blackboard and using the ABB control model. Based on a series of experiments, we show that the new architecture allows to achieve much greater effective parallelism in a blackboard system. We also present some ways in which the system can be tailored to specific application needs, improving in this way its overall performance.

Contents

Declaration 1

Acknowledgements 3

Abstract 4

Table of Contents 5

1 Introduction 8

1.1 Introduction 9
1.2 Problem and Domain 11
1.3 Organisation of the Thesis 13

2 A Blackboard System Performance Framework. 14

2.1 Introduction 15
2.2 Basic Definitions 15
2.3 Blackboard System Structure 18
2.4 Blackboard System Performance 29
2.5 Summary 43

3 Case Studies 44

3.1 Introduction 45
3.2 The Jigsaw Puzzle (JSP) Solver 45
3.3 Case 1: BB1 47
3.4 Case 2: EPBS 53
3.5 Case 3: Blondie-I 59
3.6 Case 4: Blondie-III 65
3.7 General Discussion 73
3.8 Summary and Conclusions 75

4 The ABB System **77**

4.1 Introduction 78

4.2 Blackboard, Experts and Desks. 81

4.3 The Active Blackboard (*ABB*) Model 82

4.4 ABB Model Specification 89

4.5 Summary and Conclusions 106

5 The ABB System Implementation **108**

5.1 Introduction 109

5.2 The ABB Parallel Prototype 109

5.3 Prototype System Implementation 126

5.4 Summary and Conclusions 131

6 The ABB System Performance **132**

6.1 Introduction 133

6.2 Test Suite 134

6.3 Prototype Performance 141

6.4 Conclusions 148

7 Conclusions **150**

7.1 Summary and Conclusions 151

7.2 Further Work 153

Bibliography **156**

A A Brief Introduction to Blackboard Systems **164**

A.1 Introduction 165

A.2 The blackboard model 166

A.3 Blackboard System Components 170

A.4 Parallelism on the blackboard model 173

B A Glossary of the Blackboard Performance Framework Terms **176**

B.1 Basic Terms 177

B.2 Blackboard System Performance 180

C Jigsaw-Puzzle Solver, Sample Implementation: EPBS **183**

D An Introduction to Z Terms and Expressions **192**

E The ABB Model Detailed Specification **197**

E.1 Introduction 198
E.2 Knowledge Sources 200
E.3 Blackboard 204
E.4 Blackboard Operations 210
E.5 Desks 213
E.6 The ABB System 214
E.7 Global System Operations 216
E.8 ABB System Operation 223

F ABB Communication Mechanism **225**

G Jigsaw-Puzzle Solver, ABB Implementation **229**

Chapter 1

Introduction

1.1 Introduction

It is universally acknowledged that the main features of the blackboard model arose from abstract features of the Hearsay-II [LE77, EL78, EHRLR80] speech-understanding system developed at Carnegie-Mellon University between 1971 and 1976. The goals of the project were to provide recognition of utterances from a limited vocabulary in near-real time. Hearsay-II recognises connected speech in a 1000-word vocabulary interpreting correctly 85% of test sentences. Although Hearsay-II was less successful than other systems in pure performance terms, it introduced a number of novel ideas that were to prove fruitful in many applications.

The Hearsay-II system introduced a new form of problem-solving based on the metaphor of a group of experts trying to solve a difficult problem together with the help of a blackboard. The initial blackboard model had two basic elements, a global structure called the *Blackboard* and a set of experts called *Knowledge Sources*. These knowledge sources use a blackboard as the common vehicle for cooperation.

The blackboard holds information about the current status of the problem. The knowledge sources look at the blackboard, when they see a problem that falls within their specific area of expertise, they solve it and place the solution back on the blackboard. The knowledge sources, then, are seen as having a two-part structure. The first part identifies the problems that the knowledge source knows how to solve, and the second part actually solves the problem and stores the solution on the blackboard. Knowledge sources are not allowed to interact with each other directly, they have to communicate via the blackboard. Figure 1.1 shows a graphical representation of the initial blackboard model.

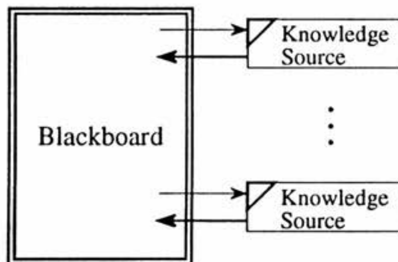

Figure 1.1. Initial Blackboard Model

When this model was implemented in the Hearsay-II system, the authors found that at any given moment they had several knowledge sources that had identified

appropriate problems and wanted to solve them. The initial model assumed a sequential environment, so that only one knowledge source could be executed at a given time. The system had to order the knowledge source operation in some way. The way they solved the problem was by having each knowledge source record its intention to execute in a global agenda. They then included a scheduler that ordered them, and chose one for execution at a given moment. The main function of this control component was to avoid a combinatorial explosion of the search space, and meet the requirements of the speech understanding problem. The inclusion of the scheduler introduced the possibility of expressing control knowledge (or metaknowledge), and including it into the system in a very natural way. They also refined the blackboard concept further by giving it a structure where elements in the blackboard could be organised in different levels. Figure 1.2 shows a graphical representation of the Hearsay-II implementation model.

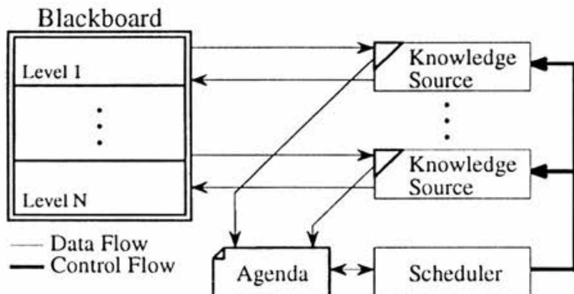

Figure 1.2. Hearsay-II Implementation Model

Further developments on blackboard systems included changes to the structure and operation of the blackboard, knowledge sources and scheduler. For example, the GBB system [GCJ88] introduces a special method for storing and retrieving information from the blackboard that improves its performance. The BB1 system [HR84, HRH85, AMM87] not only stores information about the problem domain, but also stores information about the problem solving activity. Systems such as HASP/SIAP [NFAR82] and BB1 also changed the way the scheduler works. HASP/SIAP included knowledge about expectations, goals and events into the scheduler. BB1 simplified the operation of the scheduler; its main task is to follow the current control plan, which is built by specialised knowledge sources.

The Hearsay-II system showed the blackboard architecture to be general enough

to be applied in other problem domains. Using the Hearsay-II experiences, several other systems were developed in widely different problem domains such as ocean surveillance [NFAR82], protein analysis [Ter83, BJL+84], planning [HRHRRC79], design [Der87, PSC+86, VC86, Sri86], medical diagnosis [Aie83], vehicle monitoring [LC83] and autonomous mobile robots [Elf86]. The experiences of some of those systems were used in developing general all-purpose expert system shells in order to facilitate the development of blackboard systems. Examples of those systems are AGE [NA79], HEARSAY-III [ELF81], Stanford's BB1 [HR84], Massachusetts's GBB [GCJ88], Boeing's BBB [BJD86] and Edinburgh's EPBS [JM86].

1.2 Problem and Domain

In this thesis, we will deal with blackboard system performance. Our main goal is to show that blackboard system performance can be improved successfully using parallel processing strategies.

The Hearsay-II system already faced performance problems when compared to other speech-understanding problems of the time. The Hearsay-II group begun exploring some ways in which they could improve this performance. They were the first group to consider using parallel processing within the blackboard architecture. In fact, good behaviour while executing in a multiprocessor was one of the initial design goals for Hearsay-II. They performed some simulations to test this behaviour. The simulation data for the initial system configuration showed a speed-up of four to six. These results were less than expected. When they analysed the possible causes for this poor performance, they found that it was mostly caused by the type of locking scheme that they used for insuring blackboard consistency.

The issue of performance is present in most blackboard system research. For example, GBB [GC89] concentrated in providing a powerful and efficient blackboard structure. GBB in turn, was used to tune DVMT [LC83]. DVMT itself, used a distributed approach to fit its problem domain's spatial distribution.

One of the appeals of the blackboard model is that it has several characteristics that make it a good candidate to be considered as an inherently parallel model. The model is based on the interaction of a group of completely independent entities (the knowledge sources) using a very well defined communication medium (the blackboard). This simplicity makes it a very clean and seemingly useful model for parallel problem solving. This, however, is not completely true.

All current blackboard systems base their particular architectures on the Hearsay-II implementation model. In particular they are all based on the idea of a centralised scheduler. The way this mechanism integrates with the blackboard model in a parallel or distributed environment is a problem that has not been properly solved. The amount and strength of control knowledge that is optimal for solution formation is a serious and difficult question. Too little control leads to very inefficient and ineffective problem solving. Too much control "serialises" the computation excessively, therefore losing any advantages offered by parallel or distributed processing.

We consider that many of the performance problems confronted by parallel and distributed systems arise from the fact that they try to adapt the idea of centralised resource management to a non-sequential environment. In blackboard systems this is represented by the central scheduler. Distributed systems partially avoid the problem by assuming that the problem to be solved can be spatially decomposed, and tackling each subproblem with a separate blackboard system (each with its own centralised scheduler). Examples of these systems are DVMT, Blondie-III [FvLV88], CASSANDRA [Cra89], and the GEST [GRT89] system. This last system used a hierarchically organised distributed architecture. Parallel systems, however, do face the problem of adapting the scheduler operation. For example, the CAGE [Aie86] system has a Hearsay-II-like scheduler that chooses for execution several knowledge sources at a time (instead of just one). The Blondie-II [Vel91] system also uses this approach. In these systems the scheduler is, effectively, a bottleneck. The POLIGON system [NAR89], on the other hand, solves this problem by not having any scheduler whatsoever. However, we do not consider this to be a good solution, because most of the system processing power can be diverted to solving superfluous problems.

In this thesis we propose a new blackboard performance framework that helps to direct optimization efforts. We also propose a new blackboard architecture, the active blackboard architecture (ABB), based on an active blackboard, capable of processing on its own, and a decentralised control model in an attempt to overcome scheduler contention. This architecture introduces a new system element, the desk, which manages processing resources. This new architecture is based on the "blackboard, experts and desks" metaphor.

The ABB architecture was described using the Z specification language, and implemented and evaluated in a multi-transputer distributed memory parallel machine, the EPCC MEIKO Computing Surface. Based on a series of experiments, we show that the new architecture allows to achieve much greater effective parallelism in a blackboard system. We also show some ways in which the system can be tailored to

specific application needs, improving in this way its overall performance.

1.3 Organisation of the Thesis

The structure of the current work reflects somewhat the approach we followed for achieving our goals. Initially, before we begin proposing some ways in which blackboard system performance can be improved, we need tools for assessing the behaviour of current blackboard systems. We developed these tools in the form of a blackboard performance framework. This framework is presented in Chapter 2.

In Chapter 3 we show the results of applying the proposed framework to some blackboard systems present in the literature. We use these results for analysing some of the ways in which the performance of those blackboard systems could be improved.

One of the most promising approaches for improving blackboard system performance is by making effective use of current parallel computers. However, from the previous experience, we know that a simple adaptation of the blackboard architecture to parallel environments is not enough. We need to re-think the way the different blackboard system components interact in a multiprocessing environment. In order to do this properly, we had to go back to the original blackboard-and-experts metaphor.

In Chapter 4 we propose the blackboard-experts-and-desks metaphor. With this metaphor as the underlying motivation, we developed the active blackboard model. In Chapter 4 we also use the Z specification language for describing formally the features and functionality of the blackboard systems based on the active blackboard model.

In Chapter 5 we describe the implementation of a parallel prototype of an active blackboard system shell. This prototype is an object oriented blackboard shell that profits from the design decisions present in the active blackboard model. This prototype is implemented in the Edinburgh Parallel Computer Center's Meiko Computing Surface, a transputer-based multiprocessor.

In Chapter 6 we show the performance results for the active blackboard prototype. We show the behaviour of the system by running several configurations on different number of processors.

In Chapter 7 we will present the conclusion of the research presented in this thesis, and we will discuss further work.

Chapter 2

A Blackboard System Performance Framework.

2.1 Introduction

In this thesis, our main goal is to show that blackboard system performance can be improved. In order to achieve this, we need a deep understanding on how the general blackboard system architecture works. It is necessary to identify the essence of blackboard systems in terms of their basic elements, structure and problem solving strategy.

In the literature, we find that the various authors implemented the basic blackboard architecture in several different ways. Each one of these implementations shows a variation or extension of the "core" system. They also show how it can be made to work in widely different problem domains. Each domain requires particular capabilities from the point of view of performance and system functionality.

We need a framework within which we can study all these systems from the performance point of view. By comparing blackboard systems using such a framework we can identify the similarities and differences between them, contributing in this way to our understanding of the architecture and its variations.

In [Vel91], Velthuijsen presented a very complete blackboard system framework. This framework's emphasis was in the functional characteristics of blackboard systems. However we need some understanding on the performance of the different components of the blackboard architecture, and their interactions.

In the present chapter, we will present a performance framework that will complement Velthuijsen's work. In our performance framework, we will concentrate in specifying a number of measurements that will help in the analysis of the performance of a particular blackboard system as a whole, as well as of its modules. This will help in deciding whether a particular implementation of a modification or extension of the basic blackboard architecture is a good or bad idea from the performance point of view.

Our framework has two main parts. The first serves to express the structure and architectural issues of a blackboard system. This serves as a basis on which the system performance can be described. The second part of the framework, then, concentrates on the system's performance. Figure 2.1 shows the general outline of the framework.

2.2 Basic Definitions

One problem we encounter when describing blackboard systems is the fact that many systems call the same structures differently. Although there is recently a tendency to unify the nomenclature, there are still many systems using different terms to refer

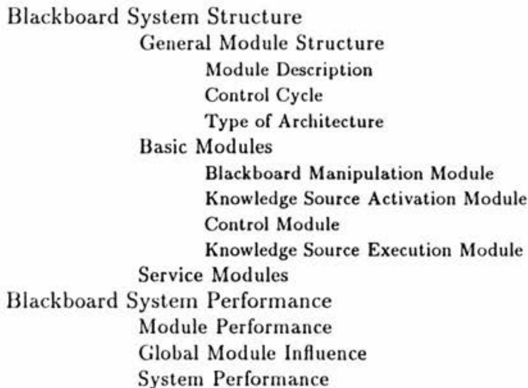

Figure 2.1. Blackboard framework

to similar elements. We will now specify each blackboard element structure using a common nomenclature.

The central structure of any blackboard system is the blackboard. We define every blackboard system as having only one global structure called the blackboard. This structure stores information that is needed by several modules of the system. Initially it may hold the problem hypothesis and initial data. It may also hold control information. With the progress of the problem solving activity it holds partial results and calculated or inferred data.

Definition 2.1 : Blackboard
The blackboard is the global structure that stores information about hypothesis, global data, partial results, goals and control information.

In order to organise the information stored in the blackboard, it is subdivided into several partitions called levels. Information stored in one level is conceptually different from that stored in another level. Usually the different levels of the blackboard are organised in a hierarchy according to abstraction levels in the information of the problem domain. The lower levels in the hierarchy correspond to specialisations of higher levels or they represent problem domain data seen at a lower abstraction level (more specific).

Definition 2.2 : Level
A level is a partition of the blackboard that holds a particular kind of information. Levels can also be subdivided into sublevels, each of them holding a different type of information.

The blackboard levels serve as a repository of items of information of a particular type, in the data-type sense. We call each one of these items blackboard elements (BBels). A bbel is an instance of information of the type represented by the level in which it is stored.

Definition 2.3 : BBel
A bbel (pronounced "bel") or blackboard element, is a complex structure holding a particular instance of a given type of information. This type of information is given by the level in which it is stored.

The information stored within a bbel is structured in internal slots or fields. A bbel can be seen as a PASCAL record, a C structure or an object. A level can be seen as a type declaration and a bbel as an object of that type.

Definition 2.4 : Field
A field represents a particular characteristic that composes a particular piece of information (bbel). A given bbel is formed by a number of fields, one per property.

Another central structure within the blackboard architecture is the knowledge sources. These structures represent the experts that actually solve the domain problem.

Definition 2.5 : Knowledge Source
A knowledge source is the embodiment of the knowledge on a particular domain. The use of this knowledge will allow the system to solve particular problems.

The knowledge held in any single knowledge source does not usually cover the problem domain of the system. Each usually knows how to solve only part of the overall problem. It is the interaction between several knowledge sources that the system's domain problem can be solved. The knowledge sources are subdivided in two distinct parts: the trigger condition and the body.

Definition 2.6 : Trigger Condition
A trigger condition is the part of the knowledge source that identifies its problem domain. It holds the specification of the problem or problems that the knowledge source knows how to solve.

Definition 2.7 : Knowledge Source Body
The body is the part of the knowledge source that knows how to solve its problem domain. It holds the knowledge needed for achieving a solution of a problem identified by the knowledge source's trigger condition.

In all blackboard systems, there is always a control element, usually called the scheduler. This element allows to guide knowledge source activity to the most promising areas. We can use this element to express meta-knowledge about the reasoning process in specific domains. In some systems, this meta-knowledge is included implicitly within the scheduler, usually in the form of and ordering function, or of a knowledge source evaluation heuristic. Other systems provide an elaborate mechanism in which control knowledge can be stored in the blackboard in the form of strategies and heuristics, which will be used by the scheduler to evaluate the pending knowledge sources. Blackboard systems are not restricted to use one scheduler. It can be separated in several, more specific, sub-schedulers.

Definition 2.8 : Scheduler
The scheduler is the part or parts of a blackboard system that decides on the order of execution of the knowledge sources. It is the embodiment of the meta-knowledge not held already in the knowledge sources.

The scheduler's actions are usually directed toward ordering and choosing one of the knowledge sources waiting for evaluation. The set of knowledge sources waiting to be evaluated is usually called the agenda.

Definition 2.9 : Agenda
The Agenda is the set of knowledge sources that are ready, and waiting for evaluation in a given moment of a blackboard system execution.

2.3 Blackboard System Structure

We can describe the blackboard system structure in terms of the different functional modules of the basic blackboard architecture. Any system considered a blackboard system, must include at least, the basic blackboard elements. It may, of course, introduce variations to the structure and functionality of the basic blackboard elements and it can include new ones.

We consider the basic blackboard elements to be the following:

- The global data storage area, called *the blackboard.*
- The embodiment of the knowledge of different experts, called *the knowledge sources.* This knowledge is split in two parts, the knowledge about identifying the problem domain of the expert, called trigger conditions, and the knowledge about how to solve a problem, called the *body* of the knowledge source.
- The control mechanism called *the scheduler.*

Each of these blackboard elements defines a functional module. For each of these modules, the framework will specify the structure and functionality of the particular blackboard system element it implements, and how it is implemented. This is necessary as the characteristics of every system module will depend highly on the structure and functionality of its blackboard system element. For example, the implementation of the blackboard module will depend in a large degree on the possible structure of the blackboard and the structure of the elements it can hold. For the blackboard manipulation module we will first describe the blackboard structure and organisation, and then how this structure is implemented.

The first subsection describes global system characteristics such as module structure, control cycle and type of global architecture. The subsequent subsections describe each system module in greater detail.

2.3.1 General Module Structure

Blackboard systems are usually very modular programs. In general, it is relatively easy to identify several different functional modules in their structure. By identifying those modules and their interactions, the framework describes the global system structure in a simple, uniform and comprehensive way.

We will show the overall module structure by giving the general description of the different modules involved in a particular blackboard system. We will then specify the different module interactions by showing the system control cycle. We will also characterise the type of blackboard architectures.

Module Description

Take the basic blackboard control cycle:

1. Modifications are made to the blackboard.

2. The knowledge source trigger conditions are evaluated. The list of triggered knowledge sources is stored in an agenda.
3. A knowledge source from the agenda is chosen to be performed.
4. Its body is evaluated.

This simple control cycle suggests four module partitions of the basic blackboard architecture. Each module contains the set of user and system procedures needed for the completion of each one of the four steps above. We call them *Blackboard Manipulation Module* (Step 1), *Knowledge Source Activation Module* (Step 2), *Scheduler Module* (Step 3) and *Knowledge Source Execution Module* (Step 4).

All these modules are logical groupings of the system's functions. We do not imply that all blackboard systems should have these functions organised in this way, or at all. Nor do we claim that the functions need be implemented in four identifiable modules. We propose this logical organisation as we think it helps the understanding of how the blackboard architecture works.

We define these logical modules as follows:

Blackboard Manipulation Module. This module embodies all the procedures that create, modify, retrieve and delete blackboard elements, together with all its support procedures.

Knowledge Source Activation Module. This module holds all the knowledge source trigger conditions, as well as all the procedures needed for evaluating these conditions. It also contains procedures that create agenda elements and the ones that update the agenda.

Scheduler Module. It is formed by the set of procedures that decide which knowledge source body will be evaluated next. This includes all the procedures that perform an ordering of the agenda.

Knowledge Source Execution Module. This module holds all the different knowledge source bodies and all the procedures needed for evaluating them.

Note that we have separated the knowledge sources in two functionally different modules. We group the knowledge source trigger conditions in a module separated from the knowledge source bodies. All other authors in the literature prefer to associate each knowledge source with its trigger conditions. They usually see them as a unit. We, however, prefer to consider them as related, but separate logical entities. Both trigger conditions and body refer to the same problem. However, they hold fundamentally different knowledge. The first holds the knowledge of how to identify a particular problem, while the second actually knows how to solve it. We consider these two tasks

to be different enough to justify their individual study.

The Figure 2.2 gives a graphic representation of these modules and their interactions within the basic control cycle.

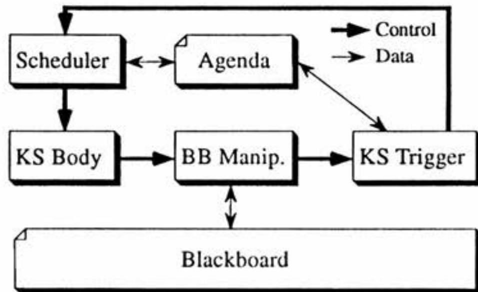

Figure 2.2. A Basic Control Cycle

Control Cycle

The control cycle shown in Figure 2.2 is separated into four distinct sequential steps for the sake of simplicity. Usually these steps are not followed one after the other. For example a given blackboard system could execute a knowledge source body and, within this execution, it can call the blackboard manipulation module and perform a modification to the blackboard. When this modification is finished the blackboard manipulation module can call the trigger evaluation module to see if there is any knowledge source interested in the modification. An alternative system control cycle would be:

1. A knowledge source body is executed. This execution makes some modifications to the blackboard. When a modification to the blackboard is made, the knowledge source trigger conditions are evaluated, and the list of triggered knowledge sources is maintained in an agenda.
2. A triggered knowledge source is chosen to be performed.

Figure 2.3 shows a graphical representation of this control cycle.

In general, the blackboard system control cycle may be very different from system to system. However the four basic modules can always be identified.

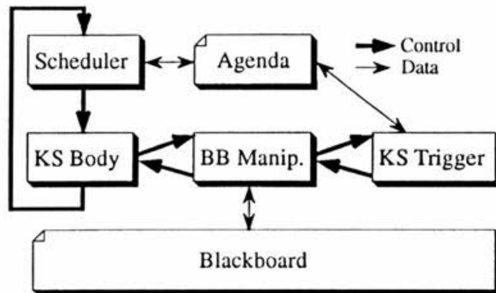

Figure 2.3. An Alternative Control Cycle Example

There are also usually a number of secondary modules that provide particular services to the four central ones. We will call these *Service Modules.* Examples of service modules could be a user interface module, a module for communicating to other systems, a module that provides an interface to a database, etc.

Type of Architecture

All the modules of a system, and their interrelations, form the conceptual architecture of that particular system. We classify these architectures by the way the modules interact with each other and how they depend of the work of the other modules.

We identify three kinds of blackboard architectures: *Sequential*, *Parallel* and *Distributed.*

A blackboard architecture is *sequential* when the interdependencies between the modules make it impossible to evaluate more than one module at a time. In a sequential architecture a module, while interacting with other modules, must either wait for them to finish in order to continue, or must finish in order to let them work.

A *parallel* blackboard architecture is the one in which the evaluation of a particular module can be started or continued without waiting for the previous module to finish. We will call parallel any architecture in which two or more of its components can be evaluated concurrently without changing its results. Note that a sequential blackboard system's architecture may be parallel. For example if, when making several modifications to the blackboard we don't assume that they are made in a specific order, then the interaction will be parallel. This is because we can start one modification and begin the next one without waiting for the first one to finish.

A *distributed* blackboard architecture is the one composed by several clusters of the basic modules having either sequential or parallel sub-architectures that are usually linked by service modules.

Figure 2.4 shows a graphic representation of sample sequential, parallel and distributed architectures.

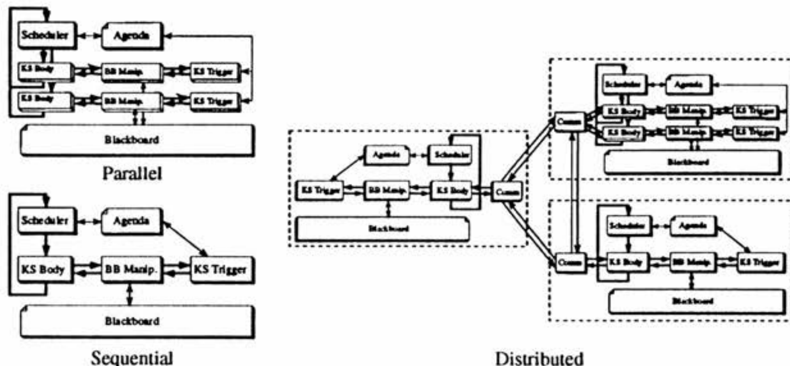

Figure 2.4. Conceptual blackboard Architectures

Note that the architecture of a system is not necessarily the same as the architecture of the computer system on which it is implemented. There can be a system with a parallel or distributed architecture implemented in a sequential computer system and vice-versa.

It is usually very difficult to find a true sequential blackboard architecture. There are always some module interactions that could be done in parallel. For example, the knowledge source trigger conditions can usually be evaluated in parallel as they only access (read) the state of the blackboard and do not modify it. However, a system that only does trigger evaluations in parallel does not necessarily make a good use of the computer resources.

The distributed system example shown in Figure 2.4 is just one of the many arrangements in which distributed blackboard systems can be designed. For example, in the figure the communication module is attached to the knowledge sources, meaning that the knowledge sources make explicit use of the communication facilities. There are other systems in which communication is done through the blackboard. We can represent this by attaching the communication module directly to the blackboard.

2.3.2 Basic Modules

The structure of the four basic modules is highly dependent on the functionality of the corresponding element of the blackboard architecture. They are basically implementations of such structure and functionality. For example, the blackboard manipulation module implements the structure and functionality of the blackboard, and the knowledge source trigger module must support the structure and functionality of the knowledge source's trigger component.

We will follow a common approach when describing each system module characteristic. We will first specify the structure and functionality of its corresponding element and then we will show the relevant characteristics of the implementation itself.

The Blackboard Manipulation Module

The blackboard manipulation module encompasses all the user and system procedures that define and make use of the blackboard. This can be the creation of a structure and storage of it in the blackboard, or the modification of a particular structure already stored in it, or the extraction of a copy of a structure, or the deletion of one from the blackboard. This can also include procedures for searching structures in the blackboard, for testing for their existence and any other procedures that use it directly.

To describe the structure of the blackboard manipulation module, we will first specify the structure and configuration of the blackboard itself. We will then show the operations that a particular blackboard system provides. We will also describe any relevant implementation details.

The mechanism used by different blackboard systems to provide the structure and functionality of the blackboard varies. All systems allow to have the blackboard elements arranged in some sort of hierarchical structure. Elements of different structure are stored on separate levels. In systems such as Hearsay-II [EL78], this element relationship is not reflected in the relationships between levels. These systems only allow simple levels that cannot be subdivided further. Systems such as BB1 [HR84], allow the possibility of having more than one blackboard. Their blackboard is formed by a number of sub-blackboards that can be subdivided into levels. This allows the description of some simple relationships between levels.

In order to distinguish between these different systems we define the function *blackboard depth*. Systems that have a blackboard depth of one, have one blackboard subdivided into several levels and these levels are not subdivided further. The systems that have several blackboards, subdivided into levels, have a blackboard depth of two.

In general, blackboard systems could have their blackboard divided in any number of nested levels. We can view the blackboard structure as a tree formed by the hierarchy of its nested levels. The length of its longest branch will be the depth of the blackboard.

Definition 2.10 : Blackboard Depth
The depth of a blackboard system is the number of nested levels it can have.

The depth of a blackboard is a mesasure of the richness of the relationships between its levels. For example, in BB1 we can define a Control blackboard that is part of the global system blackboard. We can then define a level of the control blackboard called the agenda, and another called control plan. All this could be described in a Hearsay-II type of blackboard. However, the conceptual relationship between all the control levels would be lost.

We will describe the structure and functionality of the global blackboard by using the definitions in Section 2.2, and describing its depth. Besides this structure, it is necessary to express the different operations that can be applied to it.

We organise these operations in two main groups. These are the blackboard modification and the blackboard access operations.

Definition 2.11 : Blackboard Modification Operations
Blackboard modification operations change the state of the blackboard. These operations include the creation of bbels, the modification of the information within a bbel or set of bbels and the deletion of bbels from the blackboard.

Definition 2.12 : Blackboard Access Operations
The blackboard access operations are the ones that only read from the blackboard and do not modify it. This includes the retrieval of information stored within bbels and testing for the existence of particular pieces of information.

Not all systems allow all these operations. For example, some systems do not allow the deletion of information from the blackboard. These systems keep the old versions of the information stored in the blackboard for explanation purposes or in order to allow some sort of error recovery.

The Knowledge Source Activation Module

The knowledge source activation module contains the system code for the evaluation of the trigger conditions of the different knowledge sources.

A given trigger condition is usually represented as a function or procedure that takes the current state of the blackboard and returns true if it identifies a problem that can be solved by its knowledge source. It also produces information about the problem identified and possibly the knowledge source itself. This information will be stored on the agenda. We call this information a KSAR (Knowledge Source Activation record) and we define it as follows:

Definition 2.13 : KSAR
A KSAR is the representation of a potential knowledge source execution produced by a knowledge source trigger condition. The KSAR can hold information about the the problem that the knowledge source will solve, as well as its its intentions, estimated speed or certainty of arriving at a solution.

The knowledge source activation module is usually very closely linked to the blackboard manipulation module, as the evaluation of the trigger conditions is generally done just after the blackboard is modified.

A trigger condition is, basically, an expression that describes the problems that its associated knowledge source is interested in. As the information about these problems is held in the blackboard, the trigger condition will be expressed in terms of the blackboard structure. Some systems express their trigger conditions in terms of blackboard events, usually creation, modification or deletion of bbels, and the area of the blackboard that they are interested in. This area of the blackboard is described in terms of the blackboard structure. This means that the structure of a trigger condition is highly dependent on the structure of the blackboard itself. The blackboard structures that can be specified by the trigger conditions on a particular system determine their granularity.

Definition 2.14 : Granularity
The granularity of the KS trigger conditions of a particular system is given by the smallest blackboard structure that they can specify when describing a problem.

The KS trigger granularity can be of the following types:

Blackboard The trigger conditions have blackboard granularity when they specify that they are to be called only when there is a modification to the blackboard (any modification).

BB Event The KS trigger condition is evaluated when a particular blackboard event occurs. The blackboard events are usually *creation*, *modification* and *deletion* of bbels.

Level The trigger condition is evaluated when there is a modification to a particular level of the blackboard.

BBel The trigger condition can restrict its evaluation to modifications performed to a particular bbel.

Field The trigger condition is only evaluated when there is a modification to a particular field of bbels on a particular level.

We have described the structure of the KS trigger conditions, its granularity, and the structure of the KSARs. Once the KSARs are produced, they are managed by a control structure. In the next section, we will describe this control structure.

The Control Module

In this section we will describe the system control module by describing its control structure and how it is used to decide the order of knowledge source execution.

We divide blackboard systems in two broad groups depending on the availability of control information.

Definition 2.15 : Implicit Control
A blackboard system has implicit control knowledge when this knowledge is stored within a scheduler or group of specialised schedulers. This knowledge is static in the sense that cannot be modified once the system is running.

Definition 2.16 : Explicit Control
A blackboard system has explicit control knowledge when this knowledge is stored in the blackboard. This offers the possibility of changing the current system control knowledge while the problem solving activity is in progress.

Systems such as the original Hearsay-II and DVMT [LC83] have implicit control, as they have all the system control knowledge coded on a number of schedulers. Systems such as BB1 [HR84] and Blondie-I [LVV86], are explicit control systems as the system control knowledge is stored on a global control plan stored in the blackboard and accessible to knowledge sources. The availability of control knowledge in the blackboard allows the definition of knowledge sources dedicated to identifying particular situations, and modify the system control knowledge accordingly.

This section of the framework describes the control structure of systems with explicit or implicit knowledge and the steps taken in order to achieve a decision. The system control structure can be of the following kind:

Complex Scheduler This is the control structure of those systems in which all the control knowledge is coded in one big scheduler.

Collection of Schedulers This is the usual configuration of implicit control systems. The control knowledge is separated in several control modules each dedicated to take particular control decisions.

Control Plan This is the usual configuration for explicit control systems. The system has a scheduler that decides the execution of the knowledge sources on the basis of control knowledge stored in the blackboard. The set of all public control knowledge is called the *Control Plan*. This control plan is created and updated by a set of control knowledge sources.

The Knowledge Source Execution Module

In this section, we will describe the knowledge source execution module. This is usually the simplest module, as it only executes the body of the knowledge source in a more or less straightforward way.

The body of the knowledge source will receive information about the problem to be solve through the KSAR generated in the triggering process. It then solves it and contributes in the problem solving process by placing the solution in the blackboard using the functions of the blackboard manipulation module.

The actions taken by a knowledge source can be represented in a variety of ways. They can be a collection of production rules, a set of blackboard modification calls or a particular language constructs such as LISP functions, PROLOG predicates or C/C++ procedures. Some systems describe the knowledge sources in a specially tailored language. Their execution module include an interpreter for that language.

2.3.3 Service Modules

Besides the four basic modules, we need to identify and describe all other system modules that may be contained in a particular blackboard system implementation.

These modules appear as response for particular needs. This can be the need to provide a particular interface to the end user (User Interface Module). A distributed system generally has the need to communicate to other systems (Communication Module). There may also be the need to access a large database (Database Manipulation Module).

These modules provide a particular service to one or more of the basic blackboard

modules. Sometimes, the functions of some of these service modules can be completely absorbed into a particular basic module. For example, a given blackboard system may have a user interface knowledge source that handles all user interface.

For each service module, we will describe its function, to which basic modules it is attached, and any relevant implementation issues. A description of how it fits within the global control cycle should be included in case it plays an important part on the problem solving activity. An example of such a module could be the communications module of a distributed system such as DVMT [LC83], and Cassandra [Cra89].

2.4 Blackboard System Performance

In the previous section, we concentrated on the structure of blackboard systems and how they work. This will serve as a basis of describing individual blackboard system's performance.

The performance of a system is usually measured by analysing its behaviour when realising each of its defined tasks. This performance can be seen from two different points of view. The first deals with how well the system achieves its objectives when performing a given task. We call this the *qualitative performance*. The second deals with how fast does the system perform its functions. We call this the *quantitative performance*.

The qualitative performance has to do, mainly, with how well the different system, or module, tasks are done. A description of how those tasks are made is given in the previous section of the framework.

This section of the framework concentrates on the quantitative performance of blackboard systems. This performance is measured by quantifying how fast the system performs its tasks. This will be done by measuring the speed with which the different modules of the system perform their specific tasks and how they influence the global system performance.

When comparing blackboard systems, most of the time we are dealing with systems solving widely different problem domains. Usually their only common characteristic is the fact that they are implemented using the blackboard architecture.

A particular blackboard system can be seen as formed by two implementations. The first one is the implementation of a solution of the problem domain using the blackboard architecture. We will call this the *problem implementation*. The second

one is the implementation of the general purpose blackboard architecture itself, usually tuned to the problem implementation. We will call this second one the *system implementation*.

We can only do meaningful comparisons between the system implementations of different blackboard systems, as their problem implementations may have no characteristics in common. The framework concentrates on measuring the performance of the system implementation, trying to be as independent from the particular problem implementation as possible.

Having measurements on the speed of a system is very important all by itself. They can be used to detect possible problems in particular modules or in the interaction between modules. However, we consider that a large part of their value resides in the possibility of performing comparisons between systems. They can be used as a basis to evaluate the improvements between different versions of a system, or they could be used as a decision factor when choosing one blackboard system over another, or in order to decide for particular implementation approaches.

Another consideration that we should take into account is the fact that there are three widely different blackboard architectures, the sequential, parallel and distributed ones. We need a set of measurements that allow the comparison of systems from any of the three types.

The framework introduces a number of measurements that attempt to provide a basis of judgement for comparing blackboard system performance, being as independent as possible from the particular problem implementation. These measurements are the same for sequential, parallel and distributed systems.

The performance evaluation is based on the measurement of the time spent on each system module function, as well as on the number of times those functions were evaluated. These are the building blocks on which all other measurements are based. We divide the performance measurements into four main groups. The first gives application specific information. The second group presents the performance of each system module individually. The third one shows the influence of each module on the overall system performance. The final group gives global performance information.

2.4.1 Building Blocks

All the performance measurements that we will introduce in the following sections can be calculated from a number of precise statistics. These statistics are the building blocks with which we will construct more complex measurements.

The basic measurement that we will need is the time it takes to evaluate each module function. This, together with the number of times each function is evaluated, will form the basis of all performance measurements.

In general, all blackboard systems will have a number n of modules: M_1, ..., M_n. Module M_i, will have a number m_i of functions: $f^1_{M_i}$, ..., $f^{m_i}_{M_i}$. The number of calls to function $f^j_{M_i}$ will be called $Nf^j_{M_i}$, and its total execution time will be $Tf^j_{M_i}$

In order to describe a blackboard system within the framework, we need to calculate the total time spent on each of the different module's functions.

The problem we encounter when calculating the execution time for a particular function, is that a function can be evaluated within another function, usually from different modules. We need to separate the time accounting of one function from the other. The way we solve this problem depends on the facilities of the computer system in which the blackboard system runs.

On some UNIX systems, we can define a timer for each one of the times we want to measure. When we enter a given function, we stop any active timer and activate the one belonging to that function. The system updates automatically this timer. When we exit the function we stop the function's timer and reactivate the one we stopped before. This insures that only the time spent on a function is accredited to its timer.

This behaviour can also be simulated on other computer systems. We have a set of global variables storing the total time for each function. We also call these variables' timers. The difference between these timers and the previous ones is that they are not updated automatically by the system. When we enter a function and there is an active timer, we calculate the time that has passed between its activation and the present moment and we add it to it. We then deactivate the timer and activate the new one. When we exit the function we calculate the time it took from the last activation of the timer to the present moment and accumulate it in the timer, and, finally, we deactivate it and activate the old one.

2.4.2 Module Use

Before we introduce more specific measurements, we need some from of characterising the different blackboard system applications. When we analyse a particular blackboard system architecture, we will always be solving some sort of domain problem. The responses we will get from the system will always be conditioned to the solution to the domain problem.

In this section we will characterise the different blackboard system implementations. Note that we can implement the same blackboard system on a variety of shells with different architectures and features.

We will characterise the blackboard system implementations on the basis of their system module usage. For each of the system modules, including service modules, we will show the number of times any of the modules' functions were evaluated.

Definition 2.17 : Quantitative Module Usage

$$N_{M_i} = \sum_{j=1}^{m_i} Nf_{M_i}^{j}$$

In order to show the system's time distribution, we will show the proportion of the global execution time spent on each module.

Definition 2.18 : Global Module Proportion

$$G_{M_i} = \frac{\sum_{j=1}^{m_i} Tf_{M_i}^{j}}{\sum_{k=1}^{n} \sum_{j=1}^{m_k} Tf_{M_k}^{j}}$$

2.4.3 Module Performance

For any given module we would like to measure its speed performance. This performance is given by the time it takes to accomplish each of its functions.

Within the framework, we will follow a common approach when measuring a given module's performance. We will provide a short description of the module's general function, and of each of its functions. We will then include performance measurements for each function, and for the module as a whole.

For each function, we provide the average function time. This is calculated by computing the total time spent evaluating the function and dividing it by the number of times it has been performed.

Definition 2.19 : Average Module Function Time

$$Af_{M_i}^{j} = \frac{Tf_{M_i}^{j}}{Nf_{M_i}^{j}}$$

We also provide the percentage of use of each function within the module. This gives a measure of utilisation that is useful for finding out the influence of each function on the overall module's performance. This percentage takes the total number of module function calls as 100%.

Definition 2.20 : Percentage of Use

$$Pf^j_{M_i} = \frac{Nf^j_{M_i}}{\sum_{k=1}^{m_i} Nf^k_{M_i}}$$

We then need a measure that indicates the general module performance. A given module can perform very well in one function and poorly in another function. We would like to have a measure that combines the different functions' performance and gives an idea of the general module's behaviour.

This can be done in two basic ways. We will call them the *weighted performance* and the *average performance*. The weighted performance is based on giving each function a different weight according with their use within the system. This is, add up all the total function times and divide it by the number of times that all the functions have been called.

Definition 2.21 : Weighted Module Performance

$$WM_i = \frac{\sum_{j=1}^{m_i} Tf^j_{M_i}}{\sum_{j=1}^{m_i} Nf^j_{M_i}}$$

The average performance is based on giving each one of the module's functions equal weight and simply average all the function performances. This is, add up all the functions average times and divide them by the number of functions.

Definition 2.22 : Average Module Performance

$$AM_i = \frac{\sum_{j=1}^{m_i} Af^j_{M_i}}{m_i}$$

The weighted performance has the problem that it is highly dependent on the particular problem implementation as it depends on the use given by the system to each one of the module's functions. The module performance of the same blackboard architecture under two different problem domains may give widely different results. The average performance is relatively independent of the problem domain as it does not depend as much on the number of times each function is performed.

We will take the average module performance as the general module performance measure, which we will call *Average Module Time*. We will, however, retain the information about function usage proportion by giving the percentage of use of each function.

We will also include the total module execution time, as it will be needed when calculating system wide performance measurements. It will be computed by adding up all the functions' total times.

Definition 2.23 : Total Module Execution Time

$$TM_i = \sum_{j=1}^{m_i} Tf_{M_i}^j$$

As an example, we will define the corresponding measurements for the blackboard manipulation module. This should be defined for all basic modules, and any service modules in the blackboard system.

Blackboard Manipulation Module

This module contains all the procedures that create, modify, retrieve and delete blackboard elements, together with all its support procedures.

The blackboard manipulation module performs the following functions.

Create BBel:

It creates and initialises one bbel.

Read BBel:

It performs the retrieval of information stored in one bbel. This includes bbel searching and testing for their existence. This function makes use of the blackboard without changing its status.

Modify BBel:

It modifies the information stored on an existing bbel.

Delete BBel:

It removes one bbel present in the blackboard.

We now define the average function times:

The average creation time is the time it took to create all the bbels needed in a given run of the system, divided by the number of those bbels. When a procedure that creates a bbel is called, its execution time is added to the total accumulated time for bbel creations, and the count of those creations is incremented by one. When there is a procedure that creates more than bbel at a time, we do as before, we add its execution time to the total creation time and increment the bbel creation counter by the number of bbels created.

Definition 2.24 : Average BBel Creation Time
Let TC_M *be the total blackboard creation time, and* NC_M *the number of bbels created by the system. The average bbel creation time* (AC_M) *is defined as:*

$$AC_M = \frac{TC_M}{NC_M}$$

The average read time is the time it took to retrieve all the information requested by the system, divided by the number of bbels that were consulted.

Definition 2.25 : Average Blackboard Read Time
Let TR_M *be the total blackboard read time, and* NR_M *the number of individual blackboard reads performed by the system. We define the average blackboard read time* (AR_M) *as:*

$$AR_M = \frac{TR_M}{NR_M}$$

The average modification time is calculated by dividing the time it took to perform all the bbel modifications by the number of bbels modified.

Definition 2.26 : Average Blackboard Modification Time
Let TM_M *be the total blackboard modification time, and* NM_M *the number of blackboard modifications performed by the system. We define the average blackboard modification time* (AM_M) *as:*

$$AM_M = \frac{TM_M}{NM_M}$$

The average deletion time is the total time needed to delete all the bbels needed by the system, divided by the number of bbels.

Definition 2.27 : Average BBel Deletion Time
Let TD_M be the total bbel deletion time, and ND_M the number of bbels deleted by the system. The average blackboard deletion time (AD_M) is defined as:

$$AD_M = \frac{TD_M}{ND_M}$$

The function's percentage of use is defined by:

Definition 2.28 : Percentage of Use
Let N_M be the number of times any of the module's functions were evaluated:

$$N_M = NC_M + NR_M + NM_M + ND_M$$

The bbel creation, blackboard read, blackboard modification and deletion percentage of use (PC_M, PR_M, PM_M and PD_M respectively) are defined by:

$$PC_M = \frac{NC_M}{N_M},\ PR_M = \frac{NR_M}{N_M}$$

$$PM_M = \frac{NM_M}{N_M},\ PD_M = \frac{ND_M}{N_M}$$

We calculate the average module time by adding all the function average times and dividing it by the number of functions of the module.

Definition 2.29 : Average Blackboard Manipulation Module Time
The average blackboard manipulation module time (AM) is defined by:

$$AM = \frac{AC_M + AR_M + AM_M + AD_M}{4}$$

Finally, the total blackboard manipulation module time is given by:

Definition 2.30 : Total Blackboard Manipulation Module Time
We define the total blackboard manipulation time (TM) as:

$$TM = TC_M + TR_M + TM_M + TD_M$$

We can repeat these steps to calculate the measurements for the other basic modules, and for the support modules.

2.4.4 Global Module Influence

We will now provide a set of measurements that highlight the proportion of time spent by the system in the evaluation of each one of its modules.

The first measurement is the total time taken by the system in solving a problem.

Definition 2.31 : Total System Run Time
We define the total system run time (T_{Run}) as the time it takes the system to reach a solution of the problem domain and stop.

If we assume that all the modules are evaluated in a sequential machine, we could assume that the total run time of the system is the sum of all the modules total times. We call this time the system sequential run time.

Definition 2.32 : Sequential Run Time
The system theoretical sequential run time (T_{Seq}) is the time taken by all the modules of the system as if they all are performed one after the other. Let TSp be the time taken by the evaluation of all the system support modules. TM, TT, TS, and TB are the total blackboard manipulation time, the total triggering time, the total scheduler time, and the total body execution time respectively. We calculate the theoretical sequential run time as:

$$T_{Seq} = TM + TT + TS + TB + TSp$$

In a sequential system, this time should be equal to the total system run time. This, however, is not necessarily true. For a sequential system, the T_{Seq} is usually smaller than the T_{Run}. This is because generally not all the execution of the system falls necessarily within the execution of a particular system module. This extra time could be, for example, due to module switching or to user interaction. In general it is due to procedures that do not belong to any of the defined modules. If this difference is too big, we could think of analysing the system and identifying, and defining, more service modules. We call this time spent outside the defined modules, the total idle time.

Definition 2.33 : Total Unaccounted Time
The total unaccounted time (TUa) is the time spent by the system when it is not evaluating any of the defined modules.

On a sequential system, the total unaccounted time is calculated by performing the difference between T_{Run} and T_{Seq}. On distributed systems we can treat each node

as a sequential or parallel blackboard system and calculate its total system run time and its own theoretical sequential run time. The total unaccounted time, then, can be calculated as the addition of all the node's TUa's.

On a parallel system the calculation is more complicated. A parallel system runs on a number of processors, each one of which is evaluating a number of system modules. Each processor is usually dedicated to the evaluation of a particular part of a system module. For example, there can be a parallel system in which a processor only evaluates a particular knowledge source and nothing else.

For parallel systems we calculate the total unaccounted time as follows:

We deal with each of the system processors separately. For each of them we calculate the total module times as if we were in a sequential system. We call these times TM_i, TT_i, TS_i, TB_i, TSp_i, where i is the particular processor. We call n the number of processors on which the system runs. Note that for some processors some of the total module times be zero, as they do not perform any function of those modules.

We calculate the total unaccounted time for each processor as follows:

$$TUa_i = T_{Run\,i} - (TM_i + TT_i + TS_i + TB_i + TSp_i)$$

Where $T_{Run\,i}$ is the total run time of the system on the processor i. This time is always smaller or equal to T_{Run}.

Let p be the number of processors used by the system. The total system unaccounted time is then calculated by:

$$TUa = \sum_{i=1}^{p} TUa_i$$

Note that the total unaccounted time of sequential and distributed systems can be calculated by exactly the same method. In fact sequential systems are the special case in which the number of processors equals one. On sequential and distributed systems, all of the total module times have non-zero values.

Besides the TUa measure, we need to show the proportion of the unaccounted time with respect to the total system run time. On sequential systems, we would simply divide TUa by T_{Run}. However this is not appropriate for parallel and distributed systems. We would expect, for concurrent systems, the unaccounted time to increase as the result of synchronisation-induced delays. Note that for these systems the T_{Run} is measured only once, but TUa is accumulated for all processors. We are, basically, multiplying the unaccounted time by the number of processors.

In order to avoid this, and to produce a more sensible measure, we calculate the proportion of unaccounted time with respect to the total sequential run time, including the unaccounted time. This can be easily calculated by multiplying T_{Run} by the number of processors (p). Again, the sequential system is a special case, where $p = 1$.

Definition 2.34 : Proportion of Unaccounted Time
Let p be the number of processors used by the system. The proportion of unaccounted time is defined by:

$$P_{U_a} = \frac{TU_a}{p * T_{Run}}$$

The framework includes a set of measurements that show the influence of each of the system modules in a form that is relatively independent from the domain application. This measurement shows the proportion between each module's AM_i (average module performance).

Definition 2.35 : Module Average Influence

$$I_{M_i} = \frac{AM_i}{\sum_{k=1}^{n} AM_k}$$

These measures show the most expensive modules from the performance point of view. The module average influences can be presented on a graph that helps visualisation of proportions, such as a pie chart. Note that we are not using the total module times to show each module's influence. This is already presented in the global module proportion (Definition 2.18), and has the problem that is very application dependent as a module's influence increases the more it is used. The most used module for a particular application is not necessarily the most expensive one for other applications. By using the average module times we achieve a degree of independence from the application domain. However, if our application uses heavily one of the more expensive modules, we would either modify the implementation, or direct optimisation efforts to that module.

2.4.5 System Performance

It is certainly useful to have a system performance measure that shows the global behaviour of the system. This measure should show how fast the system performs, including gains that could be achieved by performing tasks concurrently by distributed and parallel systems.

The purpose of this framework is to highlight generic blackboard characteristics and not the ones specific to particular problem domains or user implementations such as control strategies or solutions to domain problems.

The global system performance measure should depend as little as possible on the system's problem domain, and it should be relatively independent from user implementations. It should concentrate on the speed of the underlying architecture.

For example, the total system run time is not a good measure as it depends highly on the particular system problem domain, and even within the same problem domain, it depends on how the user implemented, say, the system control strategy. It gives very different results even from two executions of the same system. The basic problem with this measure is that it depends directly on the number of control cycles done by the system within a run. This makes it highly sensitive to how the user implemented its control strategy and how fast do the knowledge sources reach a solution. In short, the total system run time is very problem specific.

The blackboard problem solving activity is encompassed by the control cycle of, scheduler decision, body execution, blackboard modification and trigger evaluation, together with the support modules' execution and unaccounted time.

The framework proposes a performance measure that shows how fast the system performs the above cycle. This is the average cycle time.

Definition 2.36 : Average Cycle Time (A_{C_y})
The average cycle time is the time it takes the system, on average, to evaluate one blackboard control cycle.

Note that this measure is independent of the number of cycles and thus it is relatively independent of the user control implementation. It is independent of how good the user control strategy is. However, it is dependent on the problem domain as it includes the knowledge source body time, and the scheduler time. We consider that to disregard these times would mean to lose valuable information about the blackboard system's behaviour. In order to be able to compare the performance of two blackboard systems, we will need to test them with the same problem domain.

The average control cycle can be calculated by dividing the total run time by the number of cycles performed by the system. This, however, assumes that we can calculate the number of control cycles for any system.

In a sequential system we can calculate it easily, as the scheduler decision, body execution and the other steps are done all in one place (one processor). In a distributed system we can calculate the number of control cycles in each node of the architecture

and add them all together.

For a parallel system, however, this computation is more complicated as every processor can be evaluating a different part of different modules. It can be difficult to keep track of the control cycle.

We consider that the central step within the control cycle is to evaluate a knowledge source body. The other steps are equally important, but they are all done in preparation to, or on support of, that evaluation. Both for sequential systems and distributed systems, the number of KS bodies evaluated by the system equal the number of cycles performed by the system.

We can, in fact, use the number of KS bodies evaluated as the number of cycles of the system. This number is equally easy to calculate on sequential, distributed or parallel systems.

Let NB_B be the number of knowledge source body executions. The average cycle time can then be calculated by:

$$A_{Cy} = \frac{T_{Run}}{NB_B}$$

Note that for distributed and parallel systems the A_{Cy} will be smaller depending on the amount of concurrency achieved by the system. This means that for the same run time there are more cycles being done concurrently.

The framework also includes the calculation of the system speedup, with respect to the sequential system. Note that T_{Seq} is an estimate of the time that would take to run the system on one processor, and T_{Run} is the total run time. If the system considered is a distributed or parallel system, T_{Run} is the time taken to run the system on several processors. We can estimate the overall system speed up as follows.

Definition 2.37 : System Speed Up

The system speedup (SU) is the proportion between the theoretical sequential run time of the system, and its real run time.

$$SU = \frac{T_{Seq}}{T_{Run}}$$

For different versions of the same system, let T_{Run_i} be the total run time of a previous version and T_{Run_j} the total run time of the new version ($i < j$), the speedup between them is:

$$SU = \frac{T_{Run_i}}{T_{Run_j}}$$

2.4.6 General Performance Table

All the performance measurements introduced in the previous subsections can be presented using a summarisation table. For example, Table 2.1, shows the different performance measures for the BB1 shell, on the Jigsaw-Puzzle solver application. This will be explained in greater detail in the next chapter.

The table includes a bar chart showing, at a glance, the number of calls to each system module, and to its right, a pie chart showing the proportion of time usage for each module. We also present a second pie chart showing the module proportion, also shown in numerical form to its left. The times presented in the following table are measured in seconds.

JSP Application: Module Use

Module	Number of calls		Time Use		Number of Calls / Time Use
BB. Manipulation Module	N_M	891	G_M	44.53%	
KS. Activation Module	N_T	181	G_T	15.57%	
Scheduler Module	N_S	180	G_S	34.09%	
KS. Execution Module	N_B	35	G_B	5.82%	

Module Performance

Module	Function	Average Time		Percentage of Use		Module Times Average		Module Times Total	
BB. Manipulation Module						AM	0.16	TM	103.84
	BBel Create	AC_M	0.09	PC_M	73.00%				
	BBel Access	AR_M	0.00	PR_M	0.00%				
	BBel Modify	AM_M	0.57	PM_M	27.00%				
	BBel Destroy	AD_M	0.00	PD_M	0.00%				
KS. Activation Module						AT	0.20	TT	36.31
	KS. Trigger	AT_T	0.20	PT_T	100.00%				
Scheduler Module						AS	0.44	TS	79.49
	Choose KS.	AS_S	0.44	PS_S	100.00%				
KS. Execution Module						AB	0.38	TB	13.57
	Evaluate KS.	AB_B	0.38	PB_B	100.00%				

Module Influence

Total Sequential Run Time	T_{Seq}	233.21
Total Unaccounted Time	TUa	82.48
Total System Run Time	T_{Run}	315.69
Unaccounted Time Prop.	P_{Ua}	26.13%
Module Average Influence		
BB. Manipulation Module	I_M	10.17%
KS. Activation Module	I_T	17.50%
Scheduler Module	I_S	38.52%
KS. Execution Module	I_B	33.82%
Service Modules	I_{Sp}	0.00%

System Performance

Average Cycle Time	A_{Cy}	9.02
System Speedup	SU	0.74

Table 2.1: BB1 General Performance table

Note that the first pie chart shows the proportion between the total module times shown numerically in the module performance table. This pie char allows us to find the most important modules for the given application, either because they are expensive, or they are used extensibly. The second pie chart, however, shows the proportion between the average module times. This allows to identify the most expensive modules, regardless of the number of times they are used.

2.5 Summary

In the present chapter, we introduced a new performance framework for the study of blackboard systems. This is one of the contributions of this thesis.

This framework may be used to study the behaviour of blackboard systems as a whole, or to investigate the effects of specific design decisions on the overall system performance. This will help direct the implementation of applications, as well as system optimisation efforts. It contributes in this way to improve the system's performance. The framework presented here complements and completes the work of other authors.

In the next chapter we will show how the framework can be used to study the performance of existing blackboard systems. We will present some case studies of blackboard systems present in the literature, and for which we could obtain source code. This made possible the inclusion of profiling procedures for performance measurement. We will describe them using the first section of the present framework, and we will study their performance using the second section.

Chapter 3

Case Studies

3.1 Introduction

In this chapter, our aim is to study several blackboard systems present in the literature with the help of the framework introduced in the previous chapter. Having enough information about the different systems will allow us to draw conclusions about their design and implementation decisions. This will highlight the usefulness of having such a common description.

We will study Stanford's BB1 [HR84], Edinburgh's EPBS [JMR86], and Dr. Neher Labs' Blondie–I and Blondie–III [LVV86, FvLV88]. For these systems, we were able to obtain source code, which was needed for the inclusion of the various performance measurement routines that calculate the data presented in the performance framework.

Each system used a different implementation language and, in the case of BB1, a different computer platform. The framework, however allows us to compare them in terms of their functionality and relative performance. With the use of the blackboard framework, we can identify their structural similarities and we can use the performance measurements to identify the relative impact of the different design and implementation decisions.

All the systems analysed are blackboard shells. This means that we can run the system on very different application domains. In the performance framework, there are some measurements that are used to characterise the application itself. These allow us to identify application classes. If we analyse two blackboard systems with similar applications, we would have more basis for comparison between them.

As we were able to use the blackboard shells under analysis, we decided to implement the same blackboard application in all of them. This is the jigsaw-puzzle solver application, an extensible test blackboard system.

3.2 The Jigsaw Puzzle (JSP) Solver

The task of the jigsaw-puzzle (JSP) system is to solve a jigsaw-puzzle. This task is achieved by a number of knowledge sources that represent several players. Each knowledge source will have a number of pieces assigned to it. At each cycle of the game, each knowledge source tries to place one piece into a board. Each piece's sides must fit in the sides of the adjacent pieces on the board.

The size of the jigsaw may represent the complexity of the overall system. The task of solving a bigger jigsaw is considerably more complex than the solving of a smaller one. The number of pieces that each knowledge source holds may be seen as

the amount of knowledge that they have.

The *pieces* are represented by a structure (or object) with four sides. On each side, the piece has a number identifying its shape. The pieces can be rotated and placed in any orientation. The board is made out of a number of piece *positions*. Each position also has four sides. Initially the position's sides are uninstantiated, except for those on the side of the board.

The JSP blackboard will have three levels, one for holding the pieces, one for the board positions, and a third one specifying which pieces correspond to each player.

The JSP system will have a number of knowledge sources representing the players. There can be any number of players collaborating to solve the jigsaw. Initially, each player will be provided of a number of pieces. The task of each player is identical. They look at the board to find out if any of their pieces match any position. When a piece is to be placed on a given position, the player has to modify not only the position, but also its four adjacent positions in order to instantiate their side shapes. In order to restrict the number of places where a piece can match, we force the knowledge sources to look for positions the have at least two sides instantiated. This avoids having everybody placing pieces in the middle board positions at random.

As the pieces can be rotated, and the number of side shapes is limited, it is possible for several pieces to legally match a given position. This means that the players can place pieces in the wrong places. We need an extra knowledge source that identifies when this happens, and corrects it. We will call it the *Piece Misplaced* knowledge source. Its job is, basically, to backtrack from decisions taken by the players. It will take out any possible offending piece, and will return it to its player. It will also temporarily mark the position so that the same piece cannot be placed back to it.

The piece-misplaced knowledge source will look for legal positions (two of their sides are instantiated) on which there is no free piece that can be placed. There are two ways of looking for this type of condition. The first one is the exhaustive approach. The knowledge source must have access to all the free pieces, and will check if there is no piece that matches a given position. In that moment, the knowledge source activates and will try to correct the error. This approach, however, is highly redundant, as the job of matching pieces against positions is already being done by the players.

The second approach avoids the duplication of effort and consists of assuming that there is always a problem, and that there is some entity such as a controller or scheduler that ultimately decides if there is really a problem. We can then define this knowledge source with the trivial trigger condition (always true). This means that the confidence of having really identified a problem is very low, and thus, its priority must

be also very low. The piece-misplaced knowledge source is allowed to be evaluated only when no other knowledge source has been activated for a given position.

Note that, in order to be able to implement the non-redundant approach, the blackboard system needs to have a scheduler, or some sort of knowledge source execution control.

Once a position that cannot be filled is identified, the piece-misplaced knowledge source will remove from the board the last piece placed in one of its adjacent positions. This piece is assumed to be the wrong choice.

Besides the player, and the piece-misplaced, there are two more knowledge sources. These are the initialisation, and termination knowledge sources. The first initialises the state of the problem. It creates the board positions and the jigsaw pieces. It also assigns the pieces to the players. The termination knowledge source detects when all pieces have been placed, and stops the system.

The JSP can be configured for any size of (rectangular) jigsaw-puzzle, any number of player knowledge sources, and any distribution of the pieces among these knowledge sources.

3.3 Case 1: BB1

BB1 is an expert system shell that uses a variation of the Hearsay-II blackboard architecture to provide a uniform mechanism for reasoning about problems and problem solving actions. BB1 considers the scheduling problem to be complex enough to be suitable to be solved using the blackboard model.

BB1 was implemented in LISP, and was tested on a MicroExplorer, a LISP machine from Texas Instruments coupled to an Apple Macintosh.

3.3.1 Blackboard System Structure

General Module Structure

Module Description

According to [AMM87], we can identify the four basic blackboard modules:

- The *knowledge source execution module* (KSE), formed by the actions evaluator in the BB1 *interpreter*.

- The *blackboard manipulation module* (BBM), formed by all the blackboard manipulation routines within the BB1 interpreter.
- The *knowledge source activation module* (KSA), made up of the trigger manipulation and evaluation routines of the BB1 module *agenda-manager.*
- The *scheduler module* (SCH), formed by the agenda rating functions of the agenda-manager and the BB1 *scheduler.*

As a research tool, BB1 provides a number of user interface facilities for tracing and examining all the system components. In each cycle, the system shows which knowledge source it is evaluating, and any changes to the blackboard, corresponding to levels being displayed, are updated.

The operations needed for providing this user interface are spread among all the other modules' functions, and cannot be clearly identified. They cannot be easily turned off either. The only way we could find for measuring the time spent in user interaction was to measure the time in which the system was not performing any of the other modules' functions. This is equivalent to actually measure the system's unaccounted time.

We decided not to define any support modules, as their functions could not be clearly identified and defined. However, we must keep in mind that the system's unaccounted time is spent, mostly, in user interaction.

Control Cycle

The basic module evaluation sequence within the BB1 control cycle is as follows. The detailed description of each step will be given when describing the basic modules.

1. A knowledge source body is executed (KSE). This execution makes some modifications to the blackboard (BBM).
2. The knowledge source trigger conditions are evaluated and the list of triggered knowledge sources is stored in an agenda (KSA).
3. A triggered knowledge source is chosen to be performed (SCH).

Figure 3.1 gives a graphic representation of the basic modules and their interactions within the BB1 control cycle.

Type of Architecture

The BB1 blackboard architecture is designed as a *sequential* architecture.

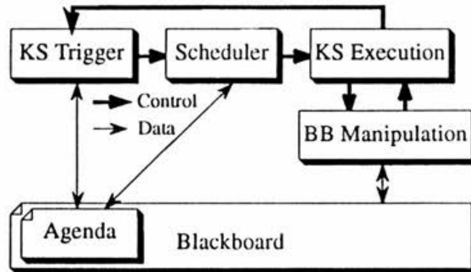

Figure 3.1. The BB1 Control Cycle

Basic Modules

The Blackboard Manipulation Module

The BB1 blackboard has a depth of two. This is, the global blackboard is formed by several sub-blackboards. These smaller blackboards can be subdivided into several levels, each of which will hold objects of a given type.

The BB1 bbels belong to one and only one level. They are LISP structures that have an unique name within its level, and any number of attribute–value pairs. They can also have any number of links (references) to other objects that symbolise relations between objects. The BB1 links allow the inheritance of bbel attributes.

The BB1 blackboard manipulation module provides a set of functions and procedures for accessing and modifying bbels. The access functions can be used in the different parts of the system, but they are usually utilised in the body of the knowledge sources. They allow the retrieval of a given value of an object, or the objects linked to an object. There are also several functions for finding objects within the several blackboard levels.

The Knowledge Source Activation Module

The BB1 knowledge sources are stored as objects on a given system blackboard. The BB1 knowledge sources consist of a *trigger condition*, a *precondition*, an *obviation condition*, a set of *actions*, *cost*, and *reliability*.

The BB1 knowledge source activation is performed in two stages. First, the trigger conditions are evaluated whenever there is a modification of the blackboard that concerns the particular knowledge source. If the conditions are true, a KSAR is created and placed on the "triggered" agenda. Second, the preconditions are evaluated. If the preconditions are true, the KSAR passes to the "executable" agenda and become

eligible to be chosen by the scheduler to be evaluated the next cycle. However, if any precondition is false the KSAR stays on the "triggered" agenda.

The BB1 trigger condition granularity is *event*. This is, the knowledge sources trigger when a blackboard event occurs. BB1 produces events when creating, modifying, and deleting bbels.

The Control Module

The BB1 control module also works on a two step fashion. It first evaluates the obviation conditions of all KSAR's. If this condition is true for a particular KSAR, this KSAR is discarded from the system. The second step is to rate all remaining KSAR's on the "executable" agenda against the current control plan.

The BB1 control plan is defined as a set of bbels in a particular blackboard called "Control Blackboard". This control blackboard is subdivided in three levels, namely the *Strategy* level, the *Focus* level and the *Heuristic* level. Each one of these levels will hold objects that, together, form the global control plan. These objects will typically use the estimated cost and reliability of the knowledge sources.

A BB1 strategy is a partial control plan represented by a set of sub-strategies and foci, that apply in a certain situation. These strategies are usually created and activated by special knowledge sources called *Control Knowledge Sources.*

A BB1 focus is an object that specifies a set of heuristics that will be used to rate all executable KSAR's. The BB1 heuristics contain specific KSAR rating functions.

When the scheduler is rating all the executable KSAR's, it looks at every focus in the current control plan, and evaluates all the heuristics they specify for each KSAR. The results from these evaluations are composed (simply added) using weights described on each focus. For each KSAR this produces a rating number for each focus. These numbers are all added up together to form the final KSAR rating. The scheduler finally chooses the KSAR with the highest rating.

The Knowledge Source Execution Module

The knowledge source execution module interprets the set of rules stored in its body (actions). The result of those rules is to perform modifications to the blackboard. These modifications are performed using the functions provided by the blackboard modification module.

3.3.2 Blackboard System Performance

For the evaluation of BB1, we used the jigsaw-puzzle solver application that solves a 3 by 4 jigsaw-puzzle. The pieces are distributed equally among four player knowledge

sources. In total the system has six knowledge sources. The initialisation and termination knowledge sources are implemented as one. The following table shows the module usage of the JSP application. All time measurements are expressed in seconds.

JSP (3x4) Application: Module Use

Module	Number of calls	Time Use	
BB. Manipulation Module	N_M 891	G_M 44.53%	
KS. Activation Module	N_T 181	G_T 15.57%	
Scheduler Module	N_S 180	G_S 34.09%	
KS. Execution Module	N_B 35	G_B 5.82%	

Number of Calls   Time Use

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times Average	Module Times Total
BB. Manipulation Module				AM 0.16	TM 103.84
	BBel Create	AM_M 0.57	PM_M 27.00%		
	BBel Access	AR_M 0.00	PR_M 0.00%		
	BBel Modify	AC_M 0.09	PC_M 73.00%		
	BBel Destroy	AD_M 0.00	PD_M 0.00%		
KS. Activation Module				AT 0.20	TT 36.31
	KS. Trigger	AT_T 0.20	PT_T 100.00%		
Scheduler Module				AS 0.44	TS 79.49
	Choose KS.	AS_S 0.44	PS_S 100.00%		
KS. Execution Module				AB 0.38	TB 13.57
	Evaluate KS.	AB_B 0.38	PB_B 100.00%		

Module Influence

Total Sequential Run Time	T_{Seq}	233.21
Total Unaccounted Time	TUa	82.48
Total System Run Time	T_{Run}	315.69
Unaccounted Time Prop.	P_{Ua}	26.13%
Module Average Influence		
BB. Manipulation Module	I_M	10.17%
KS. Activation Module	I_T	17.50%
Scheduler Module	I_S	38.52%
KS. Execution Module	I_B	33.82%
Service Modules	I_{Sp}	0.00%

System Performance

Average Cycle Time	A_{Cy}	9.02
System Speedup	SU	0.74

Table 3.1: BB1 General Performance table

Note that the percentage of use of each module function refers to the proportion of times the given function is called, within the total number of calls to its module. The average module time is calculated by averaging the average times of the module's functions. For a more complete description, please look at the previous chapter

(Section 2.4.3, Definition 2.21). Note, also, the difference between the two pie charts presented in the performance table. The time use chart is a representation of the total module time column showing the proportion of time spent on each module for the application being considered. The module average influence chart is a representation of the average module time column showing each module's influence in a more application independent fashion. This highlights which modules are more expensive. Note that this chart, combined with the number of calls of each module yields the time use chart.

From the application's module use table, we can see that the JSP system makes heavy use of the blackboard. The jigsaw-puzzle is being solved by placing and removing pieces on the blackboard.

In all blackboard systems, the solution of a problem is achieved by the body of the knowledge sources, and the modifications made by it to the blackboard. These two times represent the time taken by the players to solve the puzzle. All other times serve for player preparation (trigger) and guidance (scheduler). From the time use pie chart, we can see that the body and blackboard times amount half the system's execution time. The other half is spent in triggering and scheduling. We consider this to be excessive. BB1 would benefit from an optimisation of its triggering and scheduling processes.

In the Module performance table, we can see that the system does not access (read) or delete any bbel. In the JSP application, all blackboard accesses are done via the triggering conditions, and both the pieces and board positions are never deleted. We also notice that bbel modification is much more efficient than bbel creation. This, combined with the system's heavy use of blackboard modification functions, show an efficient blackboard module.

We can see in the module proportion pie chart that the triggering and blackboard modules proportion is larger than the body and blackboard modules combined. This reinforces the conclusion that BB1's triggering and scheduling processes should be revised.

We can also see that there is a large amount of unaccounted time (26.13%). As we explained earlier, this is due to large user interface activity, including lengthy screen refreshes every cycle. A more flexible and configurable user interface would be advisable. This would allow the user to specify the type and amount of user interaction needed. For example, when testing the system it is useful to be able to trace each cycle decisions just as it works now. However, when the system is in production this trace is not needed.

3.3.3 Discussion

When working towards a solution of a particular domain problem, any blackboard system must deal with several other problems. The most important of those problems is how to control the activity of the knowledge sources. This is known as the *control problem*.

The BB1 approach considers control as a complex problem suitable to be solved using the blackboard model. BB1 solves both the domain and the control problems in the same environment. This places a heavy load on the blackboard manipulation, and knowledge source activation modules.

It is not completely clear as to what are the advantages of having a two step triggering process. Similar functionality could be achieved by composing both triggers, and preconditions, into one trigger condition. A KSAR only needs to be created and manipulated when both triggers and preconditions are satisfied. A different knowledge source activation module design and implementation may provide a better performance.

The main emphasis of the BB1 system is on its control architecture. As such, we expect that it would spend an appreciable amount of time in the control module. This can certainly be seen in the module time use, and in the module proportion pie charts. However, this is not necessarily bad. It can improve the overall system performance by reducing the amount of cycles (knowledge source executions) needed for solving a problem. We consider that the scheduling process should be revised and optimised, however we think that the extra expressiveness introduced on the system is needed. The module optimisation should be done without compromising its functionality.

The BB1 system would benefit from an optimisation of the implementation of each one of its modules. We believe that, in order to improve further, we would need to migrate to a more parallel architecture. The BB1 system, as it stands, would not profit from such a migration, mainly because of its control structure design. The whole BB1 architecture is based on the idea of a single central scheduler. This would impose a strong control bottleneck that would restrict the efficient use of parallelism.

3.4 Case 2: EPBS

EPBS [JMR86, JM86] is a PROLOG-based expert system shell that uses a variation of the Hearsay-II blackboard architecture. It was initially developed as a tool in the domain of user modelling. However, it is designed to be general enough to be used in other problem domains.

3.4.1 Blackboard System Structure

General Module Structure

Module Description

Using the framework definitions, we can identify the four basic blackboard modules [JMR86] are:

- The *knowledge source execution module* (KSE), formed by the set of knowledge sources.
- The *blackboard manipulation module* (BBM). This includes procedures for maintaining the consistency of the blackboard as a whole.
- The *knowledge source activation module* (KSA)
- The *scheduler module* (SCH)

Control Cycle

[JM86] describes the EPBS control cycle broadly as follows:

1. Choose a KSAR from the agenda. This task is performed by the scheduler.
2. Execute that KSAR. This involves the execution of the action corresponding to the knowledge source associated to the chosen KSAR and modifying the blackboard accordingly.
3. Construct a new agenda. The new status of the blackboard may trigger several knowledge sources. Their trigger conditions are revised and new KSAR's are created for those that activate.

The EPBS agenda is stored in the blackboard and thus, the global consistency mechanism applies to the KSAR's stored in it. Figure 3.2 gives a graphic representation of the basic modules and their interactions within the EPBS control cycle.

Type of Architecture

The EPBS blackboard architecture is *sequential.*

Basic Modules

The Blackboard Manipulation Module

The EPBS blackboard does not exist as an explicit data structure, it is a collection of terms within the global PROLOG database that represent the blackboard bbels.

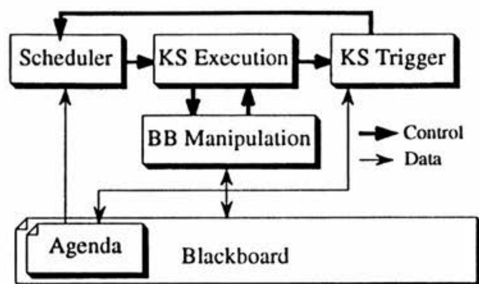

Figure 3.2. The EPBS Control Cycle

The user has total freedom as to the structure of the blackboard. It is possible to define the blackboard as having any amount of nested levels, each one relating to a conceptual division of the problem domain. This feature allows the EPBS blackboard to be defined as having any desired depth.

EPBS provides a built-in consistency checking mechanism. This mechanism is a support-based truth maintenance system.

Every new bbel of the blackboard is assumed to be a consequence of the existence, or non-existence, of other bbels in the blackboard. The old bbels are said to "support" the new bbel. Whenever a new bbel is added to (or deleted from) the blackboard, the consistency of the bbels already in it must be maintained. In particular, all the bbels that are supported by the non-existence (or existence) of the new must be discarded.

There are two basic operations provided by the blackboard manipulation module. These are `add` and `amend`. The `delete` operation is performed indirectly by the blackboard consistency mechanism. The `read` operation is not provided by the system. EPBS assumes that all the read access to the blackboard is performed via the trigger conditions of the knowledge sources. This is needed in order to simplify the way the system keeps track of the supports of the modifications performed by the knowledge source.

The Knowledge Source Activation Module

The EPBS knowledge sources are stored as predicates on the PROLOG database. They have four components that are `Condition`, `Body`, `Effect` and `Est`. The `Condition` represents the knowledge source trigger condition and it is made up a series of PROLOG tests. The tests within a knowledge source trigger condition will be used as the supports of all the modifications made by it. The `Body` and `Effect` are used by the

knowledge source evaluation module, and the `Est` (Estimate) is used by the control module.

The knowledge source activation module is called once all modifications corresponding to a given knowledge source are performed onto the blackboard. The tests are evaluated, then, whenever there is a modification to the level of the blackboard specified in the `Condition`.

According to the framework definitions, the EPBS has *Level* trigger *granularity*.

The Control Module

EPBS control module provides a default *implicit* control scheduler mechanism. This mechanism is based on the `Est` (Estimate) part of the knowledge sources, which is a numeric value by default. The control mechanism provided chooses the KSAR whose knowledge source has the lowest estimate.

This mechanism is implemented using the `rating/2` predicate that takes a knowledge source estimate and computes its rating, and using the `less_rating/2` predicate that compares two ratings. The EPBS system allows the user to redefine these predicates. This makes it possible to have non-numeric estimates and more complex schedulers. Because there is no system provided blackboard read mechanism, there is no clean way to implement *explicit* control mechanisms.

The Knowledge Source Execution Module

The EPBS knowledge sources body is divided in two parts. The first part, or `Body`, is an arbitrary PROLOG goal that is evaluated using the standard `call/1` predicate. The second part is the `Effect` that is composed of a list of additions or modifications to the blackboard that are evaluated calling the blackboard manipulation module functions.

3.4.2 Blackboard System Performance

For the performance evaluation of the EPBS system, we used the jigsaw-puzzle solver application that solves a 3 by 4 jigsaw-puzzle. The pieces are distributed equally among four player knowledge sources. In total, the system has six knowledge sources. The initialisation and termination knowledge sources are implemented as one. The following table shows the module usage of the `JSP` application.

JSP (3x4) Application: Module Use

Module	Number of calls	Time Use
BB. Manipulation Module	N_M 164	G_M 77.29%
KS. Activation Module	N_T 61	G_T 0.14%
Scheduler Module	N_S 13	G_S 22.56%
KS. Execution Module	N_B 13	G_B 0.02%

Number of Calls Time Use

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times Average	Module Times Total
BB. Manipulation Module				AM 35.3040	TM 1836.64
	BBel Create	AC_M 0.0060	PC_M 92.07%		
	BBel Access	AR_M 0.00	PR_M 0.00%		
	BBel Modify	AM_M 141.21	PM_M 7.93%		
	BBel Destroy	AD_M 0.00	PD_M 0.00%		
KS. Activation Module				AT 0.0534	TT 3.26
	KS. Trigger	AT_T 0.0534	PT_T 100.00%		
Scheduler Module				AS 41.2321	TS 536.02
	Choose KS.	AS_S 41.2321	PS_S 100.00%		
KS. Execution Module				AB 0.0347	TB 0.45
	Evaluate KS.	AB_B 0.0347	PB_B 100.00%		

Module Influence

Total Sequential Run Time	T_{Seq}	2376.36
Total Unaccounted Time	TUa	1.0
Total System Run Time	T_{Run}	2377.63
Unaccounted Time Prop.	P_{Ua}	0.04%
Module Average Influence		
BB. Manipulation Module	I_M	46.07%
KS. Activation Module	I_T	0.07%
Scheduler Module	I_S	53.81%
KS. Execution Module	I_B	0.05%
Service Modules	I_{Sp}	0.00%

System Performance

Average Cycle Time	A_{Cy}	182.80
System Speedup	SU	1.00

Table 3.2: EPBS General Performance table

The 2 by 4 jigsaw was the biggest JSP configuration that we could run. For this example, the system took almost forty five minutes to finish.

We can see in the modules' time use pie chart that most of the time is spent in blackboard manipulation, and the rest, almost a quarter, in knowledge source scheduling. These two times seem excessive in comparison to the knowledge source activation and execution modules.

The module performance table shows that the knowledge source trigger, and

execution modules are very efficient. However, the blackboard manipulation module, and specially the scheduler module are very inefficient. Within the blackboard manipulation module, we can see that the bbel modification function implementation is extremely expensive. This function's implementation should be optimised in order to improve the overall system performance.

The EPBS system does not make a heavy use of the bbel modification function. The blackboard consistency mechanism forces the creation of new bbels for each bbel modification. The old copies are kept in order to allow the system to undo the modifications when their supports are invalidated. The amount of objects in the PROLOG database affects all the modules that use the blackboard. These are the blackboard manipulation module, and the scheduler.

The scheduler orders knowledge source execution, and creates and updates the agenda. This agenda is kept in the blackboard, and the same blackboard consistency mechanism applies to it. In fact, the scheduler relies on this mechanism for removing no longer valid KSAR's from the agenda.

We can see in the module proportion pie chart that the scheduler is the most expensive module. When improving the performance of the EPBS system, this module should be redesigned. Specifically, we would advise handling the agenda without relying on the blackboard consistency mechanism.

3.4.3 Discussion

EPBS, as in Hearsay-III [ELF81], considers that it is important to maintain the consistency of the partial solutions stored in the blackboard. EPBS provides a clear definition of the consistency mechanism to be employed. This mechanism is a support-based truth maintenance system.

Other features of EPBS derive from its simple basic design and the use of PROLOG as implementation language, leading to a simple and very flexible system.

EPBS, like BB1, stores the agenda in the blackboard itself. This means that the KSAR's in the agenda are managed using the same consistency mechanism as the other bbels. In particular, the task of eliminating no longer valid KSAR's from the agenda is performed automatically.

From the framework description, we can see that the EPBS control cycle is very similar to the one of BB1, it varies in the order of module evaluation and in the way each module performs its task. The main difference between BB1 and EPBS is the control architecture of BB1 and the blackboard consistency mechanism of EPBS.

Use of the blackboard consistency mechanism is a good idea from the conceptual point of view. It simplifies the maintenance of the blackboard information for the user, and it allows the system to keep track of the justifications of all partial solutions reached by it, as well as the sequence of steps that generated it. This is extremely useful for providing good explanation facilities.

It, however, places an extremely heavy burden on the system as a whole. The use of this consistency mechanism degrades the response time of the blackboard, and thus, affects all system modules; in particular, the knowledge source activation module. We can see in the EPBS performance table, that the amount of time spent in blackboard manipulation, and knowledge source activation is excessive.

It would be better to leave the blackboard consistency mechanism as an option to the user. Currently, this mechanism cannot be turned off completely, as the knowledge source trigger module relies on it to maintain the agenda. From the performance point of view, it would be better if this module managed its agenda independently. The EPBS needs a more efficient blackboard manipulation module implementation, and a better way to manage its agenda.

3.5 Case 3: Blondie-I

Blondie-I is a blackboard shell based on the BB1 system shell [LVV86]. The Blondie-I system shell was the first of three blackboard shells produced in the EXPO project at the Dr. Neher Laboratories of the Dutch PTT. It is a relatively simple sequential blackboard shell. However it is general and flexible enough to permit experimentation using the blackboard architecture for problem solving. Blondie-I was implemented using POP-11 [BRS85], within the POPLOG AI environment.

3.5.1 Blackboard System Structure

General Module Structure

Module Description

According to [LVV86], and using the framework definitions, we can identify the four basic blackboard modules within the Blondie-I system:

- The *knowledge source execution module* (KSE), formed by the evaluation part of Blondie-I's `ks-handler`.
- The *blackboard manipulation module* (BBM), called `bb-handler`.

- The *knowledge source activation module* (KSA), formed by the test part of the **ks-handler** of Blondie-I, together with the **ksar-handler**.
- The *scheduler module* (SCH), called **inf-engine**.

We will define the Blondie-I *event handler* as a support module. This module plays a major role in the problem solving activity of Blondie-I. It works as an interface between the BB manipulation module and the KS activation module, preparing the information about the recent changes of the blackboard that will be used by the KS activation module to decide which knowledge sources trigger.

Blondie-I defines other modules: **cmdint**, **trace**, **io_handler** and **library**. We consider that they do not affect the system performance in a significant way.

Control Cycle

Blondie-I assumes that there is one knowledge source (initial knowledge source) that triggers on an empty event. The Blondie-I control cycle begins by activating the initial knowledge source (KS activation module), this KS is trivially chosen to be executed (scheduler module) and its body is evaluated (KS execution module). The body actions solve a specific problem, and modify the state of the blackboard (BB manipulation module). This produces a blackboard event for each modification that is placed in the event queue. The KS activation module then takes the event queue and produces the agenda, containing the list of active knowledge sources. The scheduler then chooses the knowledge source to be evaluated next. This cycle is repeated until a user defined function (problem_solved()) determines that a solution to the domain problem has been reached.

Figure 3.3 graphically describes the Blondie-I control cycle.

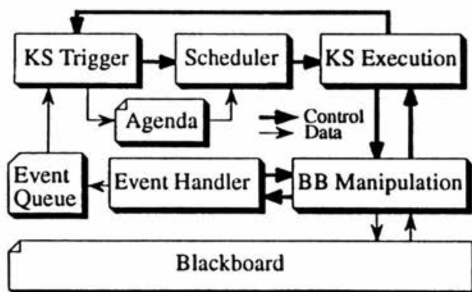

Figure 3.3. The Blondie-I Control Cycle

Type of Architecture

We can characterise the Blondie-I architecture as a *sequential* blackboard architecture.

Basic Modules

The Blackboard Manipulation Module

The Blondie-I blackboard structure is very similar to the BB1 one. It has a blackboard depth of two. This is, a blackboard can be divided into any number of sub-blackboards and each sub-blackboard can contain any number of levels. The different levels are the places where bbels are stored. In Blondie-I bbels are implemented as lists of attribute-value pairs with one of the attributes (NAME) unique within a level.

Blondie-I stores all system control information in the blackboard. To implement this, the Blondie-I system pre-defines a special sub-blackboard, the control blackboard, that uses the same level subdivision as BB1's control blackboard.

Blondie-I provides a set of functions to manipulate the blackboard and its contents. Apart from the initialisation and display of the blackboard contents, the blackboard manipulation module provides functions for:

- *Creation of bbels* on a given blackboard level.
- *Modification of a bbel.*
- *Searching for bbels* that match a given pattern (Read).

Note that there is no "delete" function. This is because every created bbel is kept for explanatory purposes.

Every time a bbel is created or modified, an event is produced. The BB manipulation module calls the event-handler to create a new event and update the event queue.

The Knowledge Source Activation Module

The Blondie-I knowledge sources are composed of two different parts. The first part consists of attribute-value pairs that contain the triggering conditions and general information about the knowledge source. The second part is the actual body of the knowledge source.

The knowledge source trigger conditions have an *Event* granularity. This is, they describe the event or events on which the associated knowledge source will trigger.

The Blondie-I triggering process is performed in a two step fashion. The trigger condition is divided in two parts. The first part is called *trigger* and describes the area of the blackboard on which the event originated and the type of event (add or modify). The second part is called *precondition* and is a procedure that performs any test and sets the values of local variables. When a given event matches the description in the trigger, a KSAR is generated and stored in a "triggered" list. This list is not the agenda. The KSAR's are kept in the triggered list until the procedure in the precondition yields a positive result. In this case the KSAR is said to be "invokable" and is stored in the agenda. During each cycle the system evaluates the preconditions of the KSAR's in the triggered list and moves the successful ones to the agenda. The KSAR's are removed from the agenda when they are scheduled for execution.

The Control Module

Blondie-I has an *explicit* control structure very similar to that of BB1. In fact, it uses a control blackboard whose structure is taken directly from BB1. The control blackboard is used to store the agenda and the control plan, allowing for the definition of control knowledge sources.

[LVV86] describes the Blondie-I control module (Control Unit) as composed of three procedures that implement the basic control loop. These procedures are:

`Update-To-Do-Set`, `Choose-KSAR` and `Interpret-KSAR`.

According to the framework definitions, the first procedure is part of the KS activation module, and the last one is part of the KS execution module.

The procedure `Choose-KSAR` is the actual implementation of the Blondie-I scheduler. It uses two user defined procedures to decide which KSAR to execute next. These are `determine_priority`, used for calculating the priority of the KSAR's in the Agenda, and `select_ksar` used to choose one KSAR to be evaluated.

The Knowledge Source Execution Module

The Blondie-I knowledge source execution module is implemented by the Interpret-KSAR function. This function calls the body of the knowledge source, written in POP-11. This calls other procedures and uses the procedures defined in the BB manipulation module to change the state of the blackboard.

Service Modules

Event Handler Module

The event handler module holds all the functions for the manipulation of events.

Events are structures that record blackboard manipulation operations. Blondie-I creates events when a new bbel is created and when one bbel is modified. This module includes functions for creating the event structure and maintaining the event queue. Its basic functions are:

event_create Creates a new event and adds it to the event queue.

event_get Returns the next event and removes it from the event queue.

The event handler module sits between the blackboard manipulation module and the knowledge source activation module. It prepares the blackboard modification information for consumption by the KS activation module.

3.5.2 Blackboard System Performance

We evaluated the Blondie-I system using the jigsaw-puzzle solver application on a 3 by 4 jigsaw. The pieces are distributed equally among four player knowledge sources. In total, the system has seven knowledge sources. The initialisation and termination knowledge sources are implemented independently.

The following table shows the module usage of the JSP application.

JSP (3x4) Application: Module Use

Module	Number of calls	Time Use
BB. Manipulation Module	N_M 2237	G_M 73.16%
KS. Activation Module	N_T 121	G_T 5.48%
Scheduler Module	N_S 14	G_S 3.57%
KS. Execution Module	N_B 14	G_B 2.51%
Service Modules	N_{S_p} 872	G_{S_p} 15.27%

Number of Calls Time Use

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times Average	Module Times Total
BB. Manipulation Module				AM 0.0040	TM 7.7332
	BBel Create	AC_M 0.0089	PC_M 1.25%		
	BBel Access	AR_M 0.0034	PR_M 95.53%		
	BBel Modify	AM_M 0.0037	PM_M 3.22%		
	BBel Destroy	AD_M 0.0000	PD_M 0.00%		
KS. Activation Module				AT 0.0048	TT 0.5759
	KS. Trigger	AT_T 0.0048	PT_T 100.00%		
Scheduler Module				AS 0.0270	TS 0.3773
	Choose KS.	AS_S 0.0270	PS_S 100.00%		
KS. Execution Module				AB 0.0189	TB 0.2650
	Evaluate KS.	AB_B 0.0189	PB_B 100.00%		
Event Handler Module				AEv 0.0018	TEv 1.6145
	event_create	A_{ec} 0.0024	PEc_E 57.00%		
	event_get	A_{ed} 0.0011	PEg_E 43.00%		

Module Influence

Total System Run Time	T_{Run}	10.57	
Total Unaccounted Time	TUa	0.09	
Total System Time	T_{Sys}	10.66	
Unaccounted Time Prop.	P_{Ua}	0.84%	
Module Average Influence			
BB. Manipulation Module	I_M	7.07%	■
KS. Activation Module	I_T	8.49%	■
Scheduler Module	I_S	47.76%	⊟
KS. Execution Module	I_B	33.55%	◫
Service Modules	I_{Sp}	3.13%	⊞

System Performance

Average Cycle Time	A_{Cy}	0.75
System Speedup	SU	0.99

Table 3.3: Blondie-I General Performance table

We can see in the modules' time use pie chart that most of Blondie-I execution time is dominated by the blackboard manipulation time (73.16%). This may seem excessive, however, it is a direct result of the JSP application. This application make very heavy use of the blackboard.

Note that the module average influence pie chart is very different from the first pie chart presented. The modules' time use pie chart shows the proportion between the total module execution times, while the second pie chart shows the proportion between the average module times.

In the module proportion pie chart we can compare the different blackboard modules in a way that is relatively independent from the application. We can see that the blackboard manipulation module is not one of the most expensive modules. These are the scheduler module, and the knowledge source evaluation module. When improving the performance of the Blondie-I system, we would concentrate in optimising these two modules.

3.5.3 Discussion

The Blondie-I blackboard shell's functionality is influenced by the BB1 system. Most of the discussion presented when analysing the BB1 system also applies to Blondie-I.

Comparing the function average times form the respective module performance tables, we can see that Blondie-I is about an order of magnitude faster than BB1. This, however, can be explained by the differences in implementation languages (LISP versus POP-11), and, more importantly, system processor (MicroExplorer versus Sun4

Sparc).

We can also see a remarkable similarity between BB1 and Blondie-I by comparing their respective module influence pie charts. Both charts show that the most expensive modules are the scheduler and knowledge source execution modules. These results suggest that there is an intrinsic behaviour of the BB1-type architecture, as two independent implementations of it, using different implementation languages, produced similar results.

This suggest that the most promising approach to improving the architecture performance is not to optimise the scheduler and knowledge source execution modules, but to change the way they work.

3.6 Case 4: Blondie-III

Blondie-III [FvLV88] is the third generation of blackboard shells developed at the Dr. Neher Laboratories of the Dutch PTT. The first generation was Blondie-I, a sequential blackboard shell [LVV86]. The second generation was Blondie-II, a parallel blackboard shell in which knowledge sources could be executed in parallel, with a single blackboard and control unit.

Blondie-III [FvLV88] is a distributed version of the Blondie-I blackboard shell. It is a program network in which each one of the nodes is an extension of the Blondie-I blackboard shell. The main extensions have to do with inter-process communication and external event processing.

Blondie-III was implemented using POP-11 [BRS85], within the POPLOG AI environment. Blondie-III uses UNIX message passing facilities for the communication between nodes.

3.6.1 Blackboard System Structure

General Module Structure

Module Description

The Blondie-III system is an extension of the Blondie-I system, as such, it possess the same module structure. Blondie-III changes (extends) some of the modules' functionality and adds one more module, the communications module.

Using the framework definitions, and according to [LVV86], we can identify the four basic blackboard modules within the Blondie-III system. These are:

- The *knowledge source execution module* (KSE), formed by the body evaluation part of the `ks-handler`.
- The *blackboard manipulation module* (BBM), implemented by the `bb-handler`.
- The *knowledge source activation module* (KSA), formed by the trigger evaluation part of the `ks-handler`, and the `ksar-handler`.
- The *scheduler module* (SCH), implemented by the `inf-engine`.

We will define as support modules the *event handler* and the *communication library* (communication module). Both these modules play a major role in the problem solving activity of Blondie-III. The event handler works as an interface between the BB manipulation module and the KS activation module. It prepares the information about the recent changes of the blackboard that will be used by the KS activation module to decide which knowledge sources trigger. The communication module is used for exchanging information between the nodes of the system that are contributing to the problem solving process.

Control Cycle

Blondie-III does not have a global control cycle. Its operation is determined by the control cycles local to each node. These control cycles are independent, interacting only via messages.

Blondie-III assumes that there is one knowledge source (initial knowledge source) per node that triggers on an empty event. Each local control cycle begins by activating its initial knowledge source (KS activation module), this KS is trivially chosen to be executed (scheduler module) and its body is evaluated (KS execution module). The body actions modify the state of the blackboard (BB manipulation module) and/or send messages to other nodes. Every modification to the blackboard produces a blackboard event that is placed in the event queue (event handler). The KS activation module then takes the event queue and produces the agenda, containing the list of active knowledge sources. Next, the scheduler looks at the agenda and chooses the knowledge source to be evaluated.

In earlier versions of Blondie, the control cycle is repeated until a user defined function (`problem_solved()`) determines that a solution to the domain problem has been reached. In Blondie-III this is still true, but in addition the `problem_solved()` function of a given node must send a termination message to all the other nodes. This facility is provided by the communication module.

Figure 3.4 graphically describes the Blondie-III control cycle.

Type of Architecture

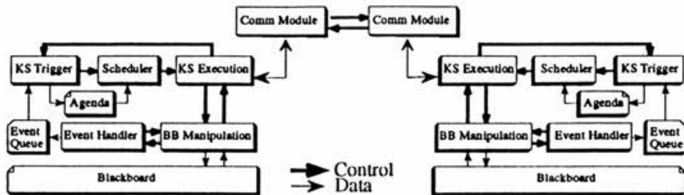

Figure 3.4. The Blondie-III Control Cycle

The Blondie-III system architecture is composed by a network nodes, where each one of the nodes is a sequential blackboard. Blondie-III's architecture is *distributed*. As such, it is formed by a number of nodes connected via a communication module. Each node is a complete blackboard system with its own set of knowledge sources, blackboard and scheduler.

Basic Modules

The Blackboard Manipulation Module

Each Blondie-III system node has the same blackboard structure as Blondie-I. This structure is very similar to BB1's. It has a blackboard depth of two. This is, a blackboard can be divided int any number of sub-blackboards and each sub-blackboard can contain any number of levels. The different levels are the places where bbels are stored. In Blondie-III bbels are implemented as lists of attribute-value pairs with one of the attributes (NAME) unique within a level.

Blondie-III stores all system control information in a special sub-blackboard, that uses the same level subdivision as BB1's control blackboard.

Blondie-III provides a set of functions to manipulate the blackboard and its contents. Apart from the initialisation and display of the blackboard contents, the blackboard manipulation module provides functions for:

Creation of bbels on a given blackboard level.

Modification of a bbel.

Searching for bbels that match a given pattern (Read).

Remove bbels from a given blackboard level.

One difference from its predecessor, Blondie-I, is the possibility of removing bbels.

Every time a bbel is created, modified, or deleted, an event is produced, and the event-handler is called to create the new event and update the event queue.

The Knowledge Source Activation Module

The Blondie-III knowledge sources are composed of two different parts. The first part consists of attribute-value pairs that contain the triggering conditions and general information about the knowledge source. The second part is the actual body of the knowledge source.

The knowledge source trigger conditions have an *Event* granularity. They describe the event or events on which the associated knowledge source will trigger.

As in Blondie-I, the Blondie-III triggering process is performed in a two step fashion. The trigger condition is divided in two parts. The first part is called *trigger* and describes the area of the blackboard on which the event originated and the type of event (add or modify). The second part is called *precondition* and it is a procedure that performs any test and sets the values of local variables. When a given event matches the description in the trigger, a KSAR is generated and stored in a "triggered" list. This list is not the agenda. The KSAR's are kept in the triggered list until the procedure in the precondition yields a positive result. In this case the KSAR is said to be "invokable" and is stored in the agenda. During each cycle the system evaluated the preconditions of the KSAR's in the triggered list and moves the successful ones to the agenda.

Blondie-III arranges the KS trigger conditions in a tree determined by the kind of events they are expecting. The first level of the tree is determined by the kind of event (new or modified), the second and third level are determined, respectively, by the "blackboard" or blackboard level on which the event occurs.

The new events now traverse the tree according to their characteristics. When in a node, the event triggers the knowledge sources in that node. An extra result of this modification is that the user is now allowed to use **AND** and **OR** constructs when specifying the trigger conditions. Apart from activating knowledge sources, it is possible for other nodes to ask to be informed when a specific event occurs (via the communication module). These requests are also stored in the nodes of the tree and answered whenever an event goes through it.

The Control Module

Within a node, Blondie-III has the same control structure as its predecessor Blondie-I. This is an *explicit* control structure very similar to that of BB1, using a control blackboard with equally named levels. The control blackboard is used to store the agenda and the control plan, allowing for the definition of control knowledge sources.

[LVV86] describes the Blondie-I control module (Control Unit) as composed of three procedures that implement the basic control loop. These procedures are:

`Update-To-Do-Set`, `Choose-KSAR` and `Interpret-KSAR`.

Blondie-III modified this control loop slightly by adding some code for dealing properly with some "special" communications, such as `comm_stop`. According to the framework definitions, the `Update-To-Do-Set` is part of the KS activation module, `Interpret-KSAR` is part of the KS execution module, and the extra code we will consider to be part of the communications module.

The Blondie-III scheduler is implemented by the `Choose-KSAR` procedure. It uses two user defined procedures to decide which KSAR to execute next. These are:

`determine_priority`, used for calculating the priority of the KSAR's in the Agenda,

and `select_ksar`, used to choose one KSAR to be evaluated.

The Knowledge Source Execution Module

Blondie-III's `Interpret-KSAR` function implements the knowledge source execution module. This function calls the body of the knowledge source, written in POP-11. This calls other procedures and uses the procedures defined in the BB manipulation module to change the state of the blackboard. It may also use procedures within the communication module in order to communicate with other nodes of the architecture.

Service Modules

Event Handler Module

The event handler module holds all the functions for the manipulation of events. This includes functions for creating the event structure and maintaining the event queue. Its basic functions are:

`event_create`	Creates a new event and adds it to the event queue.
`event_get`	Returns the next event and removes it from the event queue.

The event handler module sits between the blackboard manipulation module and the knowledge source activation module. It prepares the blackboard modification information for consumption by the KS activation module.

Communication Module

Two nodes within the Blondie-III architecture communicate by sending messages to one another. These messages are classified in two main groups, the "normal" and "special" messages. The normal messages are used to exchange domain information between nodes. The special messages contain control information. The special messages receive top priority handling. Sending domain information to a node means creating new bbels in that node's blackboard. Receiving information from another node is just

reading from their blackboard.

The communication module basic functions are:

`comm_inform`	Sends data and no answer is needed.
`comm_request`	Requests information and answer is expected.
`comm_ask`	Requests information but answer is not necessary.
`comm_event`	Asks to be notified when a given event occurs on another node.
`comm_reply`	Sends information in answer to `comm_request` or `comm_event`.
`comm_deny`	Discards a message sent earlier via `comm_event`.
`comm_stop`	Sends a shutdown message to other nodes.

3.6.2 Blackboard System Performance

We used the jigsaw-puzzle solver application for the evaluation of the Blondie-III system. This application was implemented using two nodes. These nodes are identical blackboard systems. They will each hold two players, and the piece-misplaced, initialisation and termination knowledge sources. These are five knowledge sources on each node.

In order to solve the 3 by 4 jigsaw in a distributed environment, we will split the jigsaw, and pieces in two smaller, 3 by 2 jigsaws. For simplicity, we will assume that each node has the pieces corresponding to its part of the jigsaw. The nodes, however, are not independent. The two jigsaws share the middle division. In order to solve the problem, the nodes have to communicate the placement of pieces at the jigsaw's boundary. Each node works as the blondie-I's version, solving its own local jigsaw. The solution is found when both jigsaws are completed.

The whole system was run on a Sun4 workstation, on a single processor. The concurrency was simulated using UNIX processes. The following table shows the combined module usage of the JSP application, and the Blondie-III general performance.

JSP (3x4) Application: Module Use

Module	Number of calls		Time Use	
BB. Manipulation Module	N_M	2347	G_M	74.58%
KS. Activation Module	N_T	122	G_T	4.75%
Scheduler Module	N_S	16	G_S	4.42%
KS. Execution Module	N_B	16	G_B	3.09%
Service Modules	N_{Sp}	484	G_{Sp}	13.16%

Number of Calls Time Use

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times: Average	Module Times: Total
BB. Manipulation Module				AM 0.0222	TM 64.7604
	BBel Create	AC_M 0.0378	PC_M 1.19%		
	BBel Access	AR_M 0.0276	PR_M 95.74%		
	BBel Modify	AM_M 0.0234	PM_M 3.07%		
	BBel Destroy	AD_M 0.0000	PD_M 0.00%		
KS. Activation Module				AT 0.0338	TT 4.1236
	KS. Trigger	AT_T 0.0338	PT_T 100.00%		
Scheduler Module				AS 0.2400	TS 3.8400
	Choose KS.	AS_S 0.2400	PS_S 100.00%		
KS. Execution Module				AB 0.1678	TB 2.6848
	Evaluate KS.	AB_B 0.1678	PB_B 100.00%		
Event Handler Module				AEv 0.0073	TEv 3.0104
	event_create	A_{ec} 0.0049	PEc_E 57.01%		
	event_get	A_{ed} 0.0096	PEg_E 42.99%		
Communication Module				ACm 0.0437	TCm 8.4136
	comm_inform	A_{ci} 0.1119	PCi_E 18.37%		
	comm_request	A_{cr} 0.1898	PCr_E 79.59%		
	comm_ask	A_{ca} 0.0000	PCa_E 0.00%		
	comm_event	A_{ce} 0.0000	PCe_E 0.00%		
	comm_reply	A_{cp} 0.0000	PCp_E 0.00%		
	comm_deny	A_{cd} 0.0000	PCd_E 0.00%		
	comm_stop	A_{cs} 0.0043	PCs_E 2.04%		

Module Influence

Total System Run Time	T_{Run}	86.83
Total Unaccounted Time	TUa	0.09
Total System Time	T_{Sys}	86.92
Unaccounted Time Prop.	P_{Ua}	0.10%
Module Average Influence		
BB. Manipulation Module	I_M	4.31%
KS. Activation Module	I_T	6.57%
Scheduler Module	I_S	46.62%
KS. Execution Module	I_B	32.60%
Service Modules	I_{Sp}	9.90%

System Performance

Average Cycle Time	A_{Cy}	5.43
System Speedup	SU	1.00

Table 3.4: Blondie-III General Performance table

We can see in the modules' time use pie chart that most of Blondie-III execution time is dominated by the blackboard manipulation time (74.58%). This is similar to Blondie-I performance, and it is a direct result of the JSP application. The difference with Blondie-I is that in Blondie-III, the service modules use a greater proportion of the total system run time.

In the module performance table, we can identify the most expensive functions. In the case of Blondie-III we can see that the Choose KS function is the most expensive. As with Blondie-I, the knowledge source evaluation function is also expensive. Besides these basic modules, the new communication module also have significant average function times for the `comm_inform` and `comm_request` functions. When improving the performance of the Blondie-III system, we would concentrate on improving these module functions.

The function costs are also reflected in the module proportion pie chart. The scheduler, and knowledge source evaluation are still the most expensive modules. For Blondie-III, the service modules are also important, due to the inter-process communication module.

Blondie-III, similar to BB1, uses a graphical user interface for showing the progress of each system node. We were unable to disconnect the use of this interface. The use of it affects the performance of all basic system modules. This, together with the use of the communication module, explains the difference in performance between the Blondie systems. According to the average function times, the Blondie-III system is several times slower than Blondie-I. This is between 1.5 times (AC_M), and 8.3 (AS_S) times slower.

3.6.3 Discussion

Each Blondie-III node is a blackboard system similar to Blondie-I. The implementation of these nodes included some changes to the basic blackboard modules. In particular, the triggering process was modified in order to improve its performance. However, we can see in Blondie-I module proportion pie chart that the impact of this improvement on the overall system performance is small. In fact, if we disregard the service modules (I_{S_p}), we find that Blondie-III's module proportion is very similar to that of Blondie-I.

We notice a fair amount of time spent on Blondie-III's service modules. This is due to the use of the communication module (as can be seen in the total module times). This is not unreasonable in distributed blackboard systems, as they are concurrent systems designed on the assumption that communication between processes is expensive. We would expect the time spent in this module to be more or less equally divided between the functions for sending and receiving of messages (`comm_inform` and `comm_request`). However, in the communication module table, we can see that most of the time is spent in requesting information ($PCr_E = 82.90\%$), and includes the time for waiting for other node's information.

The delays introduced by the use of communication is a usual problem of distributed and parallel systems. There are two issues that have to be dealt with when implementing a distributed blackboard system shell. One is the selection of the communication mechanism to be used. Great care should be taken in this selection, in order to reduce this type of delay to a minimum. The second is the design of the communication system itself. For example, the knowledge sources do not need to use it directly. It could be built into the blackboard, say, as a special level or levels where knowledge sources read and write information. Note that with this scheme the knowledge sources cannot explicitly wait for information, they can, however, be designed to trigger when the information is available.

Another cause of the excessive use of communication within a distributed system is the way the particular applications are implemented. In a distributed environment it is assumed that communication costs are high. In order to minimise inter-process communication, it is best to subdivide the domain problem spatially, if possible. Some problems are can be subdivided in this way more or less naturally, however, there are others for which this is a difficult task.

3.7 General Discussion

The general architectures of the systems evaluated are very similar. They arrange the basic blackboard modules in more or less the same fashion. This is to be expected, as they all evolved more or less directly from the Hearsay-II blackboard architecture. BB1 and EPBS modelled their system architecture from the Hearsay-II system, with some variations in the control mechanism (BB1) and truth maintenance system built-into the blackboard (EPBS). The Blondie-I system based its implementation directly on BB1, and Blondie-III is a distributed version of it. These similarities can easily be seen in the framework description.

EPBS is different from the other three systems in the choice of blackboard structure. This choice is, mainly, influenced by the implementation language used. The result was a extremely flexible blackboard structure based on PROLOG terms. The BB1 and Blondie's blackboard structures are based on LISP (and POP-11) named-lists structure, restricted to three levels of nesting. This, although general enough, is less flexible than the unrestricted PROLOG terms.

EPBS also has a different control structure. It uses an implicit control scheme, while BB1 and the Blondie systems use an explicit control plan. In this case, BB1 and the blondie systems have a much more flexible control structure, using the blackboard

to represent control knowledge. This allows the design and implementation of high level control knowledge sources specific to the particular domain problem.

From the performance point of view, Blondie-I seems to be the fastest system, as reflected on its small average cycle time (A_{Cy}), compared to that of BB1, EPBS and Blondie-III. However, this cannot be totally stated as there are too many variations in the implementation platform and language used.

We do notice that the performance decreased from Blondie-I to Blondie-III. The average module times of Blondie-III are consistently higher that those of Blondie-I. From the data present in their respective performance tables we cannot identify a cause for this behaviour. However, the tables show that both systems behave in a similar way. Besides the increase in service module activity due to communications, Blondie-III's basic module proportion is very similar to that of Blondie-I. According to the authors, the implementation of both systems is very similar, which is reflected in the similarity in the proportion between the basic blackboard modules. All this points to a change in the systems' environment as the cause for the drop in blackboard performance. Differences in implementation would affect each basic module in different ways, thus modifying their proportions. Both systems were implemented using the POPLOG environment. However, Blondie-III loads and uses the POPLOG graphical user interface, and an inter-process communication mechanism. Blondie-I, on the other hand, does not need to communicate, and only has a line-by-line user interface. The user interface speed seems to be a problem in Blondie-III. The BB1 system also has problems with a slow user interface affecting the performance of all the basic modules.

The blackboard implementation in all four systems is very simple. They all use the facilities provided by their implementation language. BB1 uses LISP lists and a matching mechanism provided with the Explorer system, EPBS uses the PROLOG database, and the Blondie systems use the POP-11 list database and associated matching utilities. As all the system agendas are stored in the blackboard, the performance of the particular language reflects on the blackboard manipulation and knowledge source activation modules.

In the EPBS system, the use of a truth maintenance system built-in to the blackboard, coupled with the particular use of the PROLOG database affected, in a large degree, the general system performance. If such a consistency mechanism is needed, we should take great care on its implementation and together with that of the blackboard module.

As we can see, in general all systems would benefit from an optimisation of some of their internal processes. We can use the module proportion data to guide

optimization efforts to those modules that are more expensive. We can also use the application specific data to decide if the most used modules should be also optimized.

We believe that to improve performance further, there is need for the efficient use of available parallel processing technology. We also think that the blackboard architecture is well suited for parallel processing.

However, the systems analysed in this chapter do not adapt well to a parallel environment. All of them, except Blondie-III, are specifically designed for a sequential environment. Blondie-III is a distributed system, and as such, its global structure is not a blackboard system. It is a number of communicating nodes, each being a blackboard system. The individual nodes are designed as sequential blackboard systems.

All the analysed systems depend on a sequential centralised scheduler (by node, in the case of Blondie-III). Their control structure and problem solving strategies depend on this fact. For example, the complex control mechanism of BB1 depends on a sequential scheduler and an unique agenda. This is also true for the Blondie systems.

The systems presented here would also need to address issues such as controlling simultaneous accesses and modifications to the blackboard, and maintaining a coherent global state. EPBS, in particular would need a specialised concurrency control mechanism, coupled to the blackboard's truth maintenance system.

3.8 Summary and Conclusions

In this chapter, we have showed how we can use the framework described in the previous chapter for the analysis of the performance of blackboard systems. We used it to analyse four blackboard system shells present in the literature. We showed how it can be used to decide where optimisation efforts should be concentrated. These efforts can be directed to those blackboard modules that have the greatest impact on the overall system performance. The framework, also, can be used for the evaluations of the results of the system optimisations.

The blackboard systems analysed, being research tools, concentrate on studying particular extensions to the blackboard model proposed in the Hearsay-II system. BB1 explored the use of a blackboard architecture for control, EPBS incorporated a truth maintenance system into the blackboard, and Blondie-I was the starting point for Blondie-III that explored the distributed blackboard model. The performance of these systems was not optimised. All these systems would benefit from the optimisation of all their modules.

The performance framework can be used to direct optimisation efforts to the most promising areas. For example, when analysing BB1, we concluded that optimisation efforts should be directed towards the scheduler and knowledge source execution modules, as they represent the most expensive system functions (their average module times, AS and AB, are much higher than AM and AT).

The study of the four blackboard systems presented in this chapter allows us to draw a number of conclusions about the design and implementation of blackboard systems. From the EPBS experience, we conclude that we should be very careful when including new functionality in the blackboard. We have the risk of overloading it, affecting the whole system performance significatively. Although a high level blackboard truth maintenance system may be desirable, we believe that it should not be imposed in all problem domains.

The Blondie-III system shows possible problems that may be confronting when implementing distributed blackboard systems. First, is the importance of choosing a good communication scheme. This is even more important for parallel blackboard systems, as they are designed to communicate constantly. The other is the importance of a good application design. Blackboard shells are only programming tools. It is not unusual to implement an inefficient system using very efficient tools. The system, however, should enforce some sort of programming restrictions without compromising flexibility and expressivity.

We think there is scope for low-level process optimisation in some modules of the systems presented here, and in blackboard systems in general. However, we feel that not all the high-level design options have been explored. In particular, we find that the parallel blackboard architecture has not been fully researched. We believe that, although necessary, the module optimisations can help up to a point. The real improvement would come from attacking the issues from a different perspective. The parallel blackboard architecture provides us with such a possibility.

Chapter 4

The ABB System

4.1 Introduction

In the this chapter we will lay out the basis of a new blackboard system architecture. This new architecture will present a number of design variations and improvements from the traditional blackboard architecture. These variations are aimed to allow the efficient use of parallel and distributed computer architectures. This will demonstrate that it is possible to improve blackboard system performance by modifying high level design characteristics.

We consider that many of the performance problems confronted by parallel and distributed systems arise from the fact that they try to adapt the idea of centralised resource management to a non-sequential environment. Distributed systems partially avoid the problem by assuming that the problem to be solved can be spatially decomposed, and tackling each sub-problem with a separate blackboard system (each with its own centralised scheduler). Parallel systems, however, do face the problem of adapting the scheduler operation. For example, the CAGE system [Aie86] has a Hearsay-II-like scheduler that chooses for execution several knowledge sources at a time (instead of just one). In the CAGE system the scheduler is, effectively, a bottleneck. The POLIGON system [Ric86] solves this problem by not having any scheduler whatsoever. However, we do not consider this to be a good solution. Most of the system processing can disperse by solving superfluous problems.

From this, and from the experiences presented in the previous chapter, we find that a successful parallel blackboard system must comply with a number of characteristics. We can divide these characteristics into two basic groups. The first are system design characteristics, and the second desirable implementation characteristics.

Design Characteristics.

- *Control architecture.*

  A good parallel blackboard system should have some sort of control structure that allows the control of knowledge source execution. Ideally, this control structure will make possible to express and enforce high level control decisions. It should be possible to design a control architecture similar in functionality to the one provided in BB1.

- *No high level communication structure.*

  This is not to say that there should be no communication module. The system should be designed in a way that inter-process communication is totally transparent to the user. This communication is dealt with at an implementation level.

In this way, the communication infrastructure can be tuned to the actual needs of the blackboard system, without the current applications depending on it.

Implementation Characteristics.

- *Distributed control structure.*

  This is needed in order to avoid control bottlenecks. The absence of a centralised scheduler would also improve the system reliability.

- *Efficient concurrency control mechanism.*

  In any parallel blackboard system, there may be several system elements that need to query, or modify, global information. The mechanism that controls the access to the information should be chosen very carefully. It must provide maximum information availability, while at the same time, maintain a coherent global state.

- *Efficient communication mechanism*

  All parallel systems are built on the assumption that inter-process communication is fast. This means that communication is thoroughly used in all the system modules. The mechanism chosen for performing this communication should be as efficient as possible. It should avoid unnecessary process blocking because of transmission or reception of data, thus allowing a maximum use of the parallel processing resources.

- *Efficient knowledge source activation.*

  As with blackboard systems in general, a parallel blackboard system needs to implement its knowledge source activation module in an efficient fashion. Each knowledge source triggering should be performed as independently as possible from other knowledge source activations in order to allow it to be performed in parallel. Note that this is completely different from sequential implementations, where the triggering process usually avoids duplicating efforts by integrating all knowledge source activations within a RETE-like [For82] algorithm. Efficient compromises can be achieved between these two approaches by implementing some mixture of trigger distribution and RETE-like evaluation.

- *Maximum resource utilisation.*

  In particular, the system should maximise the number of knowledge sources working on the problem domain, while maintaining control on them. All control efforts should be concentrated on insuring that the knowledge sources work towards a

common goal, or perhaps towards complementary goals that may prove useful, but that do not complicate the work of the other knowledge sources.

In the present chapter we will concentrate in the high-level design and specification of a new blackboard architecture. The implementation decisions will be presented in the next chapter.

We will present a new variation of the traditional blackboard model that extends the Hearsay-II style scheduler in a way that allows control decisions to be performed in parallel and asynchronously. In this new model, control knowledge is not centralised. The scheduler is still present, but it is no longer a single entity; its operation is spread among a set of control elements associated all the other system components.

With the definition of the new blackboard model we will also try to solve a second problem present in parallel systems. This problem is related to the data transformation needs of the system. Parallel systems, and specially real-time systems, have to deal with low level data transformation that derives from several areas such as, data filtering, data sumarisation, and close-loop control and monitoring. Hayes-Roth [HR87] tried to solve this problem by having a group of extra-blackboard agents filtering external data, so that the ones posted in the blackboard have a higher level nature. This, however, is a very specialised solution, tailored for one specific system.

In the model we present we solve the problem by introducing the idea of an **active blackboard**, capable of processing on its own. This allows the definition and placement of low level data transformations, close to where the data is to be kept. This avoids the need for agents not contemplated in the blackboard model, or the need for creating small specialised knowledge sources dedicated to this task.

In order to design a fully parallel architecture, we had to re-think the blackboard architecture from its origins. In Section 4.2 we present a modification of the traditional blackboard and experts metaphor that, we consider, expresses in a clear and intuitive form the interactions between different blackboard components.

In Section 4.3 we will describe informally a high-level, implementation independent, blackboard model, the active blackboard (*ABB*) model. In Section 4.4 we will formally specify the ABB model using the Z specification language.

In Chapter 5 we will present an implementation of a blackboard system shell based on the ABB model. The architecture is able to make full use of parallelism as in POLIGON, while providing means of controlling knowledge source execution at every stage of the problem solving activity.

4.2 Blackboard, Experts and Desks.

In order to fully understand the different interactions and problem solving activity, that can be generated from parallel and distributed blackboard systems, we consider it useful to elaborate a little on the original "blackboard and experts" metaphor that originated in earlier works on blackboard systems.

Suppose that we need to solve a difficult problem. This problem can be solved by the interaction of a group of experts. Each expert knows how to solve specific sub-problems within the global problem. Their knowledge does not normally cover the solution of the whole domain problem. It is the interactions between the different experts that will produce the solution of the global problem.

In order to provide a framework for the experts to work together, we prepare a room with a blackboard and some desks. The experts will use the blackboard to hold the information about the problem, as well as the partial solutions they reach. The desks are the areas where the experts work and think. They need to sit on a desk in order to solve any problem. Although a desk can be seen as a place where recent information is kept, we prefer to view it as the centre of the expert's working area. All public information (recent or not) will be strored in the blackboard. As the desks may be used by several experts, we do not allow them to leave private information in it. All experts will carry their own private information. The desks will only be used as working areas. The steps that experts follow in the execution of their work are as follows:

- Initially, the experts look at the blackboard, searching for problems that fall within their area of expertise.
- When one expert finds one such problem, it goes to a desk.
- Once at a desk, the expert solves the problem.
- When it is solved, the expert places the solution in the blackboard.

Following these steps, the experts solve sub-problems and cooperate in the solution of greater problems. Note that the experts cannot solve problems while standing-up. They need to use a desk. We could think of some situations in which the number of desks is smaller than the number of experts. This will force the experts to wait until a free desk becomes available.

On a blackboard system, the blackboard is represented as a global data structure; the experts are represented by the different knowledge sources, and the desks are the

actual places where the knowledge sources are executed. The desks are meant to represent the processors where the knowledge sources are evaluated.

4.3 The Active Blackboard (*ABB*) Model

The active blackboard model is a modification of the original blackboard model that keeps all its original simplicity, but it defines the way in which the expert's work can be managed and directed. This model is based on the "blackboard, experts and desks" metaphor, described in the previous section.

The problem solving activity of the experts in the metaphor seems quite straight forward. However, there are a number of assumptions made based on the fact that the experts are human beings, and know how to solve some conflicts that may arise while they perform their jobs. When we try to design a problem solving architecture based on this metaphor we find that we need some mechanisms with which to solve the conflicts. In order to do this, we need to identify and make explicit the basic assumptions that make the experts' work possible. These are:

- *The experts know how to understand other expert's results.*

  This is the basis of communication and cooperation between experts. This understanding is represented by the blackboard structure. All experts must agree in the configuration of their common areas. This defines the blackboard structure, which can be seen as the specification of the common language between experts. Without this understanding the blackboard model would not work.

- *The experts know how to get out of each other's way.*

  This assumption has two parts. First, the experts should know how to get out of each other's way when choosing problems on the blackboard. Second, the experts should know how to use the desks in an orderly manner.

  When looking for problems in the blackboard, there may be more than one expert interested in a particular piece of information. This information may form part of the problems identified by the experts; indeed, the information may be the actual problem they want to solve, in which case they must decide which expert should solve it (maybe more than one).

  When an expert decides that it will solve a problem, it goes to a desk and (hopefully) solves it. At this point, it should know how to wait for a free desk and avoid fighting with other experts who may also be waiting.

- *The experts know how to prioritise their work.*

  At a given moment, an expert might identify several problems in the blackboard that it can solve. It must decide which problem should be solved first. This could be done either by the expert knowing how to give priorities to each problem, or by having somebody else provide the guidelines and the expert following them.

The first assumption is addressed by all blackboard systems by providing a central blackboard. The richness of the structure of this blackboard varies from system to system, however, it always allows the storage of partial solutions in a language common to all experts. The experts (knowledge sources) know how to place and retrieve information from it.

The other two points can be seen as control assumptions. They refer to the need of ordering or guidance of the expert's work in order to avoid and resolve conflicts in the use of resources. These resources may be the blackboard, the desks, or the experts' time.

Early sequential systems, such as Hearsay-II faced the problem that their experts had only one desk in which to perform their work. They solved the problem by placing the knowledge about conflict resolution and priorities in a scheduler that kept guard of the desk and allowed one expert at a time to use it. This means that the central scheduler can closely control the activity of the experts. It can prioritise the problems to be solved by the experts, not only individually, but as a group.

The problem of controlling the experts is considered a difficult problem that can be solved with the blackboard and experts' scheme. This was the basis of BB1. It had a group of experts that solved the domain problem and another group that built a control plan that was then followed by the scheduler. This control plan was formed by strategies, control heuristics and focuses.

In general, the control schemes within blackboard systems were based on the notion of a strong scheduler, capable of controlling expert activity as a whole. This produced a well understood problem solving mechanism. Parallel and distributed systems tried to keep this mechanism more or less intact. The problem now is that the experts have more than one desk in which to solve problems, and one central scheduler is no longer appropriate.

The distributed architectures are based on the premise that the cost of taking information from one desk to another (communication cost) is high. This led to an organization of semi-independent nodes that communicate sporadically. That is, they have several rooms with a set of experts, a blackboard, a scheduler, and a desk. Each

room solves similar or related problems, and they communicate with each other whenever they come to a conclusion that they think will be useful to some other room. Given the communication restriction, this is a good architecture.

Parallel architectures, however, assume that communication costs are small. Systems such as CAGE kept a central scheduler that controlled the access to the different desks. This had the problem that the operation of the scheduler itself was a bottleneck of the system. The POLIGON system went to the other extreme and eliminated the use of the scheduler, losing any ability to control and direct the execution of the experts.

4.3.1 Control Mechanism

At first, it seemed that the issue of control on parallel systems was a case of finding a compromise between the CAGE and POLIGON approaches. We think, however, that this is not the case. If we understand the control assumptions as presented earlier, we can present a model of parallel blackboard problem solving in which all conflicts can be solved without having any bottleneck. This model is the active blackboard (ABB) model.

The basis of the ABB control model is the separation of the three control decisions identified earlier. These decisions control the use of problems in the blackboard, the use of the desks and the use of the experts. These three types of decisions can then be placed close to their respective blackboard elements, the blackboard, the desks, and the knowledge sources. *In the ABB model, there will be three types of schedulers.* The first one will be close to each type of bbel, and will decide which knowledge source (or knowledge sources) will be allowed to solve a particular problem. There will be one of these schedulers per type of bbel. The second type will be close to the knowledge sources, and will decide which problem should be solved first. There will be one of these schedulers per knowledge source. Finally, the last type will decide the order of the knowledge sources within one desk. There will be one of these per desk in the system. This means that control decisions are fully distributed, that there is no central scheduler, and that these decisions can be performed in parallel.

As in the original model, our knowledge sources look at the blackboard for suitable problems to solve. Whenever such a problem appears in the blackboard, the knowledge source activates (triggers). Let P_i be information about problem i, and K_j information about the knowledge source j. The tuple $< P_i, K_j >$ uniquely identifies the activation. We call this tuple the *Knowledge Source Activation Record* (**KSAR**). In the next definitions, we will be using tradition mathematical notation, with the help of the Z specification language [Spi89]. See Appendix D for a brief introduction to the

Z specification language.

Definition 4.1 : KSAR

$$KSAR \in \{< P_i, K_j >, \forall\, i, j\}$$

The act of identifying a problem is fundamentally different to that of actually solving it. The knowledge sources are then divided in two parts, the *Trigger Condition* and the *Action*. These two parts are quite independent, and do not necessarily reside in the same place within the system. We define that the KSARs are the only means of communication between a knowledge source trigger condition and its action part. When an activation occurs, a KSAR is generated and sent to the knowledge source action part for processing. At a given moment of the evolution of the solution, there will be a set of KSARs waiting to be processed. This set of KSARs is called the *Global Agenda.*

Definition 4.2 : Global Agenda

$$\mathrm{Agenda}_{\mathrm{G}} \subset \{< P_i, K_j >, \forall\, i, j\}$$

We define the global agenda as a generic set with no predefined ordering. In some systems this agenda presents a total or partial ordering imposed by the particular implementation of the system's scheduler. As this is an implementation issue, we prefer to leave the ordering specification to that stage.

From the assumptions described earlier, we can identify three places where control decisions should be made. These are the blackboard, the knowledge sources and the desks. We define a control function that embodies the control knowledge needed at each one of these points.

First, a modification to a blackboard element (bbel), say 'a', may trigger a set of knowledge sources. We will call *Problem Agenda* the set of KSARs produced. Note that this set of KSARs is a subset of the global agenda.

Definition 4.3 : Problem Agenda

$$\mathrm{Agenda}_{P_a} = \{< P_a, K_j > | < P_a, K_j > \in \mathrm{Agenda}_{\mathrm{G}}\}$$

At this point, we must decide which knowledge source activations should proceed. A control decision at the problem level allows the definition of default knowledge sources. That is, knowledge sources that trigger when no other triggers. This mechanism also allows the inhibition of knowledge source triggering that can be used as a form of focusing their execution. This control decision is carried out by a *Blackboard Control Function* (**BBCf**). This function takes a problem agenda and returns a subset of it.

Definition 4.4 : Blackboard Control Function

$$\text{BBCf}_a : \mathbf{P}\,\text{Agenda}_{P_a} \rightarrow \mathbf{P}\,\text{Agenda}_{P_a}$$

The expression $\mathbf{P}\,X$ represents the power set of set X. This means that the BBCf is a function that receives a subset of $Agenda_{P_a}$ and returns another subset of it.

Second, one knowledge source can be triggered by more that one bbel at the same time. That is; there can be more than one problem to be solved by a particular knowledge source '*b*' at a given time. We will call *Knowledge Source Agenda*, the set of KSARs waiting to be processed by a given knowledge source.

Definition 4.5 : Knowledge Source Agenda

$$\text{Agenda}_{K_b} = \{< P_i, K_b >| < P_i, K_b > \in \text{Agenda}_\text{G}\}$$

At this point, we must decide which problem or problems should be solved first. This control decision will also help us focus on particular problems as well as follow different strategies, and provide better control over resource utilisation. This decision is made by the associated *Knowledge Source Control Function* (**KSCf**) which takes a knowledge source agenda and produces the chosen KSAR.

Definition 4.6 : Knowledge Source Control Function

$$\text{KSCf}_b : \mathbf{P}\,\text{Agenda}_{K_b} \rightarrow \text{Agenda}_{P_a}$$

Third, and last, there can be conflicts on the utilization of the desks. That is, at a given moment, there may be a group of knowledge sources that may want to execute on a given processor 'c'. The desk agenda will be formed by the chosen KSARs of each of the knowledge sources that wish to work there. We will call this agenda, the *Desk Agenda.*

Definition 4.7 : Desk Agenda

$$\text{Agenda}_{\text{D}_c} = \{< P_i, K_j >| \,\forall j(K_j \text{ runs in } \text{D}_c) \wedge (< P_i, K_j > = \text{KS}_j\text{Cf}(\text{Agenda}_{K_j}))\}$$

Here we must decide which knowledge source should execute first. This decision will be made taking into account which knowledge sources are competing, as well as the importance of the problems they want to solve. This will be carried out by the *Desk Control Function* that takes a desk agenda and returns the chosen KSAR. This KSAR will then be sent to its associated knowledge source for processing.

Definition 4.8 : Desk Control Function

$$\text{DeskCf}_c : \mathbf{P}\,\text{Agenda}_{\text{Desk}_c} \rightarrow \text{Agenda}_{\text{Desk}_c}$$

Each of the control functions introduced above works as a scheduler acting on a restricted domain. There will be one BBCf per group of similar problems (one per blackboard level), one KSCf per knowledge source, and one DeskCf per desk. These schedulers could receive directions from a global control plan stored in the blackboard, just as BB1 does. This would open the area of designing global control plans and defining the mapping to local strategies and heuristics.

4.3.2 The Blackboard

One of the central ideas of the ABB model is that information should only be moved around if it is strictly necessary. This means that information should be as close as possible to where it is needed.

In blackboard systems, as well as in other architectures, there are two fundamentally different types of processing being performed. First, there is the high level, usually symbolic, processing that is carried out by the knowledge sources. For this, the information about the problem to be solved (KSAR) and the knowledge stored in the knowledge source itself are needed. In blackboard systems, it is assumed that the knowledge sources have considerable knowledge of restricted domains. It is then much better to move the problem information to it than to move the knowledge source experience to the problem. This is how blackboard systems normally work.

Second, in all parallel systems, and specially in real time systems, there is a large amount of low-level information that must be transformed and processed before it can be used for knowledge source consumption. In traditional blackboard systems this processing is performed, either by very small knowledge sources, or by special non-blackboard entities. The knowledge needed to perform such transformations is generally small and well defined. The information, however, is usually received in large quantities. In this case it is better to move the knowledge to the problem than vice-versa.

The ABB model clearly separates these two kinds of processing. The high-level knowledge is kept in the knowledge sources, and the low-level data processing and transformation is kept in the blackboard itself. The model provides the architectural mechanisms for specifying this low-level data processing. The ABB blackboard is an active entity within the model that is capable of processing on its own. Whenever new information enters the blackboard, there can be an access oriented process that executes. This process is capable of modifying existing information, as well as producing new information.

The ABB blackboard holds information in blackboard elements (*bbels*). These bbels have the following structure:

- A set of characteristics that describe the type of information they contain.
- A set of operations called *Guards*, evaluated before the bbel information is accessed. The data access is allowed only if all the guards succeed.
- A set of operations called *Triggers*, that are evaluated whenever a characteristic is modified.

The guards allow the implementation of security schemes. The triggers provide the same access oriented capabilities as found in some frame languages such as KRL [BW77] and FRL [RG77], and in LOOPS [SBK86]. Triggers are usually called "when changed" methods or daemons.

A *level* is defined as a set of bbels with a number of characteristics in common. A given level can be described by the set of characteristics present in all the bbels it contains. The ABB levels can be divided in further levels. The set of characteristics common to all the bbels within a level will be the set of characteristics defined by the parent level plus some others, forming some sort of specialisation of the parent level. This defines a level hierarchy with inheritance of characteristics. The blackboard itself is the top level of the hierarchy.

4.3.3 Knowledge Sources

The ABB knowledge sources, as the experts in the blackboard metaphor, operate in two steps. First, they wait for something interesting to appear on the blackboard. The specification of these "interesting" conditions will be called *Triggering Conditions*. Second, when its triggering conditions are met (the knowledge source is *triggered*), the knowledge source will perform appropriate actions that may involve accesses and modifications to the blackboard. We will call these actions, the *Body* of the knowledge source.

On traditional blackboard systems, knowledge sources are supposed to use the blackboard as the only means of interacting between themselves; however, they were allowed to communicate with the system environment by performing I/O either with the user or with remote systems. This forces the knowledge source implementor to think in terms of several different interaction protocols.

In the ABB model, the knowledge sources can only interact with the blackboard. The blackboard will then provide the interface for interacting with the environment

in a transparent manner. This unifies all knowledge source interaction, simplifying their coding. For example, we could think of having an I/O level in the blackboard representing a particular window manager, say; by creating a new bbel on this level, a knowledge source may generate a new interaction window that can then be used for user consultation.

4.4 ABB Model Specification

In this section, we will give a formal description of the ABB model, using the Z specification language [Spi89]. In Appendix D, we will provide a brief introduction to the Z specification language. We advise the reader to get familiarised with its notation, as we will be using it throughout the rest of this chapter. We will not be defining new notation other than the specification function and constant definitions. All notation corresponds to the Z specification language. See [Cra91] for the formal specification in Z of the CASSANDRA blackboard system.

In this section, we will guide the reader through the ABB model specification. For clarity, we will not present all the properties that the different objects specified must satisfy. We will present their structure and functionality. In Appendix E, we present a fully detailed ABB model specification.

The specification of the system as a whole will be called *ABBSystem*. This will be made up of a global blackboard, a set of knowledge sources and a set of desks. These three components will be described by the *BlackBoard*, *KS*, and *Desk* specifications. The form and structure of these elements will be defined later. For the moment, we will treat them as Z basic types. This just means that they are to be regarded as sets of generic objects or elements.

$[ABBSystem, BlackBoard, KS, Desk]$

Within the system there is the need to provide names for the different elements, be they blackboard elements, knowledge sources or desks. We will assume the existence of a set of names called *NAME*. The ABB system components will have composite names formed by sequences of *NAME* elements ($\text{seq}\, NAME$). These sequences will be called *PATH*'s, defined using the Z abbreviation definition ($==$). They will be used, for example, to name the different blackboard object in a way that reflect the hierarchical structure of the blackboard. We can also define the absence of a path, or the null path, to be an empty sequence ($\langle\rangle$).

$[NAME]$
$PATH == \text{seq}\, NAME$
$NullPATH == \langle\rangle$

The information and knowledge needed by the different elements of the model is symbolised by the *INFO* set. The *NullINFO* element represents the absence of information.

$[INFO]$
$NullINFO \in INFO$

Within the ABB system, there is also the need to manipulate information associated with the different system components. This is useful for controlling blackboard access and knowledge source execution, for example. Each system component will have some information that may change with time or in certain situations. We will associate a set of information with each system component. We represent this by a function ($\rightarrow$) that associates information sets (*finsetINFO*) to system component names (*PATH*). The only restriction on such functions is that all information coming from different system elements must be different, or as we express on the Z expression below, if the information of two system elements is equal, then they are the same element. This can be easily accomplished by including the element's path in all its information. The symbol $\bullet$ is the Z notation for "*Such That*". The expression $\mathbb{F}\, X$ represents the set of all finite subsets of X.

$InfoOf : PATH \rightarrow \mathbb{F}\, INFO$
$\forall\, p_1, p_2 : PATH \bullet \exists\, i_1 : InfoOf(p_1);\; i_2 : InfoOf(p_2) \bullet (i_1 = i_2 \Rightarrow p_1 = p_2)$

The ABB model contemplates the interaction, via a blackboard, of a set of experts working at desks. The experts are represented by knowledge sources executing on desks, and interacting via a global data structure called the blackboard. When performing their tasks, the knowledge sources and the other ABB system components perform operations on the system. These operations can query the status of the system, or can actually produce a change to it. We specify them using the Z generic constant syntax. A system operation (*SystemOp*) is a partial function that receives the current state of the system and returns some response.

$$\begin{array}{l} [X] \\ \hline SystemOp : ABBSystem \twoheadrightarrow X \end{array}$$

This response can be boolean (the result of a test), a system (when its state is modified), or any other information.

$$TestOp == SystemOp[\{true, false\}]$$
$$ModifyOp == SystemOp[ABBSystem]$$

We will represent any given procedure involving system operations as a sequence of those operations. These procedures will serve to specify the actions of the different system elements.

$$Procedure == \mathrm{seq}\, SystemOp$$

4.4.1 Knowledge Sources

The ABB knowledge sources perform three basic functions. These are:

Identify: The recognition of problems that fall within the area of expertise of the knowledge source.

Prioritise: The ordering of the pending problems, according to their importance at a given moment. This is part of the control structure of the ABB model.

Solve: The actual resolution of a problem.

The knowledge about how to identify suitable problems to be solved by the knowledge source, is encoded within a condition. Given the current state of the blackboard, this condition succeeds if there is one such problems. The knowledge about how to solve a particular problem is encoded into the body of the knowledge source. Given some information about the problem that has been identified by the trigger condition, the knowledge source body solves it and changes the state of the blackboard to reflect its solution.

The communication between the trigger condition and the body of the knowledge source is done via a *KSAR*. This *KSAR* will contain the *PATH* of the blackboard element where the problem was identified (*Path*), some information about the knowledge source that it activates (*KsInfo*), and a finite set of fields specifying the information about the problem needed by the knowledge source body (*Fields*). The actual contents

of the information specified by each field is also kept in the KSAR. We can specify this by a partial function ($\rightarrow\!\!\!\!\!\!+$) that associates each field name with its content (*Content*). We specify the KSAR using the Z schema syntax. This defines the object structure and a set of conditions that this structure must satisfy. In the case of *KSAR*, the domain (dom) of the content function must be the set of fields of the KSAR. This means that each field must have a content and that there will be no more contents than fields.

$$
\begin{array}{l}
\textit{KSAR} \\
\hline
Path : PATH \\
KsInfo : INFO \\
Fields : \mathbb{F}\ NAME \\
Content : NAME \rightarrow\!\!\!\!\!\!+ INFO \\
\hline
\mathrm{dom}\ Content = Fields
\end{array}
$$

The initial state of a *KSAR* is one in which it has no contents, or rather, in which its fields contain *NullINFO*. Whenever a new KSAR is generated, it starts up in this initial state.

$$
\begin{array}{l}
\textit{NullKSAR} \\
\hline
KSAR \\
\hline
Content = \{\ n \mapsto NullINFO \mid n \in Fields\ \}
\end{array}
$$

Within the ABB model, control is performed by a set of functions that operate mainly on sets of KSARs. We can identify two kinds of such functions. First, there are a number of (possibly partial) functions that take a finite set of KSARs and return a modified subset of it. These functions will be called *FilterCFn*. Second, there are the functions that take a set of KSARs and return one of them. We will call these functions, *SelectCFn*.

$$FilterCFn == \mathbb{F}\ KSAR \rightarrow\!\!\!\!\!\!+ \mathbb{F}\ KSAR$$

$$SelectCFn == \mathbb{F}\ KSAR \rightarrow KSAR$$

The individual problem identification is made by the trigger condition via a set of activation conditions (*ActivationCond*). Each one of these conditions contains a blackboard operation (*ActivationOp*) that tests for a specific type of problem and,

when it succeeds, returns an activation record (*Test*).

$$ActivationOp == BlackBoard \rightarrow KSAR$$

The activation conditions are targeted to a specific blackboard level, represented by its *PATH* (*BBlevel*). It also defines the structure of the KSARs produced by its *Test* by specifying the set of fields that they should have (*Fields*), and the information about the knowledge source (*KsInfo*).

```
_ActivationCond______________________
| Test : ActivationOp
| BBlevel : PATH
| Fields : F NAME
| KsInfo : INFO
|________
|_____________________________________
```

There are, also, a number of special activation conditions that are only satisfied when the system starts. These are called *StartCond*'s, and are characterised by not being attached to any blackboard level. This means thet their *BBlevel* is the *NullPATH*. The *StartCond*'s are used to specify the initial knowledge sources that will begin the problem solving activity of the system.

```
_StartCond___________________________
| ActivationCond
|________
| BBlevel = NullPATH
|_____________________________________
```

We can now specify the knowledge source trigger condition. A given trigger condition (*TriggerCond*) will contain a set of activation conditions (*Conditions*), an internal structure (*Fields*), and a reference to its knowledge source (*KsName*). The structure of the *Conditions* will be inherited from the trigger condition. The trigger condition schema is then:

```
_TriggerCond_________________________
| Conditions : F1 ActivationCond
| Fields : F NAME
| KsName : PATH
|________
|_____________________________________
```

When an *ActivationCond* within a *TriggerCond* succeeds, a *KSAR* is produced and submitted to the system for consideration. Eventually, the system may decide to process it. The processing of the KSARs is performed by a procedure called the *Body* of the knowledge source. This procedure is where the knowledge source actually solves the problems within its domain. The body procedure (*KSBodyProc*) takes the KSAR to be processed and, as a result, it can modify the state of the system (add or modify information in the blackboard, for example).

$$KSBodyProc == KSAR \nrightarrow Procedure$$

The knowledge source is then defined by an unique name (*Name*), a trigger condition that identifies its problem domain (*Trigger*), a knowledge source control function that prioritises its work (*KSCf*), and the body that actually solves the problems (*Body*). The list of KSARs, representing the problems that remain to be solved is kept in a local agenda (*KSAgenda*). The trigger condition keeps a reference to the knowledge source, represented by its name. The control function must be defined for all possible local agendas. The body of the knowledge source must only accept KSARs produced by it. Finally, the knowledge source local agenda is a set of KSARs produced by its trigger condition.

$$\begin{array}{l}
\underline{\quad KnowledgeSource \quad} \\
Name : PATH \\
Trigger : TriggerCond \\
KSCf : FilterCFn \\
Body : KSBodyProc \\
KSAgenda : \mathbb{F}\ KSAR \\
\hline
\end{array}$$

The knowledge sources that have one *StartCond* within its trigger conditions will be called initialisation knowledge sources (*InitKS*'s). They are activated when the system starts, and are specially suited for performing all the initialisations that may be needed by it.

$$\begin{array}{l}
\underline{\quad InitKS \quad} \\
KnowledgeSource \\
\hline
\exists\, c : StartCond \bullet c \in Trigger.Conditions \\
\hline
\end{array}$$

4.4.2 Blackboard

The ABB blackboard is a structure used to store information of global interest. This information is structured in blackboard elements (*BBels*) stored in different levels within a level hierarchy. A bbel is a structure with a name (*Name*), and a set of fields (*Fields*) with associated contents (*Contents*). This field-content association is specified via a partial function, whose domain (dom) must be the bbel's field set. The type of a particular bbel is defined by the number and type of its fields.

$$
\begin{array}{|l}
\textit{BBel} \\
\hline
Name : PATH \\
Fields : \mathbb{F}\ NAME \\
Contents : NAME \rightharpoonup INFO \\
\hline
\operatorname{dom} Contents = Fields \\
\hline
\end{array}
$$

The initial state of the bbels of any given type is defined by the *NullBBel* schema. This defines a bbel whose fields have no information defined.

$$
\begin{array}{|l}
\textit{NullBBel} \\
\hline
BBel \\
\hline
Contents = \{\ n \mapsto NullINFO \mid n \in Fields\ \} \\
\hline
\end{array}
$$

There are a number of procedures associated with all the bbels of a given type. These procedures are, an initialisation procedure (*InitBBel*), a set of *Guards*, and a set of *Triggers*. The initialisation procedure is used to define a number of default values for bbels (*Defaults?*). It is evaluated whenever a new bbel is created. The default values are specified as a partial function, similar to the *Content* function. It should be defined for some (maybe all) of the fields of the bbel. The result of this procedure is a new bbel whose contents contain the defaults. In terms of functions, this means that the defaults functionally overrides the previous contents.

$$
\begin{array}{|l}
\textit{InitBBel} \\
\hline
\Delta BBel \\
Defaults? : NAME \rightharpoonup INFO \\
\hline
\\
\hline
\end{array}
$$

When a new bbel is created, a *NullBBel* is generated and then it is initialised with *InitBBel*. This is specified using the Z schema sequential composition ($\fatsemi$) as follows.

$$InitialBBel \mathrel{\widehat=} NullBBel \fatsemi InitBBel$$

The guards are operations that are evaluated whenever a system component tries to access a bbel within the blackboard. They consist of a test about the status of the blackboard, upon which they decide whether to grant or deny the access to a bbel. This facility can be used to hide some areas of the blackboard to specific blackboard system elements. This allows the implementation of security schemes.

$$Guard == INFO \rightarrow (BlackBoard \rightarrow \{ \ true, false \ \})$$

The triggers are operations evaluated after a change has been made to the blackboard. Given the bbel modified, the trigger may change the state of the system.

$$Trigger == BBel \rightarrow ModifyOp$$

Both guards and triggers have associated names that identify them unequivocally. The set of guards and triggers of a particular type of bbel can be seen as partial functions that map the different names to the actual guards or triggers.

$$Guards_{bbel} == NAME \nrightarrow Guard$$

$$Triggers_{bbel} == NAME \nrightarrow Trigger$$

Bbels of the same type are grouped together within a level. The level itself can be seen as the definition of a particular type of bbels. An ABB level is defined by a structure that holds its name (*Name*), a set of fields that define the type of the bbels it holds (*Fields*), a bbel initialisation procedure (*Init*), a set of triggers (*Triggers*), a set of guards (*Guards*), the actual set of bbels stored in it (*BBels*), the blackboard control function (*BBCf*), and the set of activated triggers, waiting to be evaluated (*Eval*).

LevelStructure

$Name : PATH$
$Fields : \mathbb{F}\ NAME$
$Init : InitialBBel$
$Triggers : NAME \rightarrowtail Trigger$
$Guards : NAME \rightarrowtail Guard$
$BBels : \mathbb{F}\ BBel$
$BBCf : FilterCFn$
$Eval : \mathbb{F}\ ModifyOp$

The name of all the bbels within a level is formed by the composition of the name of the level and a bbel name, unique within it. The level also defines the structure of the bbels stored in it. That is, the fields of the bbels are defined to be the same fields as its level. The level initialisation procedure must define the contents of those fields defined by the level. It can define defaults for all or part of the level's fields. The blackboard control function operates on KSARs generated in this level. The default blackboard control function is the identity function.

Note that there is no level-wide agenda. This is because the blackboard agenda is specific to each bbel that has been modified. A given level may have several agendas. Furthermore, it is not necessary to store these agendas. When a bbel is modified, the relevant activation conditions can be evaluated. The set of KSARs they return can then be processed by the level control function and the resulting set of KSARs can be sent directly to the appropriate knowledge sources. This is possible because the blackboard agendas are produced by knowledge source activation conditions, and they do not modify the state of the blackboard.

An ABB level is then a tree-like structure that has a level structure on each node and a finite set of sub-levels. We specify it with the Z syntax definition. This definition is recursive as the levels can be nested arbitrarily.

$$Level ::= LevelStructure \times \mathbb{F}\ Level$$

In order to be properly defined, the level must have a number of characteristics that define the form of its components.

The name formation rule for the sublevels is identical to the one for the bbels within the level's structure. This ensures that the name of any particular level or bbel

reflects its position within the blackboard level hierarchy. Also, the names of the bbels and sublevels within a level must be all unique.

A given level partially defines the structure of its sub-levels. They inherit the fields of their parent level. That is, the fields of each sub-level are the fields of the parent level plus some other specific to itself. The structure of each sub-level initialisation procedure is partially defined, also, by its parent level. The defaults defined in the level initialisation procedure will be used unless overridden by the sub-level.

The blackboard control function is also inherited, unless the sub-levels override it. That is, the sub-level control function is defined by the parent's function in all of the elements of the domain, except where defined locally.

Finally, the guards and triggers are also inherited. The sub-levels can redefine the parent's guards and triggers by defining new procedures with the same associated names.

The ABB blackboard is a level identical to any other. Its only particularity is that it is the top of the level hierarchy. As such, its associated path is a sequence of only one name.

$$BlackBoard \subset Level$$
$$\forall\, l : Blackboard \bullet \exists\, n : NAME \bullet Struct(l).Name = \langle n \rangle$$

When defining a blackboard for a particular application, we will place the common global structure and defaults in the blackboard level. Given the problem domain information dependencies, this blackboard will have as many sub-levels as necessary. The sub-levels will inherit the global structure which will be extended with particular specialisations. These sub-levels can be further subdivided to suit the problem structure.

The ABB blackboard defines a data and knowledge hierarchy based on structure and procedure inheritance. This is similar to the hierarchy found in most object-oriented languages and systems. The ABB root object class is the blackboard and all other classes of objects are derived from it.

4.4.3 Blackboard Operations

When any ABB system element needs to access or modify the blackboard structure, it must use the blackboard operations available. The ABB system defines four basic blackboard operations that allow the creation, access, modification and deletion of

blackboard elements. All of them first evaluate the bbel's guards in order to check if the operation is allowed, and then perform the operation.

After the operations are executed, the state of the blackboard may have changed. If this is the case, the triggers of the bbels that took part on the operation must be activated. The operations will place these triggers in the level evaluation list.

The creation of bbels is performed via the *LCreateBBels* operation. It takes information about the system element performing the operation (needed for guard evaluation), a set of bbel names (paths) and a level. If the guards allow the modification, the operation returns a modified level with the new bbels added.

$$LCreateBBels : INFO \times \mathsf{F}\ PATH \times Level \rightarrow Level$$

The *LReadBBels* function allows the access to the blackboard elements. It takes a set of bbel names (paths) and a level, and if the guards succeed, it returns the set of bbels with the indicated names.

$$LReadBBels : INFO \times \mathsf{F}\ PATH \times Level \rightarrow \mathsf{F}\ BBel$$

The modification of bbels is performed via the *LModifyBBels* function. The modifications are represented by a bbel with the same name as the one we want to modify. Its fields must be a subset of those of the target bbel. Its contents will replace those on the target bbel. In a way similar to the creation function, *LModifyBBels* takes information about the element performing the modifications, the set of bbels to be modified and a level. If the guards succeed, it returns the given level with the modifications done.

$$LModifyBBels : INFO \times \mathsf{F}\ BBel \times Level \rightarrow Level$$

Finally, the *LDeleteBBels* function allows the deletion of bbels from the blackboard. It takes a list of bbel names and a level, and returns the level with the bbels with those names removed.

$$LDeleteBBels : INFO \times \mathsf{F}\ PATH \times Level \rightarrow Level$$

4.4.4 Desks

The desks are the ABB system components that represent the place where the experts (knowledge sources) perform their work. They will hold knowledge about how to administer their processing resources. This knowledge is embodied in the desk control function. This function decides when a given knowledge source should execute.

A desk is formed by a name that identifies it uniquely (*Name*), a set of the names of the knowledge sources that may execute in it (*KSnames*), the list of KSARs of those knowledge sources ready to be evaluated (*DeskAgenda*), and the local control function (*DeskCf*). It also holds a reference to the knowledge source that is currently being evaluated (*ActualKS*), and the actions or operations that remain to be evaluated (*Eval*). Each knowledge source will be able to execute in at least one of the desks of the system.

Desk

$Name : PATH$
$KSnames : \mathbb{F}\ PATH$
$DeskAgenda : \mathbb{F}\ KSAR$
$DeskCf : SelectCFn$
$ActualKS : PATH$
$Eval : Procedure$

4.4.5 The ABB System

The ABB system, as we described in the Section 4.3, consists of a global blackboard (*BBoard*), a set of knowledge sources (*KSources*), and a set of desks (*Desks*). These system elements interact with each other in such a way that the system as a whole works toward the solution of a problem in the domain.

There are a number of conditions that the system elements must satisfy, in order to insure the proper execution of the system. In order to avoid possible confusion, we define that all knowledge sources and all desks must have different names. All knowledge sources must be able to be evaluated in, at least, one desk. Finally, there must be at least one initialisation knowledge source.

```
┌─ ABBSystem ──────────────────────────
│ BBoard : BlackBoard
│ KSources : F KS
│ Desks : F Desk
└──────────────────────────────────────
```

When the system starts its operation only the initialisation knowledge sources are triggered. Their activation records are stored in their respective desks. The knowledge sources then have empty agendas. The blackboard, in this initial state, is empty and totally inactive. That is, it has no triggers waiting to be evaluated. The ABB system initial state is called *InitialABBSystem*.

$InitialABBSystem \in ABBSystem$

From the initial state, the ABB system will begin operation. One of the desks holding KSARs will choose one to be evaluated. The body of the appropriate knowledge source will be taken by it and it will begin executing its operations. Meanwhile another desk can be doing the same. When the initial knowledge sources execute, they will be changing the state of the blackboard. This will provoke the activation of other knowledge sources, as well as the evaluation of blackboard triggers. All this execution will produce further knowledge source and blackboard trigger activations.

This process will continue until the final state is reached. We define this final state as the one in which there are no more KSARs queued in any of the agendas, and all evaluations are finished. We call this the *StoppedABBSystem*.

$StoppedABBSystem \in ABBSystem$

This final state can be reached naturally by the system by exhausting all possible solution paths, or it can be forced by a knowledge source or blackboard trigger when it identifies the domain problem termination condition.

4.4.6 Global System Operations

The ABB system defines a number of operations on the system as a whole that are available to the user. These operations are used to form the knowledge source body procedures, as well as blackboard triggers. These operations implement the different blackboard manipulation functions, as well as a global stop function.

The general procedure is to modify the blackboard using the appropriate level operation, obtain the KSARs generated and add them to the appropriate knowledge source agenda. The guard checking, and trigger generation are performed by the level operation.

There will be five global system operations available to the user. These are *BBCreateOp*, *BBModifyOp*, *BBReadOp*, *BBDeleteOp*, and *BBStopOp*. The first one allows the creation of new blackboard elements. All *BBCreateOp* will have exactly one set of paths associated. These paths indicate where the new bbels must be created.

$$BBCreateOp : \mathsf{F}\ PATH \rightarrow ModifyOp$$

The *BBModifyOp* operation allows the modification of blackboard elements already in the blackboard. All of these operations will have exactly one set of associated bbels. These bbels represent the modifications. Their structure is defined by the level bbel modification function (*LModifyBBels*). That is, they must have names of bbels already present in the blackboard, and their fields must be a subset of those bbels they modify.

$$BBModifyOp : \mathsf{F}\ BBel \rightarrow ModifyOp$$

The third operation permits access to the information held in the blackboard. This operation does not change the contents of the blackboard. It does, however, change the state of the system. It may influence the procedure where it originated by changing some operations, or simply instantiating relevant information.

All blackboard access operations (*BBReadOp*) will have a specific set of paths associated that indicate the bbels it will look for. As a result of the access, the system will be unchanged, except for the desk where the operation originated.

$$BBReadOp : \mathsf{F}\ PATH \rightarrow ModifyOp$$

The next operation allows the elimination of information held in the blackboard. For each of these operations there is one particular set of paths referencing the bbels that will be removed by it. The resulting system will have the given bbels removed (if the guards allow it). There may be a number of knowledge sources that are activated by these changes. The resulting knowledge sources will have the KSARs generated added to their respective agendas.

$BBDeleteOp : \mathbb{F}\ PATH \rightarrow ModifyOp$

The last user operation allows to stop the system at any moment. Usually, this will be done whenever the termination condition is detected by a knowledge source body or blackboard trigger operation. This condition will be met whenever the problem domain is solved, or it is known that no solution can be found.

$BBStopOp : ModifyOp$

We can now define the set of operations available to the user as the union of the operations described above.

$$UserOp == BBCreateOp \cup BBModifyOp \cup BBReadOp \\ \cup BBDeleteOp \cup BBStopOp$$

In all ABB blackboard systems, the knowledge source bodies will be formed only by user operations. The blackboard trigger operations will be any user operations, except read operations. This is because the blackboard triggers are not procedures, and the result of such access would be lost.

We now have the functional specification of the different elements of the active blackboard architecture, and of the operations between them. These operations are stored in different parts of the system, and when they are evaluated, produce changes to the general system state.

However, we still need global system operations that actually evaluate the different user operations in the different levels of the system. We will need four such operations.

The *ExecBBOp* evaluates user operations held in the blackboard. These operations are the result of triggers activated by any of the blackboard modification operations. It does nothing if there are no user operations anywhere on the blackboard. If there is at least one operation, then it evaluates it. The new system will be the actual system with the operation applied to it, and with the operation taken out of the evaluation list.

$ExecBBOp : ModifyOp$

The second operation selects and prepares a knowledge source body to be evaluated. This operation will do nothing if there are no free desks. If there is at least

one free desk, select one and choose one KSAR within its agenda, using its control function. Take the body of the associated knowledge source, instantiate it with the chosen KSAR and place it in the evaluation list.

$$DeskPrepareOp : ModifyOp$$

The third operation evaluates operations held in the desks, product of the previous operation. This operation will do nothing if there are no user operations waiting to be evaluated in any desk. If there is one such operation, it evaluates it. The new system will be the actual system with the operation applied to it, and with the operation taken out of the evaluation list.

$$ExecDeskOp : ModifyOp$$

Finally, the fourth operation schedules KSARs from the knowledge source agendas to the appropriate desk agenda. The other movements of KSARs between the different agendas is already performed by the blackboard user operations. This operation will do nothing if there are no KSARs queued in any of the knowledge source agendas. If there is at least one non-empty agenda, it will choose a subset of its KSARs using the knowledge source control function, and will place them within the agenda of its associated desk.

$$MoveKSARsOp : ModifyOp$$

The global system operations are, then, the union of those four operations.

$$ABBSystemOp = ExecBBOp \cup ExecDeskOp \cup DeskPrepareOp \cup MoveKSAROp$$

4.4.7 Distributed Control

In the ABB model, control of knowledge source activation is performed in several steps, evaluated within several system operations. This step by step evaluation allows the distribution of the control decisions among all system components. Figure 4.1 shows a simple representation of the ABB control cycle.

KSARs are generated by the knowledge source trigger conditions as a result of a modification in the blackboard. Blackboard modifications are produced by *UserOp*'s which are evaluated by *ExecDeskOp* and *ExecBBOp*. Within the *UserOp*'s, KSARs are created, processed by the blackboard control functions, and sent to their knowledge

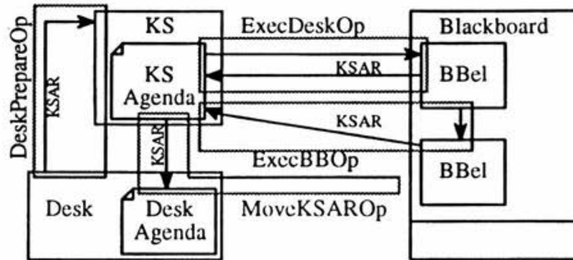

Figure 4.1. ABB Control Cycle

source's agenda.

Once in a knowledge source agenda, the ksars are evaluated by the knowledge source control function, and sent to the associated desk agenda. This is done by the *MoveKSAROp*. When the desk finished evaluating a knowledge source body, it evaluates its agenda using the desk control function, and schedules a new KSAR for evaluation. This repeats the cycle.

Note that in a parallel implementation of the model, we can have one such cicles for each desk in the system, and each cycle could be in a different stage of evaluation.

4.4.8 ABB System Operation

The global system operation, then, results in the application of successive *ABBSystemOp*'s. We can define a transition function $\rightarrow$ that relates one system state with the one resulting from the application of a given system operation.

$$\rightarrow : ABBSystem \times ABBSystemOp \times ABBSystem$$

$$\forall\, S, S' : ABBSystem;\ \odot : ABBSystemOp \bullet$$
$$S \xrightarrow{\odot} S' \Leftrightarrow S' = \odot(S)$$

By convention, we will write $S \xrightarrow{\odot_1} S' \xrightarrow{\odot_2} S''$, meaning $S \xrightarrow{\odot_1} S' \wedge S' \xrightarrow{\odot_2} S''$. In general,

$S \xrightarrow{\odot_1} \ldots \xrightarrow{\odot_n} S_n \Leftrightarrow S \xrightarrow{\odot_1} S_1 \wedge \ldots \wedge S_{n-1} \xrightarrow{\odot_n} S_n$

Let S be any blackboard system state. This state should be the result of applying several *ABBSystemOp*'s to a corresponding initial state. Let S_i be it's initial state. S will be a consistent state of the system if there is a sequence of *ABBSystemOp*'s that,

after being applied, result in S.

$$\textit{Coherent} : \textit{ABBSystem}$$

$$\forall S : \textit{ABBSystem} \bullet \exists S_i : \textit{InitialABBSystem} \mid S_i = \textit{InitialSystem}(S) \bullet$$
$$\textit{Coherent}(S) \Leftrightarrow (\exists \odot_1, ..., \odot_n : \textit{ABBSystemOp} \bullet S_i \xrightarrow{\odot_1} ... \xrightarrow{\odot_n} S$$

This property says that a given system state is coherent if there exists at least one sequential ordering of its operations with which we can reach it from the system's initial state. That is, it forces some sort of serialisation of its operations.

A system may be in a incoherent state when there is a modification of the blackboard and there are blackboard triggers or knowledge source activation procedures that fail to activate.

A system may also be incoherent if there were two consecutive blackboard modifications that were interleaved. For example, suppose that there are two blackboard operations, $\odot_1$ and $\odot_2$, and that they both modify bbels b_1 and b_2. We will denote b_1^1, b_2^1, and b_1^2, b_2^2 the result of the $\odot_1$, and $\odot_2$ modifications, respectively. If we apply $\odot_1$, and then $\odot_2$, we will end up with b_1^2, and b_2^2. If we do it the other way around, we obtain b_1^1, and b_2^1.

Suppose, now, that we are working in a parallel environment, and that while $\odot_1$ is modifying b_1, $\odot_2$ is modifying b_2. Once they modify the first bbel they go and modify the other. When the modifications finish, we may end up with b_1^2, and b_2^1. This combination is impossible to duplicate using $\odot_1$ and $\odot_2$ sequentially.

A given implementation of the ABB specification must provide the functionality described here, and must comply with its properties. This includes the coherence property. The implementation must insure that it does not fall in an incoherent state. The way this is achieved is irrelevant to the specification. Note that the implementation could be sequential or parallel. Those considerations are purely implementation issues.

4.5 Summary and Conclusions

In this chapter we have presented the active blackboard (ABB) model for problem solving. This is a blackboard model based on the "Blackboard, Experts and Desk" metaphor.

We consider the new metaphor, a variation of the traditional "Blackboard and Experts" metaphor to be a significant contribution, as it allows to visualize in a simple

and intuitive manner, the complex interactions that occur in a parallel blackboard system.

The other major contribution made in this chapter is the ABB model itself, together with its specification. This new model defines a new class of blackboard systems with a distributed control structure and a rich blackboard structure. These characteristics will allow the efficient use of parallel and distributed computer architectures.

The ABB model satisfies the design characteristics presented in the introduction of this chapter, and lays out the basis for a blackboard system implementation that meet the desirable implementation characteristics.

In the next chapter, we will present this implementation, the ABB parallel prototype, and in Chapter 6 we will analyse its performance.

Chapter 5

The ABB System Implementation

5.1 Introduction

In this chapter we will present a parallel ABB system prototype implemented on the EPCC Meiko Computing Surface, a transputer based multiprocessor. This implementation was made using the ANSI C language, with the CSTools communication library. This prototype provides an implementation of the ABB model introduced in the previous chapter and will satisfy the desired implementation characteristics listed in the introduction to Chapter 4.

The design and implementation of the parallel prototype was made following the object oriented approach. The functionality of the ABB model is also provided in the framework of an object oriented blackboard system.

In the following sections we will present the prototype, showing how it implements the model functionality. We will use the specification of the model, and the implementation of a simple blackboard system demonstrate the prototype characteristics. We will also discuss some implementation details, concentrating on the inter-process communication and concurrency control mechanism employed in the ABB prototype.

5.2 The ABB Parallel Prototype

The ABB system prototype is an object oriented parallel blackboard system shell. In it, all the ABB system components are described using an object-oriented approach. The ABB blackboard is a collection of objects organised by type, and both the knowledge sources and the desks are objects. The user will have the usual mechanisms for object description such as encapsulation and inheritance.

As a shell, the ABB prototype can be used to implement a wide variety of blackboard systems for very different domains. As a guide through the prototype, we will implement a very simple blackboard system, the *Thread Simulation System* (*THR*). This system will be used in the next chapter for the evaluation of the performance of the prototype.

The idea behind the THR system is that all systems, in the process of execution, can follow a number of independent solution paths. These can be the demonstration of different hypothesis, the evaluation of two separate equations, or the solution of several sub-problems. Each of these paths consists of a number of actions that need to be evaluated in a sequential order. These independent paths are called *threads*.

When we evaluate an application in parallel, the maximum speed-up possible will be restricted by the number of application tasks that can be evaluated in parallel. This

number is the number of independent threads that the application has. For example, if an application has twenty threads, the smallest run-time possible, when executing in parallel, will be the run time for the longest thread.

The THR system is extremely simple. It will represent each independent thread by a knowledge source and a blackboard element (bbel). The knowledge source will represent the actions within the thread, and the bbel will be the data being manipulated. Each thread knowledge source will trigger when its thread bbel is modified. When triggered, it will modify the thread bbel, incrementing a counter, and finishes. This modification will cause it to trigger again. This produces a cycle of knowledge source trigger, evaluation, and bbel modification, which will continue until the bbel counter reaches a pre-defined number representing the thread length.

The THR will also have an initialisation/termination knowledge source, and a control bbel. The initialisation knowledge source will create and initialise all thread and control bbels. The control bbel will hold the number of active threads. Initially, this will be the total number of threads. Every time a thread finishes, it will decrement the number of active threads. When the control bbel reaches zero, the initial knowledge source triggers again, and stops the system.

The THR system will have one blackboard level (`BB.THR`) where it stores all thread and control bbels, one initialisation/termination knowledge source (`InitKS`), a number of thread knowledge sources (`ThreadKS`), and any number of desks. The THR system can be configured for any number of threads. This will determine the number of `BB.THR` bbels, and of `ThreadKS`'s.

The implementation design of the ABB prototype will have the three ABB components specified in the model presented in the previous chapter. These are, one global *blackboard*, a set of *knowledge sources*, and a set of *desks.*

The ABB prototype organises these elements in a number of files. These files will follow similar C conventions. The definitions of types and constants will be stored in files with the `.ah` extension. The procedure definitions will be stored in files with the `.ac` extension. Each blackboard level, knowledge source, and desk will be stored in separate files. The file distribution for the THR system is as follows:

BB-Lvl.ah	BB-Lvl.ac	Top BB level description
BB.THR-Lvl.ah	BB.THR-Lvl.ac	THR level description
InitKS-KS.ah	InitKS-KS.ac	Initialisation/Termination KS
ThreadKS-KS.ah	ThreadKS-KS.ac	Thread KS description
THR-Sys.ah	THR-Sys.ac	General desk description

5.2.1 The Blackboard

In the ABB specification, the blackboard is defined as a set of bbels held in a number of nested levels. The laws governing the nesting of levels are very similar to class specialisation, especially the inheritance of its structure and operations. We decided to implement the global blackboard structure as an object hierarchy. This allows us to use the usual object oriented techniques and facilities for the definition of bbels. The blackboard levels are implemented as a group of procedures that handle sets of bbels of the same class.

The system implementor then specifies the structure of the blackboard by defining classes of blackboard objects. The top class will be the BB class. Its structure and methods will be inherited by all the other blackboard classes. In the current ABB prototype, the blackboard objects can only have single inheritance with method overriding.

For the THR example, the top level definition only specifies that all blackboard objects should have a name. It also defines a basic bbel access function that retrieves the bbel's name, the bbel initialisation function, and the default blackboard control function. The top BB level class definition is:

File: BB-LvL.ah

```
BBLVLdefine BB {
     /* Internal Structure */
     char name[NAMELENGTH];
     /* BB Methods */
     BOOL GetName(BB::BBEL * self, char * name);
     BOOL PutName(BB::BBEL * self, char * name);
     BOOL INIT::NameInit(BB::BBEL * self);
     BOOL BBCF::NotifyAll(BB:BBEL * self, LIST * BBelAgenda,
                          LIST ** ToNotify);
}
```

The functions GetName, PutName, NameInit, and NotifyAll are defined in the BB-Lvl.ac file. Note that we have placed some marks before the names of some of those functions. These marks identify special methods. The INIT mark identifies the bbel initialisation method, which will be called when a new BB bbel is created. The BBCF mark identifies the bbel control function. This will we called whenever a bbel is modified and there are knowledge sources that trigger on this modification. It will be given the list of KSAR's generated by the modification, and will produce the list of KSAR's to be sent to their respective knowledge sources. This function can filter the active KSAR's according to its own criteria. The definition of these functions is as

follows:

File: BB-Lvl.ac

```
BOOL BB::GetName(BB:BBEL * self, char * name) {
     strcpy(name, self->name);
     return(TRUE);
}
BOOL BB::PutName(BB:BBEL * self, char * name) {
     strcpy(self->name, name);
     return(TRUE);
}
BOOL BB::NameInit(BB::BBEL * self) {
     self->name[0]=NullCHAR;
     return(TRUE);
}
BOOL BB::NotifyAll(BB:BBEL * self, LIST * BBelAgenda, LIST ** ToNotify) {
     *ToNotify = BBelAgenda;
     return(TRUE);
}
```

These methods are the only legal means of manipulating the bbels. They will be used by the different system components via the blackboard manipulation functions. The `NameInit` function initialises the bbel's name as the null string. The `NotifyAll` function specifies that all triggered knowledge sources should be notified. As such, it does not modify the bbel agenda.

Besides this top level, the THR system has one level called BB.THR. Its name indicates that it is derived from the BB class. It will inherit any structure and methods defined in the BB class. The THR bbels will hold a counter which represent the step number within a thread (thread bbels), or the number of active threads (control bbel).

File: BB.THR-Lvl.ah

```
BBLVLdefine BB.THR {
/* Internal Structure */
     int  Counter;
/* BB.THR Methods */
     BOOL setCounter(BB.THR::BBEL * self, int * num);
     BOOL getCounter(BB.THR::BBEL * self, int * num);
     BOOL decCounter(BB.THR::BBEL * self);
     BOOL INIT::initThread(BB.THR::BBEL * self);
}
```

The `setCounter`, `getCounter`, `decCounter`, and `initThread` functions are defined in the `BB.THR-Lvl.ac` file. They perform the counter assignment, value query,

and decrementing the counter by one. The `initThread` function initialises the counter to zero.

There are two more marks that we can use when defining bbel methods. These are **GUARD**, and **TRIGGER**, and will identify guards, and triggers respectively. The guards will be called before any other method is executed, and they must succeed (return **TRUE**) for the operation to be allowed. The triggers will be performed after a modification has been made.

Once a bbel class is defined, we will be able to create bbels dynamically by using the global blackboard manipulation functions. These dynamically created bbels will be held and managed by the levels. A level, then, represents a set of bbels of the same type. It will be implemented by a program that will manage a group of bbels.

When the system starts all levels will be empty. There will be some bbel classes for which we will not need a level, as we will never create bbels of that class. These classes are normally used as high level concept specifications that are then specialised. We will use the `defineBBLVL` construct for defining a level that will hold and manage dynamic bbels.

The definition of the level is done by merely identifying the type of bbels that the level will hold. All the level structure and components are already defined by the bbel class. Take the level specification introduced in the previous chapter:

LevelStructure

$Name : PATH$
$Fields : \mathbb{F}\ NAME$
$Init : InitialBBel$
$Triggers : NAME \twoheadrightarrow Trigger$
$Guards : NAME \twoheadrightarrow Guard$
$BBels : \mathbb{F}\ BBel$
$BBCf : FilterCFn$
$Eval : \mathbb{F}\ ModifyOp$

The *Name* of this particular level will be the bbel class name (**BB.THR**). The *Fields* are those defined in the class (`name` and `Counter`). The *Init* bbel is the result of applying the **INIT** marked method to a newly created bbel. The *Triggers* and *Guards* are the sets of methods marked with **TRIGGER**, and **GUARD**. The *BBels* is the set of bbels belonging to the **BB.THR** class, created using the blackboard manipulation functions. The blackboard control function (*BBCf*) is implemented by the method marked **BBCF**,

and the *Eval* set is kept internally.

In some applications there are some levels that are bound to become a system bottleneck. This may be because they hold a large amount of bbels, or because a large proportion of the knowledge sources create and use its bbels. For these levels, the user has the option of defining more than one instance of it. This does not duplicates the blackboard level or its contents. Logically, there still exists one level holding bbels. However, physically there will be several places where a given bbel can be stored. BBels will then be distributed among the different bbel stores.

File: BB.THR-Lvl.ah

```
defineBBLVL BB.THR THR_Lvl_1;
defineBBLVL BB.THR THR_Lvl_2;
```

At the moment, the system has an internal procedure that tries to maintain the same amount of bbels in each level instance. Other mechanisms for bbel load balance can be implemented. However, in the current configuration of the system, the user cannot specify an arbitrary load balancing algorithm. It would be interesting to explore the possibility of allowing the user to manipulate and create load balancing mechanisms.

The system implementor has all the versatility of an object oriented language for expressing the structure and functionality of the ABB blackboard. With this feature, the ABB system prototype follows the ABB model specification, and provides a versatile and powerful mechanism for knowledge representation, satisfying in this way one of the desired design characteristics presented in the introduction to Chapter 4.

5.2.2 Blackboard Manipulation Functions

All the ABB system components will be able to access and modify the blackboard. In the ABB specification, it was stated that this has to be done through the use of a number of blackboard manipulation functions. The ABB prototype implements these blackboard manipulation functions as a library of procedures available to all system modules.

The specification defines four such functions:

$$\begin{aligned}
&LCreateBBels : INFO \times \mathsf{F}\ PATH \times Level \rightarrow Level\\
&LReadBBels \ \ : INFO \times \mathsf{F}\ PATH \times Level \rightarrow \mathsf{F}\ BBel\\
&LModifyBBels : INFO \times \mathsf{F}\ BBel \times Level \rightarrow Level\\
&LDeleteBBels \ : INFO \times \mathsf{F}\ PATH \times Level \rightarrow Level
\end{aligned}$$

The ABB prototype defines four blackboard manipulation functions. These are **CreateBBels**, **ReadBBels**, **WriteBBels**, and **DestroyBBels**. All of them specify a number of bbels to be created, read, modified, or removed (**F** *PATH* or **F** *BBel*). In the prototype, each bbel is identified by its path (a sequence of names indicating its position in the blackboard) and its name.

The information given to each operation consists of some data about the module performing it, plus information pertaining the actual operation. In the case of the prototype, the caller's information is added automatically, and the information needed for each operation is what we call a *method expression* and its arguments. This is needed because the bbels to be accessed or modified are implemented as dynamic objects. The method expression is a composition of the different bbel's access methods.

For all the operations, except `DestroyBBel`, there must be a method expression associated to each bbel. This method expression, together with its arguments, will be sent to the appropriate bbel for evaluation.

In the ABB system, a method expression (*methodExp*) is a string of comma separated method calls.

methodExp ::= "*methodList*" | " "
methodList ::= *methodCall*, *methodList* | *methodCall*

A method call is composed by the name of the method and its parameters within parenthesis. The type of the method parameters are specified with a syntax similar to that of the C language procedure `printf`. A `%c` indicates that the parameter is a character, a `%d` indicates that it is an integer, a `%f` denotes a floating point number, `%h` indicates a short integer, `%ld` specifies a long integer, `%lf` denotes a double precision floating point number, `%s` indicates an string, and `%j` denotes an arbitrary structure. Only this last description varies from the `printf` specification.

methodCall ::= *methodNameArgs*
Args ::= (*ArgList*) | ( )
ArgList ::= *ArgSpec*, *ArgList* | *ArgSpec*
ArgSpec ::= `%c` | `%d` | `%f` | `%h` | `%ld` | `%lf` | `%s` | `%j`

All the argument specifications indicate that the method expression string must be followed by a number of variables in the same order and type as presented. The only exception is the arbitrary structure specification `%j`. In order to be general, this

specification needs two arguments. The first is the size of the structure (usually obtained via the standard **sizeof** macro), and the second is the pointer to the structure itself.

Each method name must be a valid method for the associated bbel. If there is no such method, the operation is aborted and an error returned. All methods will be applied sequentially in a left to right order according to their appearence in the method expression. However the expressions for different bbels will be evaluated in parallel. All the methods evaluated must return an exit status. In the case where there is one bbel method that returns an error, the whole operation will be aborted.

The **CreateBBels** function receives, as a first parameter the number of BBels to create, and following this, the identification of each bbel together with its method expression, and the method parameters. The system insures that the creation of the bbels is performed atomically. That is, the creation is not effective until all of them have been created. If there is one bbel creation that fails (one method returns an error), it will prevent the other bbels to be created. The ANSI C prototype for the **CreateBBels** function is as follows. Note that "**...**" is a valid ANSI C macro, defined within the "**stdargs.h**" header file, and it denotes a variable number of arguments.

```
CreateBBels(int nBBels,
            PATH pathBBel1, char * nameBBel1,
            char * methodExp1, /* methodExp1 arguments */
            ... )
```

An example of use would be:

```
CreateBBels(3,
            pbbel,"Thr 1","PutName(%s),setCounter(%d)", "Thr 1",1,
            pbbel,"Thr 2","PutName(%s),setCounter(%d)", "Thr 2",1,
            pbbel,"Control","PutName(%s),setCounter(%d)","Control",2);
```

The parameters for the **ReadBBels** are expressed in a similar way, except that the method expression arguments must be given by reference (pointer to the respective variables). This function will return an error if at least one of the bbel methods fail.

```
ReadBBels(int nBBels,
          PATH pathBBel1, char * nameBBel1,
          char * methodExp1, /* methodExp1 arguments */
          ... )
```

The **WriteBBels** calling conventions are exactly as those for the **CreateBBels** function. If there is one bbel method that fails, the whole write operation is aborted. Also, the system insures that no other system component modifies any of the bbels being written on, until the whole write operation is finished. This is to insure the low level consistency of the blackboard.

```
WriteBBels(int nBBels,
           PATH pathBBel1, char * nameBBel1,
           char * methodExp1, /* methodExp1 arguments */
           ... )
```

Finally, the **DestroyBBels** function does not need any method expressions. It receives the identification of the bbels to be destroyed (deleted). This operation only fails if there is one guard of one of the bbels being eliminated that returns an error.

```
DestroyBBels(int nBBels,
             PATH pathBBel1, char * nameBBel1,
             ... )
```

5.2.3 The Knowledge Sources

The ABB specification defines a knowledge source as having a *Name*, a trigger condition (*Trigger*), a control function (*KSCf*), a *Body*, and an agenda:

KnowledgeSource

$Name : PATH$
$Trigger : TriggerCond$
$KSCf : FilterCFn$
$Body : KSBodyProc$
$KSAgenda : \mathbb{F}\ KSAR$

In the prototype implementation a knowledge source will be an object with a particular internal structure, only accessible to the system via its own access methods. The *Name* will be the actual name of the knowledge source object. The *FilterCFn* and the *KSBodyProc* will be specially marked methods. These methods will be called by the system when it needs to obtain KSAR's from the *KSAgenda*, or to evaluate the *Body*. The following example shows the definition of the THR initialisation knowledge source.

File: InitKS-KS.ah

```
KSdefine InitKS {
/* Internal Structure */
     long runTime;
/* Methods */
     void INIT::initFn(InitKS::KS * self);
     void KSCF::AllKSARs(InitKS::KS * self, LIST ** agenda, LIST ** toSched);
     void KSBODY::InitKS(InitKS::KS * self, InitKS::KSAR * ksar);
};
```

The definition for a two-threaded system would be:

File: InitKS-KS.ac

```
void InitKS::initFn(InitKS::KS * self) {
     self->runTime = 0;
}
void InitKS::AllKSARs(InitKS::KS * self, LIST ** agenda, LIST ** toSched) {
     *toSched = *agenda;
     *agenda = NULLlist;
}
void initKS::InitKS(InitKS::KS * self, InitKS::KSAR * ksar) {
     PATH pbbel = idPATH("BB.THR");
     char cbuf[150];
     if (pathPATHNAME(&(ksar->bbel)) == NULLpath) {
          self->runTime=ABB_TIME;
          CreateBBels(3, /* Initialise Threads */
               pbbel,"Thr 1","PutName(%s),setCounter(%d)", "Thr 1",1,
               pbbel,"Thr 2","PutName(%s),setCounter(%d)", "Thr 2",1,
               pbbel,"Control","PutName(%s),setCounter(%d)","Control",3);
     } else {
          StopSystem(); /* End of Program */
          sprintf(cbuf,"Total Run Time:%d", ABB_TIME - self->runTime);
          abb_putStr("InitKS",cbuf);
     }
}
```

We can specify any sort of internal structure and any number of internal methods. There will be three special methods. The method marked INIT will be called by the system when creating a knowledge source instance. The method marked with KSCF will be the knowledge source control function. This will be used by the system to process the knowledge source's local agenda. It receives a pointer to the agenda, and it returns the list of KSARs to be scheduled (to be sent to the appropriate desk). This function can modify the agenda. Finally, the third special method is the one marked with KSBODY. This is the actual body of the knowledge source. It will receive the

scheduled KSAR, and will process it. This method, as well as the others, can use the blackboard manipulation procedures to read and write information into the blackboard. The internal method definitions are defined in the `InitKS-KS.ac` file.

The ABB system considers identification of problems, represented by the trigger condition, to be a task fundamentally different from the actual solving of the problem. Their information needs, and the actual knowledge they hold, is very different.

In the ABB prototype, the *TriggerCond* will be defined as a separate object. Its structure, according to the specification is the same as that of all its activation conditions (*ActivationCond*), and this in turn, is the same structure that all the *KSARs* it generates will have. In the system, the trigger condition object will define the structure of the knowledge source activation records. Besides the trigger structure, the *ActivationCond* will have a test that will generate a *KSAR* when it succeeds. The activation condition is attached to a specific blackboard level. We implemented these tests as special methods of the trigger condition objects. They will be marked with the level to which they are attached, and will share the same structure as the trigger condition.

File: InitKS-KS.ah

```
TRIGGERdefine InitKS {
/* Internal Structure */
/* Methods (triggers) */
     BOOL SYSINIT::Initialise(InitKS::KSAR * self);
     BOOL BB.THR::AllThreadsEnded(InitKS::KSAR * self,
               BB.THR::BBEL * trigBBel);
};
```

In this particular case, the KSAR has no internal structure. It only has two internal methods that will be evaluated at system initialisation time (`SYSINIT`), and when all threads end. When we mark a method with `SYSINIT`, the system, in the initialisation phase, will create a KSAR for the knowledge source, and will call this marked method to initialise it before sending it to the knowledge source's desk. The `BB.THR` mark identifies an *ActivationCond*. The marked method will be evaluated whenever a bbel within the indicated level is modified. The definition of these methods is as follows:

File: InitKS-KS.ac

```
BOOL InitKS::Initialise(InitKS::KSAR * self) {
     returnTRUE;
}
BOOL InitKS::AllThreadsEnded(InitKS::KSAR * self, BB.THR::BBEL * trigBBel) {
     char bbelName[NAMELENGTH];
     int  thrNum;
     trigBBel->GetName(trigBBel,bbelName);
     if (strcmp(bbelName,"Control") return(FALSE);
     trigBBel->GetCounter(trigBBel,&thrNum);
     if (thrNum == 0) return(TRUE);
     return(FALSE);
}
```

This fully defines the InitKS knowledge source. In the ABB prototype we must now define the actual knowledge source object. The ABB prototype allows the definition of more that one instances of a knowledge source. In the case of the InitKS knowledge source, we only need one such instance:

File: InitKS-KS.ah

```
defineKS InitKS InitKS_1;
```

However, for the thread knowledge source, we will need one instance per thread. The definition of the thread knowledge source is as follows:

File: ThreadKS-KS.ah

```
KSdefine ThreadKS {
/* Internal Structure */
     long threadLength;
/* Methods */
     void INIT::initFn(ThreadKS::KS * self);
     void KSCF::AllKSARs(ThreadKS::KS * self, LIST ** agenda, LIST ** toSched);
     void KSBODY::ThreadKS(ThreadKS::KS * self, ThreadKS::KSAR * ksar);
}
TRIGGERdefine ThreadKS{
     int  Number;
     BOOL BB.THR::matchThread(BB.THR::BBEL * trigBBel, ThreadKS::KSAR * self);
}
```

Each ThreadKS will trigger from a separate bbel. The name of the bbel must have the same number as the knowledge source. This is reflected in the matchThread condition.

File ThreadKS-KS.ac

```
BOOL ThreadKS::matchThread(BB.THR::BBEL * trigBBel, ThreadKS::KSAR * self) {
     char tmp[NAMELENGTH];
     int  i;
     sscanf(mySelf->ksName,"ThreadKS_%d",&i);
     sprintf(tmp,"Thr %d",i);
     if (strcmp(trigBBel->name,tmp) == 0) {
          mySelf->Number = trigBBel->Counter;
          return(TRUE);
     } else return(FALSE);
}
```

The thread length is initialised, by default, to thirty. The thread knowledge source control function is to process all KSAR's. We always send the whole agenda to be scheduled. Finally, the **ThreadKS** body decrements the knowledge source **threadLength**. If that number is greater than zero, it just stores it in its thread bbel. This modification will cause the **ThreadKS** to trigger again, forming a new cycle. If the **threadLength** becomes zero, it modifies the **Control** bbel, decrementing its counter.

File ThreadKS-KS.ac

```
void ThreadKS::initFn(ThreadKS::KS * self) {
     self->threadLength = 30;
}
```

File ThreadKS-KS.ac

```
void ThreadKS::AllKSARs(ThreadKS::KS * self, LIST ** agenda, LIST ** toSched) {
     *toSched = *agenda;
     *agenda = NULLlist;
}
void ThreadKS::ThreadKS(ThreadKS::KS * mySelf, ThreadKS::KSAR * ksar) {
     PATH pbbel = idPATH("BB.THR");
     int tn = ksar->Number;
     mySelf->threadLength--; tn++;
     if (mySelf->threadLength > 0)
          st = WriteBBels(1, pbbel,ksar->bbel.name, "setCounter(%d)", tn);
     else {
          abb_putStr(mySelf->ksName,"Finishing...");
          st = WriteBBels(1, pbbel,"Control", "decCounter()");
          abb_putStr(mySelf->ksName,"Finished!");
     }
     if (st != OK) {
          sprintf(tmp,"Status Not OK:%d", st);
          abb_putStr(mySelf->ksName,tmp);
     }
}
```

This completes the definition of the thread knowledge sources and their associated triggering conditions. In the implementation, we must now define the actual knowledge source object. There can be more than one of such objects, representing several instances of the same knowledge source. For the **ThreadKS**, we can define several instances of it as follows.

File: ThreadKS-KS.ah

```
defineKS ThreadKS ThreadKS_1;
defineKS ThreadKS ThreadKS_2;
```

Note that as the knowledge sources are implemented as objects, the information held in their internal structure may be different. Furthermore, this information will not be destroyed until the object is destroyed. In the ABB implementation, all knowledge source objects are created in the initialisation phase of the system, and they are deleted only when it finishes. This means that any information placed in the knowledge source internal structure within one body execution will still be there in the next one. This allows the knowledge sources to remember information from previous body executions, and use it in the solution of future problems. This ability is not common in blackboard systems. The only way to achieve similar functionality was to make the information public by storing it in the blackboard.

5.2.4 The Desks

The third type of ABB component is the desk. In the current ABB implementation, each desk is seen as a representation of the global system. As such, all the ABB desks are objects of the same class, the *system* class. We will not be defining each desk separately, we will define them through the system class. The ABB global system definition is specified as:

File: THR-Sys.ah

```
SYSdefine THR {
/* Internal Structure */
     int sys_var;
/* Methods */
     void INIT::SysInit(SYS * self);
     void SYSCF::FIFO(SYS * self, LIST ** agenda, KSAR ** toRun);
};
```

The system class structure and methods can be freely defined by the user. There will be two special methods. One is marked with **INIT**, and will be used by the system

to initialise each desk in the starting phase of the system. The other special method is marked with SYSCF, and will be used by the system to extract one KSAR from the desk agenda. This KSAR will then be sent to the appropriate knowledge source for execution. The system control function may also change the current local desk agenda. The definition of the THR desk methods is as follows:

File: THR-Sys.ac

```
void SYS::SysInit(SYS * self) {
     self->sys_var = 1;
}
void SYS::FIFO(SYS * self, LIST ** agenda, KSAR ** toRun) {
     if (*Agenda != NULLlist) {
          *toRun = (KSAR *) headLIST(*Agenda);
          *Agenda = tailLIST(*Agenda);
     }
}
```

The desks are objects of the global system class. Each desk will have a number of knowledge sources assigned to them. The definition of the number of desks and their associated knowledge sources is done using the defineSYS construct.

File: THR-Sys.ah

```
defineSYS THR Dsk_1 {InitKS_1,ThreadKS_1};
defineSYS THR Dsk_2 {ThreadKS_2};
```

The desk's main task is to manage the evaluation of their knowledge sources. As with the knowledge sources, their structure will be active during all the execution of the system. Each desk, then, has the ability to remember its previous decisions, and may use this knowledge in the future. The type of the knowledge held on each desk is the same, but the actual knowledge will be different.

This definition implements the desk specification defined in the previous chapter:

Desk

$Name : PATH$
$KSnames : \mathbb{F}\ PATH$
$DeskAgenda : \mathbb{F}\ KSAR$
$DeskCf : SelectCFn$
$ActualKS : PATH$
$Eval : Procedure$

The *Name* of each desk, as well as the set of *KSnames*, is assigned via the `defineSYS` construct. The *DeskAgenda* is internal to each desk, and it is accessed by the system through the desk's control function (*DeskCf*). This function is implemented by the method marked `SYSCF`. The *ActualKS* and *Eval* are also internal to each desk.

5.2.5 General System Structure

We have showed how the different ABB components are described within the ABB prototype. The union of all these components implement the ABB system specification given in the previous chapter:

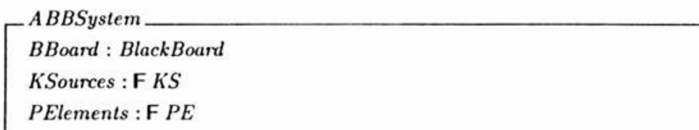
ABBSystem
BBoard : *BlackBoard*
KSources : $\mathbb{F}$ *KS*
PElements : $\mathbb{F}$ *PE*

Figure 5.1 shows a graphic representation of the ABB system components and their data interaction. We will now explain how the different control information is shared between the blackboard components in order to make the system as a whole work.

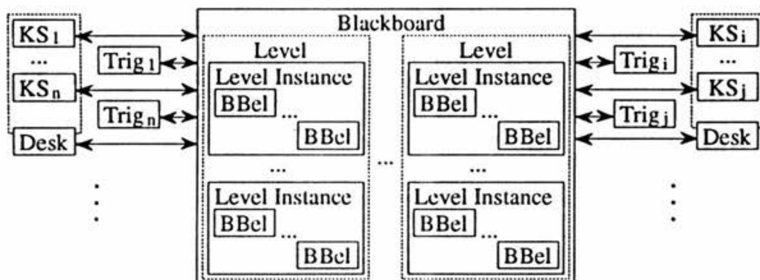

Figure 5.1. The ABB System Components.

The dotted line groups all the knowledge sources that work within a specific desk, together with the graphic representation of the desk.

In the ABB system, there is at least one knowledge source which has the start of the system as one of its trigger conditions (initialisation knowledge sources). The ABB system global operation begins by generating an activation record (KSAR) for all

initialisation knowledge sources. These KSAR's are then sent to the knowledge source's respective desks. The ABB basic control cycle continues as follows.

1. Each idle desk will choose a KSAR for execution (if it has any), and will notify the indicated knowledge source.

2. Each knowledge source, concurrently, will evaluate its body. This will use the information stored in the chosen KSAR, and will access and modify information in the blackboard.

3. When the blackboard is modified, it evaluates the knowledge source trigger conditions specific to the level in which the modification was made. This evaluation will produce a number of KSAR's. These KSARs will be filtered by the blackboard and will be sent to the appropriate knowledge sources.

4. When one knowledge source finishes its work, it will notify its desk. The desk will, then, ask the knowledge source for any KSAR's that may have arrived while it was working. The knowledge source will order the KSAR's and send the desk a number of them. It may choose to keep some of them.

5. The cycle, then, repeats until there is one knowledge source that decides that a solution has been reached, and stops the system.

Figure 5.2 shows a graphic representation of the ABB control cycle. The numbers in the figure refer to the points above.

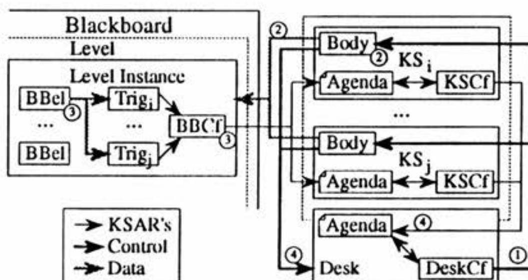

Figure 5.2. The ABB System Control Cycle.

The ABB system prototype is implemented by a number of library functions that provide all the above system functionality, plus a number of support functions

such as the blackboard manipulation functions, and some lesser ones such as PATH handling, and generic LIST handling. We will now describe the more important support functions, the blackboard manipulation functions.

5.3 Prototype System Implementation

The ABB parallel prototype is implemented by a series of processes that manage the different types of bbels, knowledge sources, and desks. The current system prototype is designed to work on a distributed memory multiprocessor machine such as the Meiko Computing Surface, or any other similar machine. It assumes that inter-process communication is fast, and that every process can communicate with all the others.

All process communication takes place via a number of data channels. Each ABB process will have at most four of those channels. There will be one channel per type of message it can receive. There will be one dedicated to command communication (`cmd`), other will be dedicated to the transmission of KSARs (`ksar`), and the other two for the communication of bbel addresses (`indx`) and bbel data (`bb`).

5.3.1 General Structure

All ABB processes are command based. They wait for commands to appear in their `cmd` channel. When they receive one, they process it, send the result, and wait for the next command. This behaviour is also called event-driven. Complex interactions can take place, and are called transactions, and are composed of several commands. All transactions are started from some process. This transaction will be processed up to the point where it needs data from other process, or it needs to synchronise. All processes will keep all the information needed to keep track of the status of each transaction. In one ABB process, at any point in time, there can be several active transactions in different points in their processing.

5.3.2 System Modules

The ABB parallel prototype implementation is built around four basic types of processes. These are the blackboard manager, the bbel index manager, the knowledge source manager, and desk manager.

BBel Index Manager and Blackboard Managers

In the current ABB prototype, bbels of the same type are stored together in several level instances. This is implemented by several ABB processes. For each level, there will be one *Index Manager* process, and as many *Blackboard Managers* as level instances are defined by the user.

The index managers will keep a list of the bbels created in their specific level with their precise location. This location has the process identification of the blackboard manager that holds the bbel, plus an internal reference supplied by the blackboard manager. The index manager task is to provide the address of each bbel within a given level as fast as possible. When any ABB system component makes use of one of the global blackboard manipulation functions, it will first contact the index manager for each of the bbels in the operation. It will ask for their location, once at the beginning of the operation. It will then send the operation, and perform any other interaction communicating directly with the appropriate blackboard managers.

A blackboard manager is the implementation of a particular level instance. It will hold a number of bbels of the same type, and its main function is to receive and serve requests from all other processes for creating, reading, modifying and deleting blackboard elements of a given type.

Another very important task of the blackboard managers is to perform the triggering of knowledge sources. Each blackboard manager will hold a copy of the trigger conditions of all those knowledge sources that trigger on modifications to the level they represent. Note that only those methods within the trigger specification that have been marked with the appropriate level will be executed. This will be done every time a bbel is modified. This evaluation will produce a list of KSARs that will be then processed using the user-defined blackboard control function for this level, and the resulting KSARs will be sent to their respective knowledge sources.

Knowledge Source Managers

In the ABB prototype all knowledge sources are implemented by separate processes called *knowledge source managers*. There will be one of such processes per knowledge source instance defined by the user. Their task is to process all knowledge source related events. They must be prepared to receive KSARs from blackboard managers at any moment, as well as evaluate the knowledge source body on demand. All knowledge source managers will be linked to a specific desk. The knowledge source manager will receive requests of KSARs from its desk. It will use the user defined control function

to decide which KSARs to send, and with which priority. The associated desk will also send orders for the processing of given KSARs.

Desk Managers

Finally, the desks of the ABB model are implemented by separate processes. Their number is specified by the user when he defines the number of desk instances. The user also specifies which knowledge sources will be attached to which desk. The desk task is to manage the execution of its assigned knowledge sources. It will ask them for KSAR's, usually when they finish execution of their body. The desk will then keep an agenda with KSARs from its knowledge sources, and will schedule them using the user defined desk control function.

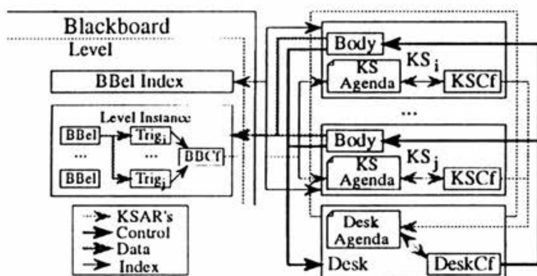

Figure 5.3. The ABB Prototype Processes.

Figure 5.3 shows a representation of the different processes that implement the ABB system prototype. The figure highlights the blackboard data paths. The control flow of the system is identical to that showed in Figure 5.2.

5.3.3 Implementation Issues

During the implementation of the ABB parallel prototype, we had to confront a number of problems introduced by the parallel environment in which it was developed.

Early on in the implementation, we found that it was extremely important to have a simple, yet efficient, inter-process communication mechanism. When designing ABB, the only communication library available to us was MEIKO's CSTools. Initially, we decided on using a subset of the CSTools library. However, we needed it to be portable enough for the system to be able to run on other systems.

Communication Mechanism

We decide to implement a extremely small and simple communication library called *SAMP* (Simple Asynchronous Message Passing). This library will be described at a greater length in Appendix F. The SAMP library is built up by eight procedures that could be grouped in four categories.

The first one is formed by global system operations for initialising (`samp_init`) and exiting gracefully from the system (`samp_exit`). The `samp_init` routine receives, as its arguments, pointers to the usual C main procedure arguments.

```
STATUS samp_init(int * argc, char *** argv);
STATUS samp_exit(int code);
```

Any given process, using the SAMP library, can access any number of communication channels. The two following routines, `samp_open` and `samp_close`, allow for the opening and closing of those channels. The complete argument description is provided in Appendix F.

```
CHANNEL samp_open(char * chan_name, char * mode);
STATUS samp_close(CHANNEL chan);
```

The next two procedures implement the usual read and write communication primitives.

```
size_t samp_read(CHANNEL mychan, char * data);
STATUS samp_write(CHANNEL tochan, size_t datasz, char * data);
```

Finally, the last two routines provide the SAMP library the capability of performing asynchronous communication. `samp_qbuf` allows the placing of a new buffer in the input queue. This allows the receiving of messages before the reader process is ready to use them. This procedure can be used whenever a process knows that a message will be received in the future. The other function is `samp_test`. It performs a non-blocking test of the input message queue.

```
STATUS samp_qbuf(CHANNEL mychan, size_t datasz, char * data);
STATUS samp_test(CHANNEL mychan, char ** data);
```

Blackboard Concurrency Control

Once we had a working communication library, the second biggest problem we faced was the way in which several concurrent blackboard accesses could be allowed. In the previous chapter, we defined the general system operations as follows.

$$ABBSystemOp = ExecBBOp \cup ExecPEOp \cup PEprepareOp \cup MoveKSAROp$$

This means that at a given moment, the system can perform a blackboard operation (*ExecBBOp*) using the blackboard manipulation functions, a knowledge source can continue with its body execution (*ExecPEOp*), a desk can choose a new KSAR to be evaluated (*PEprepareOp*), or KSARs can be moved between the different system agendas (*MoveKSAROp*). All these operations are performed by the ABB prototype.

In order to comply with the specification, the ABB prototype must maintain the global system consistency. That is, the blackboard must be *Coherent* at all moments. In the specification, it was defined that a given system state is coherent if there exists at least one sequential ordering of its operations by which we can reach the state from the system's initial state.

Note that all system operations, except *ExecBBOp* are performed by one operation in the ABB prototype. *ExecPEOp* just evaluates one operation of the body of a knowledge source, *PEprepareOp* is internal to the desk and it is implemented by the evaluation of the desk's control function. The *MoveKSAROp* is performed by each knowledge source choosing the KSARs to be sent to their desks, and sending them one by one. The communication mechanism insures the seriability of these operations.

The *ExecBBOp* are the only operations in which several system components must interact several times within one operation. Depending how these operations are implemented, the coherence condition can be broken. In order to comply with the specification, the ABB blackboard must employ some concurrency control mechanism that insures the seriability of blackboard access operations.

This problem is not particular to parallel blackboard systems, it is also shared by distributed databases. Several schemes have appeared in that research area for insuring the seriability of database (blackboard) access and modification. The most important are the transaction based [KR88], and the lock based schemes [GBPT88].

In the current prototype, we decided to implemented a lock-based schema based on its ease of implementation, and the possibility to tune it to the prototype specific needs. The current implementation of this mechanism tries to maximise the availability of the bbels for read operations. Problems arise when there are several processes trying to modify (write or delete) the same bbel.

In the implementation, all modifications to bbels are performed on a copy of it, thus allowing read access at any moment. It is only when all blackboard managers involved in a given write (or delete) operation synchronise for making it effective, that the read access is disabled. This is only for the time it takes to swap the old bbel with

the modified one.

In the ABB prototype it is very important that read access to bbels should be maintained as much as possible. This is due to the use of guards on bbels. Any given bbel may be accessed directly by a knowledge source, or indirectly by a guard of some other bbel. If it is locked for reading, it may not only stop any read operation on the bbel, but it may also block other operations somewhere else in the blackboard.

5.4 Summary and Conclusions

In this chapter we have presented the ABB system parallel prototype. With this system, we provide the desired implementation characteristics presented in the introduction to Chapter 4.

We consider the implementation itself to be the major contribution of this chapter. This implementation is completely original. Its object oriented design, its distributed control architecture, and parallel implementation, makes it a unique blackboard system in terms of functionality and versatility.

The control mechanism in the prototype is distributed among all the main system processes. All control decisions are made locally and there is no central scheduler. This opens the problem of how to translate coordinated global strategies to local control decisions. Work needs to be done to provide mechanisms and techniques similar to those in BB1 and other planning systems adapted to distributed blackboard process control.

With the development and use of the SAMP communication library we provide a simple and efficient way for process communication. This, however, is totally hidden from the ABB system user. The model in which the system implementor works is a purely blackboard model where the only hint of communication is provided by the blackboard manipulation functions.

The prototype also provides an efficient triggering mechanism. The activation conditions are evaluated only on those levels in which they apply. If there are several level instances, and there is a bbel modification on each one, the trigger evaluation will be performed in parallel.

In the next chapter, we will present a blackboard systems test suite that we developed in order to test the performance of the prototype. We will also show the performance measurements obtained by testing the system under different platforms and configurations.

Chapter 6

The ABB System Performance

6.1 Introduction

In this chapter, we will analyse the performance of the ABB parallel prototype. We developed two different blackboard systems in order to test the prototype capabilities and performance. These systems are the thread simulation system (THR), and the jigsaw-puzzle solver (JSP). In this chapter, we will show the performance measurements for a ten thread and a twenty thread system, and three configurations of a 10 by 10 jigsaw puzzle solver.

The main aim of this chapter is to show that a significant increase in performance can be obtained by making use of parallel computations. Our intention is not to provide an exhaustive test of all the possible system configurations, running on any number of processors. We will however present the performance results for the test blackboard systems showing a significant increase in performance.

The results presented in this chapter show that an order of magnitude improvement in performance can be achieved by using current parallel, distributed memory computers. This, of course, will depend on the application and its implementation. We will also show that the ABB prototype architecture can be easily adapted to the application's requirements in order to improve its performance.

We should bear in mind that the ABB parallel prototype is a research tool for demonstrating the performance improvements that can be achieved using the active blackboard model. We tried to solve each implementation problem in an efficient manner. However, it is possible to optimise the present prototype in order to obtain even better results.

6.1.1 Theoretical Speed-up

In the analysis presented in this chapter we will be comparing the ABB implementation speed-up with the best possible speed-up given a number of processors. We will use the speed-up measurement introduced within the performance framework presented in Chapter 2 alongside with more traditional ones.

In general, all speed-up measurements are based in comparing the run time of a base system with that resulting from some variation or modification of the same system. In the case of parallel computations, the base system is usually the result of running the system on one processor, and the variations are the product of running the same system using different number of processors. In the framework we use an estimate of the base system that is stronger than just using the runtime of the system while running on one processor.

In parallel systems, besides the base system, we need some sort of reference to which compare the speed-up of the different systems. These references are upper-bounds of the best possible speed-up.

In [Gel89], a number of upper bounds for the possible speed-up are presented. The most commonly used bounds are the *Linear* speed-up and *Amdahl's Law* [Amd67, Gus88]. Suppose that a program is executed in time E in a single processor. If N processors are provided, then the execution time will now be E/N. This is assuming that the program can be split on to N parallel components that execute in the same amount of time. The speed-up for such a program will be N. This is the theoretical linear speed-up.

Amdahl's Law states that, in general, there will be some amount of time C spent in communication between the different parts of the program. Amdahl's law states that the total effective execution of the parallel program is $(C + E/N)$. The maximum speed up is then the ratio of execution time with respect to that of a sequential processor.

$$\frac{E}{\frac{E}{N}+C} = \frac{N}{1+\frac{C}{E}N}$$

However, this upper bound has been shown [Gel89] to be an unnecessarily pessimistic estimate, and values close to N may be obtained for particular applications. We will present the ABB prototype speed-up compared with the more optimistic, linear estimate.

6.2 Test Suite

We will use two applications for demonstrating the possibilities offered by the new blackboard model. The two applications are designed to test different aspects of the blackboard architecture. The first blackboard system is the thread application, an extremely simple independent task system that makes heavy use of the blackboard system cycle. The second test blackboard system is an implementation of a jigsaw-puzzle solver. The emphasis of this second application is in blackboard access. Both of these applications can be easily implemented, and have the advantage that they can be easily scaled. The task system can be made to work with different number of independent tasks, and the jigsaw puzzle solver can be configured for different sizes of jigsaws, thus providing the system with a more challenging, and lengthy problem.

6.2.1 Thread (THR) Test Application

The idea behind the thread blackboard system is a very simple and controlled way of emulating application task threads. When analysing the speed-up that can be achieved for any application by making use of parallel computation, we find that it is restricted by the number of its tasks that can be successfully performed concurrently. When we analyse the application, we find that it is composed of a number of sequential tasks. These tasks will present a precedence ordering based on their data dependencies. The size of these tasks defines the granularity of the application. The execution of the application can be represented by an acyclic graph with one starting node and a termination node. Figure 6.1 illustrates one possible graph. The arcs of the graph are the task's precedence relation. We call each path within the graph, a thread. Intuitively, the amount of parallelism that can be achieved by splitting any particular application will be restricted by the number of simultaneous threads of the application graph.

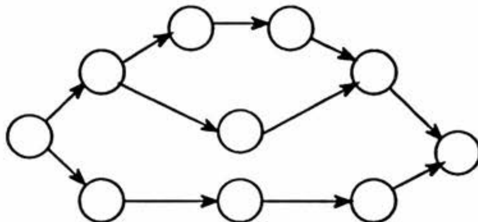

Figure 6.1. Example of an Application's Task Precedence Graph

In order to test how much the ABB prototype affects the parallelism inherent in a particular application, we developed the thread blackboard system. This system consists of one initialisation and termination knowledge source, and a number of thread knowledge sources. The task of each thread knowledge source is to decrement a counter held in a particular blackboard element, until it reaches zero. Each knowledge source will modify a different bbel. For the effect of initial synchronisation, all thread knowledge sources will trigger from the same synchronisation bbel. After that initial triggering, all thread knowledge sources will activate from the bbel they are modifying.

When the bbel of a given thread knowledge source reaches zero, it will modify a specific bbel to notify that it has finished its work. When all threads finish, the initial knowledge source will trigger again, and will print out the elapsed time between the initialisation, and the termination of all threads. Figure 6.2 shows a representation of

the thread system task graph. Each node in the graph will be a different task. The ones presented in the graph are, blackboard modification (BB), and knowledge source body execution (Init KS, THR i, and End KS). For clarity, we did not include the trigger and scheduler evaluation. The blackboard modification is always followed by trigger evaluation, and the knowledge source body execution is always preceded by a scheduler evaluation.

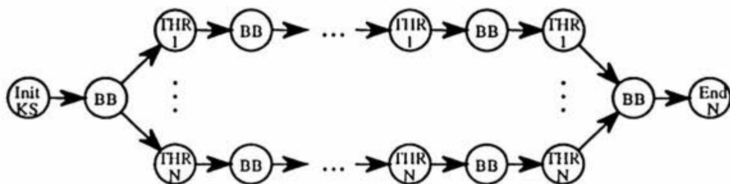

Figure 6.2. Thread Application's Task Precedence Graph

The THR blackboard system will have one blackboard level that will hold the different thread bbels, plus one used to keep track of how many threads are still active. It will also have one initialisation and termination knowledge source, and a number of thread knowledge sources. The THR application can be configured for any number of threads. For the purpose of evaluation, we will use two configurations. These will be the THR-10 system with ten threads, and the THR-20 system with twenty threads.

We will now present the thread (THR) application performance table, as specified in Chapter 2. Table 6.1 presents the module usage of the THR-10 application, and its performance measurements when running on one processor. Each thread will be a formed by thirty blackboard cycles. Note that the THR application makes equal use of all the modules, except for the scheduler module. This is because the three schedulers (blackboard, knowledge source and desk) are accounted for separately.

The ABB prototype configuration used for this evaluation uses five bbel managers for the application level, eleven knowledge sources (one initialisation and then threads), and eleven desks (one per knowledge source).

THR-10 Application: Module Use

Module	Number of calls	Time Use
BB. Manipulation Module	315	24.30%
KS. Activation Module	321	17.36%
Scheduler Module	926	32.98%
KS. Execution Module	312	25.34%

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times	
				Average	Total
BB. Manipulation				AM 0.0022	TM 0.70
Module	BBel Create	AC_M 0.0215	PC_M 15.39%		
	BBel Access	AR_M 0.0000	PR_M 0.00%		
	BBel Modify	AM_M 0.0019	PM_M 84.61%		
	BBel Destroy	AD_M 0.0000	PD_M 0.00%		
KS. Activation				AT 0.0016	TT 0.50
Module	KS. Trigger	AT_T 0.0016	PT_T 100.00%		
Scheduler				AS 0.0010	TS 0.95
Module	Choose KS.	AS_S 0.0010	PS_S 100.00%		
KS. Execution				AB 0.0023	TB 0.73
Module	Evaluate KS.	AB_B 0.0023	PB_B 100.00%		

Module Influence

Total System Run Time	T_{Run}	2.88
Total Unaccounted Time	TUa	0.15
Total System Time	T_{Sys}	3.03
Unaccounted Time Prop.	P_{Ua}	5.05%
Module Average Influence		
BB. Manipulation Module	I_M	31.14%
KS. Activation Module	I_T	21.85%
Scheduler Module	I_S	14.39%
KS. Execution Module	I_B	32.62%
Service Modules	I_{Sp}	0.00%

System Performance

Average Cycle Time	A_{Cy}	0.01
System Speedup	SU	1.00

Table 6.1: THR-10 General Performance Table – ABB One Processor.

From the module use table, we can see that the THR system uses all modules approximately equally. The scheduler module appears to be greater, as its three phases (blackboard, knowledge source, and desk), are accounted separately. The pie chart to the right of the bar chart shows the proportion of the total time spent on each system module.

In the THR system all all the blackboard functions are extremely simple. At each system cycle there is one blackboard modification, ten knowledge source trigger evaluations, and the scheduling of one KSAR.

We would expect the system to spend most of its time in the blackboard manipulation module, and the knowledge source activation module. However, in the first pie chart we can see that the system adds some overhead during knowledge source body execution and scheduling. The time spent in the evaluation of the body of the knowledge source includes the activation record communication from the scheduler to

the knowledge source, and the notification of termination from the knowledge source to the desk. This handshaking accounts for most of the time spent in that module.

By comparing the overhead produced in the knowledge source evaluation module to the blackboard manipulation time, we can conclude that this overhead is roughly equivalent to the time needed to perform one blackboard modification. This can also be seen in the second pie chart, showing the proportion between the module's average times.

The scheduler module also adds some overhead. This is mostly caused by the communication of activation records between the different system modules. By comparing this to the blackboard manipulation module, we can see that each scheduler stage adds an overhead roughly equal to half the time needed to modify one bbel.

In order to improve the performance of the THR system, we need to decrease the time spent in all blackboard modules. In the ABB prototype this can be achieved by performing some of the system's tasks in parallel. In the next section, we will show the speed-up that can be obtained in this way.

6.2.2 Jigsaw Puzzle (JSP) Solver Application

The second test blackboard system used is the jigsaw puzzle solver (JSP) already used for the evaluation of other blackboard systems. For a more complete description of the JSP system, please look at Chapter 3, Section 3.2. This system makes heavy use of the blackboard.

The task of the JSP system is to place jigsaw pieces onto a board. The pieces, and the board are stored in the blackboard. The JSP blackboard will have three levels, one for holding the pieces, one for the board positions, and a third one specifying which pieces correspond to each knowledge source.

The JSP system will have a number of knowledge sources representing the players. There can be any number of players collaborating to solve the jigsaw. Initially, each player will be provided with a number of pieces. The task of each player is identical. They look at the board to find out if there is any one of their pieces that match any position.

As the pieces can be rotated, and the number of side shapes is limited, it is possible that several pieces legally match a given position. This means that the players can place pieces in the wrong places. We need an extra knowledge source that identifies when this happens. We will call it the *Piece Misplaced* knowledge source. Its job is, basically, to backtrack from decisions taken by the players. It will take out any possible

offending piece, and will return it to its player. It will also temporarily mark the position so that the same piece cannot be placed back to it.

Table 6.2 shows the module utilisation of the JSP blackboard system, and its performance measurements using the ABB prototype running on one processor. The JSP system run for this evaluation solves a 3 by 4 jigsaw puzzle using six knowledge sources, one initialisation and termination, the piece-misplaced knowledge source, and four player knowledge sources.

The ABB prototype configuration used for this evaluation uses one bbel manager per level, six knowledge sources, and one desk. This can be seen as the standard sequential configuration.

JSP (3x4) Application: Module Use

Module	Number of calls	Time Use
BB. Manipulation Module	876	96.52%
KS. Activation Module	157	1.31%
Scheduler Module	42	1.63%
KS. Execution Module	15	0.54%

Number of Calls Time Use

Module Performance

Module	Function	Average Time	Percentage of Use	Module Times Average	Module Times Total
BB. Manipulation Module				AM 0.0080	TM 10.42
	BBel Create	AC_M 0.0075	PC_M 3.08%		
	BBel Access	AR_M 0.0120	PR_M 90.53%		
	BBel Modify	AM_M 0.0126	PM_M 6.39%		
	BBel Destroy	AD_M 0.0000	PD_M 0.0000%		
KS. Activation Module				AT 0.0009	TT 0.14
	KS. Trigger	AT_T 0.0009	PT_T 100.00%		
Scheduler Module				AS 0.0042	TS 0.18
	Choose KS.	AS_S 0.0042	PS_S 100.00%		
KS. Execution Module				AB 0.0039	TB 0.06
	Evaluate KS.	AB_B 0.0039	PB_B 100.00%		

Module Influence

Total System Run Time	T_{Run}	10.80
Total Unaccounted Time	TUa	0.01
Total System Time	T_{Sys}	10.81
Unaccounted Time Prop.	P_{Ua}	0.07%
Module Average Influence		
BB. Manipulation Module	I_M	47.14%
KS. Activation Module	I_T	5.29%
Scheduler Module	I_S	24.67%
KS. Execution Module	I_B	22.91%
Service Modules	I_{Sp}	0.00%

System Performance

Average Cycle Time	A_{Cy}	0.72
System Speedup	SU	1.00

Table 6.2: Jigsaw Puzzle Solver General Performance Table
ABB One Processor

From the module use table, and bar chart, we can see that the JSP system makes heavy use of the blackboard manipulation module. As we would expect, the pie chart shows that it spends most of its time manipulating the blackboard.

The JSP system, in comparison with the THR-10 system, makes use of more complex features of the ABB prototype. Its knowledge source trigger conditions are nontrivial. In order to trigger, a piece knowledge source must check if one of its pieces matches a given position. This means that the trigger condition must perform blackboard operations. The time spent during those operations is accounted to the blackboard manipulation module time. This accounts for the relatively fast triggering time. The JSP scheduling of knowledge sources is also nontrivial, as it has to decide when to suppress some activations, and it has to prioritise between the rest. The operations used by the body of the knowledge sources are more complicated than the THR-10 system ones, as they modify several blackboard elements with one blackboard operation. This is needed in order to insure blackboard coherence. These operations take up more time than the simple modifications used by the THR-10 system. This accounts for the increase in the average blackboard manipulation time.

Although there are large differences in implementation language, processor type, and operating system, with the other blackboard system shells studied in Chapter 3, the ABB prototype compares favourably with them. The ABB system was implemented in C, and runs on one transputer model T800. The BB1 system was written in LISP, and runs on a MicroExplorer LISP machine. The EPBS system was implemented in the Edinburgh PROLOG running on a Sun SPARCstation IPX. The Blondie systems were developed in POP-11, within the POPLOG AI environment, also running on a Sun IPX. [Don89] provides a comparison of the performance of various computer systems using standard linear equations software. He used the standard LINPACK benchmark to test the speed of several computer systems. In his report, he gives the following performance measurements, in millions of floating point operations per second (Mflop/s).

Computer	LINPACK, n=100 (Mflop/s)
SUN SPARCstation IPX	4.10
Inmos T800 (20MHz)	0.37

[Don89] only evaluated general-purpose machines. The MicroExplorer, being a specialised LISP machine does not figure in his report. However, as the MicroExplorer is optimised for the execution of LISP, we can assume that it is at least as fast as the Inmos Transputer, and surely faster.

The total run-time of the ABB prototype for the resolution of a three by four jigsaw puzzle on one transputer (10.81) is smaller than the time taken by BB1 to solve a simmilar three by four jigsaw (123.12 seconds). It is also much smaller than the time taken by EPBS to solve the same problem (2377.36 seconds). We also implemented the three by four jigsaw puzzle solver in the Blondie-I and Blondie-III systems. The ABB prototype performs faster than Blondie-III (86.92 seconds), its speed is very simmilar to that of Blondie-I (10.66 seconds).

In order to improve the performance of the JSP system, we need to concentrate on improving the actual blackboard access and manipulation times. Within the ABB prototype, this can be achieved by making use of blackboard level parallelism. We can define several blackboard managers per level, thus allowing the evaluation of several blackboard functions in parallel. In order to test this, we implemented a second JSP system configuration with the same number of knowledge sources and desks, but with three blackboard managers per level. In the next section we will present the speed-up that can be achieved by this method.

6.3 Prototype Performance

Besides the usual framework performance measurements, we will also present the system's performance from the user's point of view. We will measure the elapsed time between the user's initialisation of the application, and the identification of the solution of the domain problem.

In order to facilitate this measurement, we implemented the initialisation, and stop knowledge sources as one. This knowledge source will trigger trivially for system initialisation, and by the system's termination condition for exiting the application gracefully. When it first triggers, it will store the current time on its internal structure. This time is measured before any initialisation is performed. When the termination condition is identified, it will perform any duty necessary, and will measure the time again. It will print the elapsed time between the system's initialisation and termination. Note that this is possible, as the ABB knowledge sources keep their internal state between evaluations. This implementation has the advantage that the time measured at both moments is totally coherent, as it corresponds to the same process and processor.

If this were to be implemented by two separate knowledge sources, the possible disparity of clock times and speeds may render the information unusable, unless some type of external clock server can be used.

6.3.1 Prototype Instrumentation

We instrumented the ABB prototype trying to add as little overhead as possible. The target platform for the ABB prototype is the MEIKO transputer-based multiprocessor. In this system all I/O is performed by sending messages between processors to a particular one, or to a host computer, which will then read or write to the console or to a file. In order to avoid saturating the communication bandwidth, we decided to store all profiling data in memory, and only write it down when the system finishes. We consider that the cost of writing the profiling data to disk outweights the amount of memory needed for this task. We did not encounter memory problems for the applications tested, although this may be a problem for more complex applications. For these applications, a buffering scheme could be easily implemented. All the data presented here includes any possible delays introduced by the profiling mechanism.

6.3.2 Thread Application

We implemented two versions of the thread blackboard system. The first version implements ten threads (THR-10), and the second version runs twenty independent threads (THR-20). Each application uses one desk per knowledge source in order to try to make the threads as independent as possible. The THR-10 system uses five bbel managers, and the THR-20 system uses ten of such managers.

The idea of using the thread system is to find out how much parallelism can be achieved within the ABB prototype. In theory, the parallel THR-10 system should be able to perform ten times as fast as the same system running on one processor. The THR-20 system should be twenty times faster. The closer the system can be to those values, means that the implementation of the shell prototype does not restrict the parallelism inherent in the applications implemented on it.

The THR-10 system used twenty-eight communicating processes. There are five bbel managers, and their associated bbel index. There are also one initialisation and termination knowledge source and ten thread knowledge sources. Each knowledge source has a separate desk.

Figure 6.3 shows the total system speed-up for the ten thread system, and its speed-up in function of the number of processors used. The speed-up values shown

are calculated using the formula introduced in the blackboard framework presented in Chapter 2.

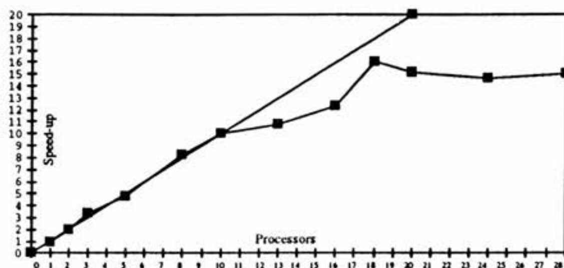

Figure 6.3. The THR(10) Framework Speed-up.

The previous figure presents an almost linear increase in performance up to the use of eighteen processors. For further processors the speed-up stabilises between fifteen and sixteen times faster. Note that, in theory, the best speed-up we could hope for in the THR-10 system is ten. However, experimentally, we obtained almost sixteen. Note that the THR-10 system uses twenty-eight processes. This extra improvement is obtained by the execution of the ABB system functions also in parallel. In order to corroborate the framework's speed-up measurement, we also measured the system speed-up from the user's point of view. This is the elapsed time between the beginning of the resolution of the problem, and its final solution. We use the run time for one processor as the reference time. The speed-up then shows how many times faster the same system running on a number of processors performs.

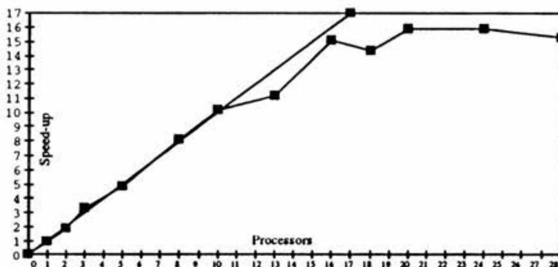

Figure 6.4. The THR(10) User Speed-up.

Figure 6.4 and figure 6.3 are slightly different, specially when the number of

processors is large, due to different measurement sensibility. However, their behaviour is very similar. Figure 6.4 shows an almost linear speedup up to sixteen processors, and then stabilises on a speed-up between fifteen and sixteen times faster.

Figure 6.5 shows the system speed-up for the twenty thread system in function of the number of processors used. The THR-20 application uses fifty-three processes, twenty-one knowledge sources, the same amount of desks, plus ten bbel managers and one bbel index.

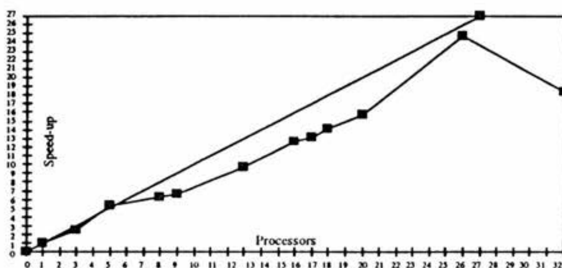

Figure 6.5. The THR(20) System Speed-up.

We notice that the speed-up increase for the THR-20 system is also linear up to the use of twenty-two processors. The system speed-up stabilises around twenty-one. In this system the effect of the system operation is smaller with respect to the number of threads of the application.

These two systems show that the ABB system can be configured in such a way as to allow full exploitation of the parallelism inherent in a blackboard application. In the following section, we will show the parallel behaviour of the jigsaw puzzle solver application.

6.3.3 Jigsaw Puzzle Solver

We implemented the jigsaw puzzle solver system presented earlier using three different ABB configurations. They all solve a ten by ten jigsaw using five players (piece knowledge sources). We will call this, the JSP-5 system. The variations in the configurations reside on the number of blackboard managers and desks used by the system.

The basic JSP-5 blackboard system uses three blackboard levels and seven knowledge sources (five players, one initialisation and one piece-misplaced knowledge source). Initially, we configured the ABB system to have three blackboard managers, three index

managers, seven knowledge sources and one desk. These are the fourteen processes that conform this implementation. We will call this configuration *JSP-5(14)*. We consider this configuration to be the classic sequential blackboard architecture.

Figure 6.6 shows the speed-up from the user point of view of the JSP-5 system for this initial configuration.

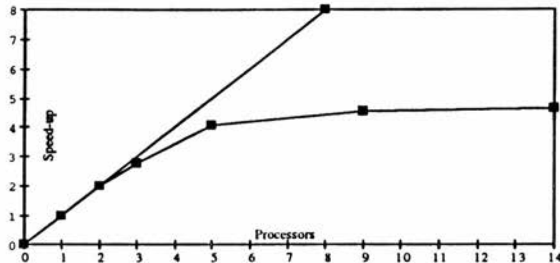

Figure 6.6. The JSP-5(14) System Speed-up (D:1, K:7, B:3, I:3).

The biggest speed-up that we could obtain, by placing each process in a separate processor was of 4.36. Initially, the speed-up is almost linear, up to five processors. It then flattens up, and the increases in performance are not significant. Note that the speed-up obtained is similar to the results presented in [RAN89] for the CAGE system.

From the module utilisation table presented in table 6.2, we can see that the JSP-4 system uses heavily the global blackboard structure. We designed a second ABB configuration with an emphasis on blackboard utilisation. We separated the burden of handling each blackboard level to three blackboard managers. This was done for the two most used levels. For the third level, we used two blackboard managers.

This second configuration, then, consists of eight blackboard managers, three indexes, seven knowledge sources and one desk. This configuration uses nineteen processes. We will call it the *JSP-5(19)* system.

Figure 6.7 shows the speedup from the user point of view for the JSP-5(19) system. In order to make comparisons possible, we used the JSP-5(14) run time for one processor as the reference run time. The graph then shows the number of times the parallel version is with respect to this reference.

We can appreciate a dramatic increase in performance. The maximum speed-up was of 15.9 for thirteen processors. After that number of processors, the speed-up decreases. This results show that the main bottleneck in the JSP-5 system is the blackboard manipulation module.

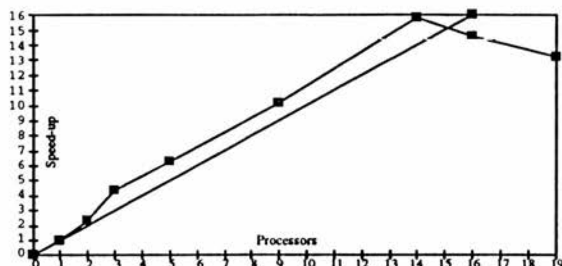

Figure 6.7. The JSP-5(19) System Speed-up (D:1, K:7, B:8, I:3).

Figure 6.7 shows a superlinear increase in performance until it decreases with the use of sixteen processors. This superlinear speedup is theoretically impossible. However, differences in the uni-processor, and multi-processor systems can yield superlinear results. This means that the multi-processor implementation is more efficient than the uni-processor one.

The actual C code for the JSP-5(14) system, and the JSP-5(19) system is exactly the same. The only variation is a slightly bigger process reference table introduced by the increased number of bbel managers in the system. These extra bbel managers are independent processes identical to the bbel managers in the JSP-5(14) system. We can discard the number of processes as a cause for this behaviour, as it is not present in the THR(20) system (Figure 6.5). The JSP-5(19) uses nineteen processes, and the THR(20) system uses fifty-three.

The JSP-5(14), and JSP-5(19) perform the same amount of inter-process communications. The only difference between the JSP-5(14) running on one processor, and the JSP-5(19) system running on several processors is the amount of inter-process communication within one processor. We measured the time taken to transmit one KSAR between processes for those systems. The average time taken by the JSP-5(14), executing on one processor, to transmit a KSAR was of 0.0052 seconds. The average time taken by the JSP-5(19) running on twenty-one processors was of 0.0018 seconds. This means that communication within one processor is 2.89 times slower that inter-processor communication. This explains the superlinear speed-up results. As the JSP-5 systems are blackboard-bound systems, they stress the communication capabilities of the system, and thus they are more sensitive to communication time variations. This is why the superlinear speedup shows itself for these systems.

Note that the JSP-5(19) system still uses only one desk. This means that the

knowledge source body execution is serialised. The JSP system knowledge sources make five simultaneous modifications to the blackboard in order to place a given piece in the board. If these modifications are performed in parallel, we would expect a five-fold speed-up. To this, we must add the speed-up obtained by performing the knowledge source trigger operations and the blackboard control functions in parallel. However, the overall effect of parallelising the blackboard is to produce a more efficient blackboard manipulation module. It is possible that a significant speed-up can be obtained by optimising the blackboard manipulation module. Depending on the application, and the resources available, these optimisations may be a better approach to increasing the performance of applications similar to the JSP-5 system. After these optimisations are performed, we could consider the use of parallelism to increase further the system performance.

The final JSP-5 system configuration allows each piece knowledge source to execute independently on its own desk. This configuration uses twenty-four processes. These are the nineteen processes from the previous configuration, plus five more desks. The desk of the previous configuration will handle the initialisation and piece misplaced knowledge sources. Figure 6.8 shows the application speed-up for the JSP-5(24) system.

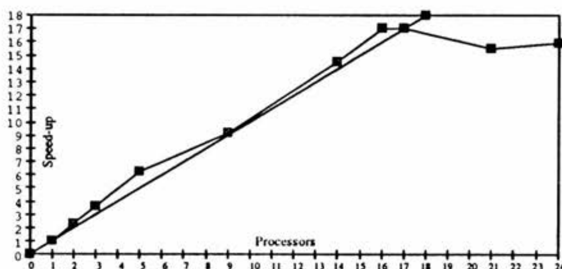

Figure 6.8. The JSP-5(24) System Speed-up. (D:6, K:7, B:8, I:3)

We could obtain a maximum speed-up of 16.9. This increased the speed-up achieved by the previous configuration. However, we can see that knowledge source parallelism is not as important as blackboard parallelism for the JSP-5 system.

6.4 Conclusions

In the present chapter, we have shown the performance results for the ABB parallel blackboard prototype. We have shown that an order of magnitude increase in performance can be achieved for two very different blackboard systems. These blackboard systems were designed to test the different blackboard architecture characteristics, and their behaviour in a parallel environment.

We have shown that the performance for the ABB parallel prototype, running on one processor is better than that of the sequential and distributed blackboard shells studied in Chapter 3. This, of course, could be attributed to the large differences in implementation language, processor type, and operating system. The ABB prototype has the advantage of being implemented in the C language, in contrast with LISP (BB1), PROLOG (EPBS), and POP-11 (Blondie-I and III). However, the ABB system runs on a Transputer T-800. This is a very slow machine compared to the Micro-Explorer Lisp Machine (BB1), and Sun's SPARC (EPBS, Blondie-I and III), as shown in [Don89].

We then presented the performance of the ABB parallel prototype on different number of processors. We achieved one order of magnitude increase in performance for the different test systems.

The Tread Simulation test systems show that the ABB prototype can be configured to achieve the complete use of the parallelism inherent in the different applications. Note that this is achieved without losing the functionality and power of the blackboard architecture, including its knowledge source control capabilities.

The Jigsaw-Puzzle Solver is an example of a blackboard-bound application. The application implementation presented in this chapter, in itself, does not present a large amount of inherent parallelism. This possible parallelism is restricted by the number of knowledge sources, and the data itself. However, we achieved a ten-fold speed-up by making use of blackboard parallelism.

The different configurations of the Jigsaw-Puzzle Solver show that it is possible to adapt the ABB prototype to specific application needs. We can increase the blackboard system performance by using both blackboard-level parallelism and knowledge source parallelism. The blackboard level parallelism is achieved by defining more blackboard managers per application level. The knowledge source parallelism is allowed by using several desks. Note that these configuration decisions can be taken after the actual system implementation is finished. The blackboard system code does not change by these changes. The only changes are in the number of processes that will be used to

run it.

In the present chapter, we have demonstrated that significant performance improvements can be achieved by making use of parallel computations within the active blackboard model. We have also shown that significant improvements can be simply found by changing configuration parameters (without recoding).

Chapter 7

Conclusions

7.1 Summary and Conclusions

In this thesis, our main goal was to show that blackboard system performance can be improved by the use of a new blackboard architecture. The structure of this thesis was designed to show the steps followed in order to demonstrate that blackboard system performance can, indeed, be improved.

In the first chapter we presented an introduction to the blackboard system field. We presented the basic metaphor in which it is founded, and the most common interpretations of the blackboard architecture.

In Chapter 2 and Chapter 3 we concentrated on studying the performance of traditional blackboard systems. In Chapter 2 we presented a performance framework in which we can describe blackboard system functionality and performance. This framework is a result in itself. It is general enough to allow the global description of widely different blackboard systems and shows their global performance, as well as the performance of the individual conceptual system modules.

In this thesis, we used this framework to study the performance of several blackboard systems present in the literature. This study was presented in Chapter 3. The performance framework allowed us to see the behavior of the different systems, and allowed us to draw conclusions about the impact that the design decisions, taken by those systems, have on the overall system performance.

The usefulness of the framework, however, is not restricted to the actual use made of it in this thesis. When we were experimenting with our own blackboard architecture, we found that the performance measurements introduced within the framework were extremely useful when designing and implementing blackboard systems. These measurements allow the system implementor and designer to identify easily problem areas from the performance point of view, and to direct optimization efforts in an informed manner.

The use of the performance framework, integrated within a blackboard system development methodology is, in itself, an excellent approach for blackboard system performance improvement. We consider that similar schemes to the one presented in this thesis should be included as analysis tools within all blackboard system shells.

The experience gathered in these first chapters, as well as the review of the extensive literature in blackboard systems, took us to choose one of the most promising approaches for the improvement blackboard system performance. This approach was to make appropriate use of parallelism within the blackboard architecture.

The use of parallel and distributed systems to improve blackboard system performance is not new in itself. The major contribution of this thesis is the design of a new blackboard architecture that makes efficient use of the resources provided by modern parallel computers, without losing any the blackboard architecture main characteristics.

The approach that we followed to arrive to the new architecture was to abstract the blackboard architecture away from any implementation issues. For this we needed to go back to the original blackboard and experts' metaphor, and visualize how the problem solving activity would work when there are no sequential restrictions. This took us to propose a small change on the basic metaphor that helped us to visualize in an intuitive fashion, how the concurrent blackboard problem solving activity works. This change was the introduction of work areas, or desks, within the metaphor.

In Chapter 4, we introduced the blackboard, experts and desks' metaphor. This metaphor was used to develop a new original blackboard model. We called this the *Active Blackboard Model* (ABB). The main features of this model are the use of a fully distributed control scheme, and the definition of the blackboard as an active entity within the global architecture. We consider it to be one of the main contributions of this thesis. We used the specification language Z for the formal description of the model. This allowed us to describe, without ambiguities, all the different aspects of the model, as well as the formal description of the properties that all active blackboard implementations should satisfy.

We then described a parallel implementation of the active blackboard model. The prototype presented in Chapter 5 is an object oriented parallel blackboard system shell that implements the active blackboard model. This prototype also presents a number of novel characteristics; some of them derived from its object oriented design. In the ABB prototype, the blackboard elements, knowledge sources and desks are independent active objects. They all manage their own structure, which is active for the whole object life. This characteristic is quite common for blackboard elements; however, it is novel for knowledge sources and desks. This allows knowledge sources and desks to have a "memory" of their own. They can use their internal structure to "remember" information between evaluations. This feature opens a wide range of possibilities. It provides a form of knowledge source and desk persistence, and can be used to implement for more efficient and "intelligent" knowledge sources.

In Chapter 6, we presented the performance results for the active blackboard prototype implemented on the Edinburgh Parallel Computer Center's MEIKO Computing Surface; a Transputer-based distributed memory multiprocessor. In this chapter we demonstrated that one order of magnitude increase in performance can be achieved

within the active blackboard prototype, by making use of parallel processing. This, of course, will depend on the application and its implementation. However, the prototype used in conjunction with the performance framework, provides an extremely efficient and powerful environment for the implementation of blackboard systems with excellent performance.

We consider the Active Blackboard Model and its prototype implementation to be the main contributions of this thesis. Besides the functionality and facilities that they provide, they open a new set of research issues that need to be addressed. In the following section, we will discuss some of these issues.

7.2 Further Work

The development of the blackboard performance framework and of the active blackboard architecture raises a number of interesting research issues that need to be addressed. These issues could be classified into implementation related issues, and architecture related ones.

7.2.1 Implementation Related Issues

When a knowledge engineer begins the development of a new blackboard system using the parallel ABB prototype, he or she will face a number of implementation decisions. We could enumerate these decisions as follows:

- Number and structure of the knowledge sources.

  The knowledge engineer has to study the system problem domain and has to decide on some sort of modularization of it. He or she has to identify the knowledge sources involved in the resolution of the domain problem.

- Structure of the global blackboard.

  The knowledge engineer also has to decide on the common language between knowledge sources. He or she has to decide on the structure of the global blackboard. That is, he or she has to decide on the number of blackboard levels, and the structure and functionality of the elements held in them.

- Control structure and architecture.

  There are problem domains in which some of the problem domain knowledge is in the form of control knowledge or meta-knowledge. This meta-knowledge is also

useful to actually direct and focus the problem solving activity. The knowledge engineer also has to decide on the structure and functionality of the system's control structure.

The previous three decisions are common in all blackboard system implementations. The ABB prototype provides rich object-oriented features to support each one of these three decision steps. The prototype, however, introduces two more necessary steps. These are:

- Blackboard manager and desk configuration.

  Once the previous implementation decisions have been taken, the knowledge engineer has to decide on the implementation architecture that the new blackboard system should have. This decision involves the choice of the number of blackboard managers that is most appropriate for each blackboard level of the system, the number of desks of the final system, and which knowledge sources should manage each of the desks.

- Process to processor allocation.

  The final decision that needs to be taken by the knowledge engineer is the allocation of all the processes that compose the active blackboard system being developed onto the available procesors. Although any configuration should work, there will be some configurations that are more efficient than others. The effect of a bad process to processor assignment in overall system performance can be extremely large, as it affects interprocess communication times.

These two aspects need to be studied further. We think that it should be possible to produce automatic algorithms that either produce an excellent initial configuration, or that are able to adapt while the problem-solving activity is in process.

When we studied the performance of the prototype, specially with the Jigsaw Puzzle Solver, we started with a initial system configuration. For the jigsaw-puzzle solver we started with a one-desk and one blackboard manager per level configuration. We found that the performance measurements introduced in the blackboard performance framework were very useful to identify problems in the configuration, and allowed us to make decisions as to where new blackboard managers should be added. We think that the use of detailed performance measurements can be used to aid in the general system configuration. They could even be used, together with facilities for process migration and creation, to adapt the overall system architecture while the problem

solving activity is in process. This would allow the system to respond in a more efficient way while solving different situations. This, however, needs to be evaluated. Also, the effect of the extra implementation complexity in the system performance is not clear.

One of the biggest problems that we faced when producing the results presented in Chapter 6 was how to assign the different processes that make up the blackboard system application, so as to achieve the best performance possible. We followed a very empirical approach based on the experience gathered after a large number of trials. However, we consider that a more direct and automatic approach to process-processor allocation is possible.

The process-processor allocation problem is addressed in the parallel processing field. A number of algorithms have been proposed. They all use a number of general heuristics to allocate the different processes to the processors. For example, MEIKO's `mrun` program implements some heuristics to provide a basic allocation. However, as these heuristics do not always produce good results, the program provides a number of directives that allows the user to describe the desired configuration. This, however, translates the decision of where the application processes should be allocated to the system implementor.

The active blackboard prototype produces a well-defined set of processes, with a very specific set of needs. It should be possible, taking into account blackboard system needs, to produce a set of heuristics to produce a good process allocation.

Alternatively, the system could allow dynamic task (or object) migration based on the performance framework measurements, or on measurements of the different module's communication times. This may produce a more flexible and adaptable system that modifies the processor's load while the problem solving activity is in process. In this case, the effect of such process migration on the overall system performance should be assessed.

7.2.2 Architecture Related Issues

Besides the previously presented implementation issues, the ABB model raises a number of problems from the high-level architecture point of view. From these, the most important one is the control problem.

The ABB Model lays out a framework in which parallel and distributed problem solving control architectures can be developed. The specific control functions at each level of the blackboard architecture, the blackboard, knowledge source, and desk control functions, provide the basic functionality that can then be used by a more complex

control architecture.

There is the need to reproduce some of the work in blackboard control architectures, using the ABB distributed control decision making as the basis, instead of an agenda based centralised scheduler. We think that a high level control structure similar to BB1, or more current architectures such as O-Plan [CT91] can be implemented on top of the basic ABB control functions.

In order to develop these high level control architectures on top of the ABB Model control functions, there is one difficult issue that must be adressed. This issue is how to map global system strategies or policies down to local decisions. The ABB Model control structure is based on the separation of the specific control decisions within the blackboard architecture. These decisions are then distributed and taken in a local context. If we want to build a global control architecture, we need to investigate the way global directives affect local decisions.

Bibliography

[Aie83] Nelleke Aiello. A comparative study of control strategies for expert systems: AGE implementation of three variations of PUFF. Report HPP-83-33, Heuristic Programming Project, Stanford University, Stanford, California 94305, 1983.

[Aie86] Nelleke Aiello. User-directed control of parallelism; the CAGE system. Technical Report KSL 86-31, Knowledge Systems Laboratory, Stanford University, Stanford, California 94305, 1986.

[Amd67] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[AMM87] Garvey Allan, Hewett Michael, and Vaugham Johnson Jr. M. BB1 user manual – common LISP version 2.0. Report KSL 86-61, Knowledge Systems Laboratory, Stanford University, 1987.

[aT86] Luiz V. Le ao and Sarosh N. Talukdar. An environment for rule-based blackboards and distributed problem solving. *Artificial Intelligence in Engineering*, 1(2):70–79, 1986.

[BJD86] M. Benda, V. Jagannathan, and R. Dodhiawala. On optimal cooperation of knowledge sources: an empirical investigation. Internal report, Boeing Advanced Technology Center, Boeing Computer Services, Seattle, Washington, 1986.

[BJL+84] B. Buchanan, O. Jardetzky, O. Lichtarge, B. Hayes-Roth, R. Altman, and M. Hewett. PROTEAN. Technical report, Stanford University, Stanford, CA., 1984.

[BRS85] R. Barrett, A. Ramsay, and A. Sloman. *POP-11 A practical Language for Artificial Intelligence.* Ellis Horwood Limited, Chichester, U.K., 1985.

[BW77] Daniel G. Bobrow and Terry Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):3–46, 1977. Also published in Readings In Knowledge Representation, Brachman and Levesque editors (1985).

[Cra89] Iain D. Craig. *The CASSANDRA Architecture: Distributed Control in a Blackboard System.* Ellis Horwood, England, 1989.

[Cra91] Iain D. Craig. *The Formal Specification of Advanced AI Architectures.* Ellis Horwood, Chichester, 1991.

[CT86] E. Cardozo and S. N. Talukdar. A distributed control strategy for energy management centers. Internal Report EDRC-05-07-86, Carnegie-Mellon University, 1986.

[CT91] K.W. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 51(1), Autumn 1991.

[Dav87] Andrew Davidson. Blackboard systems in PARLOG. Internal report, The PARLOG Group, Dept. of Computing, Imperial College, 1987.

[Der87] P. Derrington. Management of the design process. In D. Sriram and R.A. Adey, editors, *KBES in engineering: Planning and Design*, pages 19–33. Computational Mechanics Publications, 1987.

[DGHL90] Keith Decker, Alan Garvey, Marty Humphrey, and Victor Lesser. Effects of parallelism on blackboard system scheduling. In *Proceedings of the Fourth Annual Workshop on Blackboard Systems*, 1990.

[Don89] Jack J. Dongarra. Performance of various computers using standard linear equations software. Tech. Report CS-89-85, Computer Science Department, University of Tenessee, 1989.

[DS83] R. Davis and R. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

[EG85] J. Robert Ensor and John D. Gabbe. Transactional blackboards. In *Proceedings of the IJCAI-85*, pages 340–344, 1985.

[EG86] J. Robert Ensor and John D. Gabbe. Transactional blackboards. *Artificial Intelligence in Engineering*, 1(2):81–84, 1986.

[EHRLR80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys, ACM*, 12(2):213–253, June 1980.

[EL75] Lee D. Erman and Victor R. Lesser. A multilevel organisation for problem solving using many, diverse, cooperating sources of knowledge. In *Proceedings of the IJCAI-75*, pages 483–490, Tbilisi, USSR, 1975.

[EL78] Lee D. Erman and Victor R. Lesser. Hearsay-II tutorial introduction and retrospective view. Tech. Report CMU-CS-78-117, Department of Computer Science, Carnegie Mellon University, 1978.

[ELF81] Lee D. Erman, Phillip E. London, and Stephen F. Fickas. The design and an example of use of HEARSAY-III. In *Proceedings of the IJCAI-81*, pages 409–415, August 1981.

[Elf86] Alberto Elfes. A distributed control architecture for an autonomous movile robot. *Artificial Intelligence in Engineering*, 1(2):99–108, 1986.

[For82] Charles L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 1(19):17–37, 1982.

[FvLV88] A. Florescu, E.P.M. van Liempd, and H. Velthuijsen. BLONDIE–III: An environment for distributed problem solving. Internal Report 88-DNL/16, Dr Neher Laboratories, 1988.

[GBPT88] J.N. Gray, R.A. Brie, G.R. Putzolu, and I.L. Traider. Granularity of locks and degrees of consistency in a shared database. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 94–121. Morgan Kaufmann, 1988.

[GC89] Kevin Q. Gallager and Daniel D. Corkil. Performance aspects of gbb. In V. Jagannathan, R. Dodhiawala, and L.W. Baum, editors, *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[GCJ88] Kevin Q. Gallager, Daniel D. Corkill, and Philip M. Johnson. GBB reference manual, version 1.2. COINS Technical Report 88-66, University of Massachusetts at Amherst, 1988.

[Gel89] Erol Gelembe. *Multiprocessor Performance*. John Willey & Sons, England, 1989.

[GHR89] Alan Garvey and Barbara Hayes-Roth. An empirical anaysis of explicit vs. implicit control architectures. In V. Jagannathan, R. Dodhiawala, and L.W. Baum, editors, *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[Gre87] Peter E. Green. AF: a framework for real-time distributed cooperative problem solving. In Michael N. Huhns, editor, *Distributed artificial intelligence*. Pitman, London, 1987.

[GRT89] John F. Gilmore, Stefan P. Roth, and Stepthen D. Tynor. A blackboard system for distributed problem solving. In V. Jagannathan, R. Dodhiawala, and L.W. Baum, editors, *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[Gus88] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[Her88] Luis Castillo Hern. On distributed artificial intelligence. Report AIAI-PR-17, Artificial Intelligence Applications Institute, University of Edinburgh, January 1988.

[Hew77] C.E. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence Journal*, 8:323–364, June 1977.

[HR83] Barbara Hayes-Roth. The blackboard architecture: A general framework for problem solving? Report HPP-83-30, Heuristic Programming Project, Stanford University, 1983.

[HR84] Barbara Hayes-Roth. BB1: An architecture for blackboard systems that control, explain and learn about their own behaviour. Report HPP-84-16, Heuristic Programming Project, Stanford University, 1984.

[HR87] Barbara Hayes-Roth. A multi-processor interrupt-driven architecture for adaptive intelligent systems. Technical Report KSL 87-31, Knowledge Systems Laboratory, Stanford University, Stanford, California 94305, 1987.

[HRH85] Barbara Hayes-Roth and Michael Hewett. Learning control heuristics in BB1. Report HPP-85-2, Heuristic Programming Project, Stanford University, 1985.

[HRHRRC79] B. Hayes-Roth, F. Hayes-Roth, S. Rosenschein, and S. Cammarata. Modelling planning as an incremental, opportunistic process. In *Proceedings of the IJCAI-79*, pages 375–383, 1979.

[Huh87] Michael N. Huhns, editor. *Distributed artificial intelligence*. Pitman, London, 1987.

[Jag89] Vasudevan Jagannathan. Realizing the concurrent blackboard model. In V. Jagannathan, R. Dodhiawala, and L.W. Baum, editors, *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[JHR87] M. Vaughan Johnson Jr. and Barbara Hayes-Roth. Integrating diverse reasoning methods in the BB1 blackboard control architecture. In *Proceedings of the AAAI-87*, pages 30–35, 1987.

[JM86] John Jones and Mark Millington. An edinburgh PROLOG blackboard shell. DAI research report, Department of Artificial Intelligence, University of Edinburgh, 1986.

[JMR86] John Jones, Mark Millington, and Peter Ross. A blackboard shell in PROLOG. DAI Research Report 277, Department of Artificial Intelligence, University of Edinburgh, 1986.

[KR88] H.T. Kung and John T. Robinson. On optimistic methods for concurrency control. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 122–128. Morgan Kaufmann, 1988.

[LC83] Victor R. Lesser and Daniel D. Corkill. THE DISTRIBUTED VEHICLE MONITORING TESTBED: a tool for investigating distributed problem solving methods. *The AI Magazine*, pages 15–33, Fall 1983.

[LE77] Victor R. Lesser and L. D. Erman. A retrospective view of the Hearsay-II architecture. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 790–800, 1977.

[LE80] Victor R. Lesser and L. D. Erman. An experiment in distributed interpretation. *IEEE Transactions on Computers*, 29(12):1144–1163, December 1980.

[LVV86] B.J. Lippolt, H. Velthuijsen, and J.C. Vonk. BLONDIE: a blackboard shell. Internal Report 1404-DNL/86, Dr Neher Laboratories, 1986.

[NA79] H. Penny Nii and N. Aiello. AGE (attempt to generalise): A knowledge-based program for building knowledge-based programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-79)*, pages 645–655, Tokyo, 1979.

[NAR89] H. Penny Nii, Nelleke Aiello, and James Rice. Experiments on CAGE and POLIGON: Measuring the performance of parallel blackboard systems. In Les Gasser and Michael N. Huhns, editors, *Distributed artificial intelligence, Volume II.* Pitman, London, 1989.

[New69] Allen Newell. Heuristic programming: Ill-structured problems. In J. Aronofsky, editor, *Progress in Operations Research*, pages 360–414. John Wiley, New York, 1969.

[NFAR82] H. Penny Nii, Edward A. Feigenbaum, John J. Anton, and A.J. Rockmore. Signal-to-symbol transformation: HASP/SIAP case study. Internal Report HPP-82-6, Heuristic Programming Project, Stanford University, 1982.

[Nii80] H. Penny Nii. An introduction to knowledge engineering, blackboard model and AGE. Internal Report HPP-80-29, Heuristic Programming Project, Stanford University, 1980.

[Nii86a] H. Penny Nii. Blackboard systems: Blackboard application systems from an knowledge engineering perspective. *The AI Magazine*, pages 82–106, August 1986.

[Nii86b] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *The AI Magazine*, pages 38–53, Summer 1986.

[PSC+86] Robin Poplestone, Tim Smithers, Jonathan Corney, Anastasia Koutsou, Karl Millington, and Gideon Sahar. Engineering design support systems. DAI research report, Department of Artificial intelligence, University of Edinburgh, 1986.

[RAN89] James Rice, Nelleke Aiello, and H. Penny Nii. See how they run... the architecture and performance of two concurrent blackboard systems. In V. Jagannathan, R. Dodhiawala, and L.W. Baum, editors, *Blackboard Architectures and Applications.* Academic Press, Boston, 1989.

[RG77] R. Bruce Roberts and Ira P. Goldstein. The FRL primer. Technical Report AIM-408, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., 1977.

[Ric86] James P. Rice. Poligon, a system for parallel problem solving. Technical Report KSL 86-19, Knowledge Systems Laboratory, Stanford University, Stanford, California 94305, 1986.

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Keneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, pages 10–18, January 1986.

[Spi89] J. M. Spivey. *The Z Notation: a reference manual.* Prentice Hall International (UK) Ltd., London, 1989.

[Sri86] D. Sriram. DESTINY: A model for integrated structural design. *Artificial Intelligence in Engineering*, 1(2):109–116, 1986.

[Ter83] A. Terry. The CRYSALIS project: Hierarchical control of production systems. Technical Report HPP-83-19, Heuristic Programming Project, Stanford University, 1983.

[Ull82] Jeffrey D. Ullman. *Principles of database systems.* Computer science press, Inc., Maryland, 1982.

[VC86] V. Venkatasubramanian and Chun-Fu Chen. A blackboard architecture for plastics design. *Artificial Intelligence in Engineering*, 1(2):117–122, 1986.

[Vel91] H. Velthuijsen. *The Nature and Applicability of the Blackboard Architecture.* PhD thesis, September 1991.

Appendix A

A Brief Introduction to Blackboard Systems

A.1 Introduction

There is a number of problems in Artificial Intelligence (AI) for which there is no precise path that allows to reach a given goal. This is, there is no defined procedure or empirical method with which we can achieve a given goal, or solve a particular problem. Other systems have badly defined criteria for deciding when the goal has been achieved, or there is badly defined relevant knowledge. All those problems have been defined as *ill-structured problems* ([New69]).

To solve ill-structured problems, an AI system must deal with very large search spaces, complex information interdependencies, and goals that are difficult to define. AI systems may need to manipulate knowledge that is sometimes incomplete and approximate, and originated on different areas of expertise. They may also need to use different reasoning methods simultaneously.

Blackboard systems have been designed to deal with the ill-structured problems. The blackboard model provides the platform to organise knowledge from different areas ([Nii80] [EG86]), including incomplete and approximate knowledge ([EL75]) and "noisy" data ([EHRLR80] [NFAR82] [HR83]). They can manipulate complex information interdependencies ([EG86] [Der87]), make use of many different reasoning methods ([JHR87]), manage a large search space ([Nii80] [EHRLR80]) and ill-defined goals ([Der87]).

A broad spectrum of applications has been developed using the blackboard problem solving architecture. The Hearsay-II system for speech understanding ([EL75]) was the first system developed using the blackboard framework. This system introduced the new problem solving paradigm, and showed it to be general enough to be applied in other problem domains. Using the Hearsay-II experiences, several other systems were developed in widely different problem domains such as ocean surveillance ([NFAR82]), protein analysis ([Ter83] [BJL+84]), planning ([HRHRRC79]), design ([Der87] [PSC+86] [VC86] [Sri86]), medical diagnosis ([Aie83]), vehicle monitoring ([LC83]) and autonomous mobile robots ([Elf86]). The experiences of some of those systems were used in developing general all-purpose expert system shells in order to facilitate the development of blackboard systems. Examples of those systems are AGE ([NA79]), HEARSAY-III ([ELF81]), Stanford's BB1 ([HR84]), Boeing's BBB ([BJD86]) and Edinburgh's EPBS ([JM86]),

A.2 The blackboard model

The blackboard model is a problem solving paradigm in artificial intelligence that establishes a high-level organization of the factual knowledge needed for describing a problem, its solutions, and the procedural knowledge about how to solve the problem. The model also establishes the use of procedural knowledge for incremental and opportunistic problem solving. The basic blackboard model has two components:

- A global database called **blackboard** that stores the description of the problem, its possible solutions and partial solutions, as well as facts of general interest (factual knowledge). All this constitutes a description of the current state of the problem solving process.
- A set of independent **knowledge sources** that use and modify the data kept in the blackboard (procedural knowledge).

Before describing the components of the basic blackboard model further, let us illustrate its problem-solving behavior through an example.

Suppose we have to solve a complex problem, say the five year planning of the sowing in a farm, in order to maximise profits. To do this, we may need the help of several experts from different specialities. We will need an agricultural expert to propose which plants to grow and the appropriate choice of fertilisers needed. We will also need a marketing expert to provide prices of the different materials needed, as well as the probable final product price in the market, and a financial expert to analyse the probable incomes and outcomes of each proposed solution and to choose the best one.

We then prepare a room with all the furniture needed for a technical meeting: some desks, chairs and a big blackboard. Once contacted, we gather our experts into the room and we ask them to produce a good solution to our problem. In order to work without distracting each other, the group of experts decide to work silently. They decide to use the blackboard provided to make public any particular conclusion. In this way, they can reach further conclusions based on each other's work.

The experts begin by writing on the blackboard all the known facts such as soil type, usual weather, capacity and location of water sources, etc. Then each one of them, looking at the blackboard, decides if there is any contribution to make. If there is, the experts write it on the blackboard. In order to avoid confusion, they decide to organize the information on the blackboard so that all the related contributions are written close to each other. They decide to have one partition for the original known

facts, one for plants and fertilisers, another for prices and a last one for general data such as 'goodness' of partial solutions.

The agricultural expert can place in the blackboard the best plants to grow, as well as the fertilisers needed, based on the soil type and water source information. Meanwhile, the marketing expert can write down which agricultural products are most wanted in the market. The agricultural expert can then discard those he considers to be inappropriate. The marketing expert can also write down the known fertiliser and seed prices and estimate product prices for the plants already in the blackboard. The financial expert can monitor the blackboard information, discard any uneconomical choices and decide about the 'goodness' of partial solutions. When no new partial solutions are produced, the solution with the highest 'goodness' is chosen to be the answer to the problem.

This small example has some particular characteristics that makes it well suited to be solved using the blackboard model, among these features we may enumerate:

- A large solution space. There is a large amount of possible sowing plans, not all of them feasible or profitable.
- The input data is diverse. There is soil type, weather, types of plants, fertilisers, profitable crops, etc.
- It is necessary to integrate diverse information in such a way that the global solution "evolves" from partial solutions.
- It is necessary the cooperation of several semi-independent experts (sources of knowledge) to achieve a good solution
- The need of multiple reasoning methods.

In the example we can identify the two main components of the blackboard model:

- Several experts (*knowledge sources*) are interacting to solve the problem ([EL75] [HR83] [Nii86a])
- The experts work on a globally accessible data structure, the *blackboard*, which holds the current state of the problem-solving activity.

We can see that the blackboard holds diverse factual knowledge; this is, information about the problem, about the data needed to solve the problem, about its partial solutions, work hypothesis, and any other fact the experts feel the need to make public.

The blackboard also structures and organizes this knowledge in partitions, making it more modular and comprehensive, providing a common framework for knowledge source interaction. Figure A.1 shows the different blackboard partitions for our example, as well as their relation with the experts.

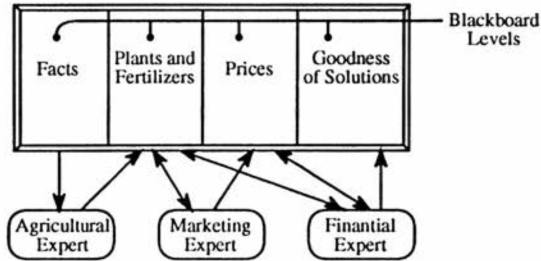

Figure A.1. Experts and Blackboard

We can see, also in the example, that the knowledge sources (experts) are independent of each other, and that the only mean of communication between two of them is the blackboard. The problem solving activity is *opportunistic*. Each expert, when seeing enough information to draw a conclusion, goes to the blackboard and writes it down. There can be experts that take known facts and conclude something towards the solution of the greater problem (forward-chaining), experts that identify problems and decompose them into smaller problems (backward-chaining), and experts that follow some other approach, all interacting and working at the same time.

The knowledge sources are a mean of organizing the procedural expertise about how to solve a particular problem. They usually hold the expertise about how to solve a specific sub-problem within the greater problem. In the example, the experts do not know how to solve the overall problem; they are specialised in solving small problems within it.

The basic model provides means of expressing factual knowledge (blackboard) and procedural knowledge (knowledge sources). It also provides a general problem-solving behavior based on the knowledge sources looking for problems in the blackboard and placing solutions in it. The solutions placed by some knowledge sources in the blackboard will probably give other knowledge sources clues as to how to solve other problems. This cycle is repeated until the desired problem is solved.

This model, however, does not specify an explicit mechanism for encoding the expertise about the reasoning process itself and how it affects the general problem

solving behavior. This expertise is called **control knowledge**.

Suppose, in the example, that all our experts decide to change or add data in the plants and fertilisers level of the blackboard at the same time. They may have some problems doing so: they may run out of chalk or they may want to modify the same item in the blackboard. To avoid conflicts they decide to modify the blackboard in an orderly fashion, so that only one expert can have access to it at any one time. They decide to form a queue. Each expert will place on one of the desks a note saying that he (or she) wants to use the blackboard. The notes are then picked up in a first-in first-out fashion by an assistant, called the *scheduler*. The expert whose note has been chosen is allowed to use the blackboard.

The scheduler's job is quite boring. He begins thinking of different ways of improving the group's performance based on the experience he has gathered by observing the problem-solving process. He decides to ask the experts to write down not only that they are interested in using the blackboard, but also the partition they want to use, and the nature of the modifications to be performed. With this extra information the scheduler can direct the expert activity to some extent by applying control policies. For example, he can identify groups of notes of experts trying to solve related problems and he can then give preference to those notes concentrating the expert's efforts towards the solution of particular sub-problems that will eventually lead to the solution of the main problem. This kind of policy is called *focus of attention*.

The problem-solving activity of the group can be seen as the iteration of the following sequence of steps:

- The scheduler chooses one pending note.
- The scheduler allows the corresponding expert to use the Blackboard.
- The experts that can now use the blackboard because of new conclusions place new notes in the queue.

In a blackboard system, the 'chalk' can be seen as the permission to use the blackboard structure. "To have the chalk", is being able to modify the blackboard. The queue of notes is a simple version of the *agenda* used in most blackboard systems ([EHRLR80] [Sri86] [HRHRRC79]). The notes used by the experts can be seen as the *knowledge source activation records* (*KSARs*) used by many systems.

A.3 Blackboard System Components

We used the farm planning example to introduce, in an intuitive way, the different concepts and components of the blackboard architecture. We will now give a more complete description of each one of these components and their interactions within a typical blackboard system. We will compare them with some blackboard systems developed and discuss the different approaches taken.

A.3.1 The knowledge sources

The knowledge sources contain the system's domain-specific procedural knowledge. They hold all the knowledge about how to solve a particular problem. Although they have the capacity, individual knowledge sources usually do not hold enough knowledge as for solving the global problem. One of the strengths of the blackboard architecture is that the solution of the overall problem can be decomposed in smaller, simpler problems that will be tackled by individual knowledge sources, or a group of them. This modularises and organises the procedural knowledge on a system, as well as it conceptually simplifies the solution of the global problem.

The individual knowledge sources have a "**IF** *condition* **THEN** *action*", structure, similar to a production rule. This structure reflects the way they operate. Just as the experts they represent; a knowledge source knows how to solve a small problem within a greater, global task. Its main task is to search the blackboard for suitable problems, solve them and place the solution back in the blackboard. When a knowledge source identifies a problem that it can solve (its condition is satisfied), it is said that it is "*triggered*".

The condition is the representation of the knowledge about how to identify appropriate problems. Likewise, the action represents the knowledge about how to solve the problem. Once the problem is solved, the action itself places the result on the blackboard. Knowledge sources are the only entities that produce changes to the blackboard contributing incrementally and cooperatively to the solution of the global problem.

Knowledge sources are independent and do not produce side-effects other than the modifications to the blackboard. They do not invoke one another directly, and they do not have an a priori knowledge of each other expertise, behavior, or existence. In blackboard systems they are typically represented as either procedures, logic assertions or groups of production rules. Knowledge sources may exploit any inference method such as top-down, bottom-up, etc.

A.3.2 The blackboard

The blackboard holds different objects of the solution space: input data, partial solutions, final solutions and possibly control data. All this constitutes all the facts about the current state of the problem and the problem solving activity known to the system.

The data stored in the blackboard are sometimes called *units* ([EL75] [ELF81]), *entries* ([HR83] [JMR86] [Sri86]), *objects* ([Nii86a] [Nii86b]) or simply *data* ([EG85]).

The blackboard information is organised and structured in different blackboard partitions (as in our example), sometimes called *information levels* ([EL75] [EHRLR80]), *abstraction levels* ([HR83] [Sri86] [JHR87]), *levels of analysis* ([Nii80] [NFAR82]) or just *levels* ([HR84]). Although it is generally accepted that blackboard systems have one global data structure called blackboard, there are systems that allow to have more than one blackboard ([AMM87], [GCJ88]). This provides one further level in the organization of blackboard data objects.

The data objects can be related or connected among themselves via *links*. These links denote relations between objects of different levels, such as "part-of" or relations, and between objects of the same level, such as "next-to". This can be used to organise hierarchically elements of the same or different levels.

Having the blackboard data elements organised into levels allows to have different vocabularies associated with different concepts that can be explicit and accessible at the appropriate level of abstraction. A knowledge source only needs to manipulate the vocabularies associated with its working levels.

The blackboard also constitutes the only communication mechanism among the knowledge sources. Its structure and organisation will be used, also, to represent a common vocabulary with which knowledge sources communicate with each other.

A.3.3 The Scheduler

The scheduler is the mechanism used in most blackboard systems for encoding control knowledge. It is the element of the blackboard model entrusted with the task of monitoring the changes on the blackboard and deciding what actions to take next. These actions usually consist on deciding which knowledge source should be allowed to solve a given problem. In this way, the scheduler solves the conflict produced by the knowledge sources trying to use, all at the same time, the available system resources.

When the condition of a given knowledge source is satisfied, the knowledge source notifies the scheduler by adding a new record to the agenda. This record is usually called

KSAR (Knowledge Source Activation Record), and can be seen as a specification of the knowledge resulting from the triggering process. It can contain information about the problem identified, an estimation of the time that may take its resolution, or any other facts that may be useful to the action part of the knowledge source or the scheduler.

The general blackboard system operation evolves around the following cycle:

- The scheduler chooses a KSAR from the agenda.
- The scheduler notifies the corresponding knowledge source and allows it to solve the problem represented in the KSAR.
- The knowledge source solves the problem and places the solution on the blackboard.
- The knowledge sources then test their conditions on the new state of the blackboard. Any newly triggered one place a new KSAR on the agenda.

We can see that the scheduler is the director of the execution of the system, and embodies the problem-solving strategy of the system. This strategy can be tuned to a specific problem domain or can be made to be flexible and dynamic. It is this characteristic what makes the blackboard framework so powerful and useful. Any type of reasoning step (data driven, goal driven, model driven, forward chaining, backward chaining and so on) can be applied at each cycle of execution.

As early as with Hearsay-II ([LE77]), it was felt the need to provide such facilities. Hearsay-II provided a control component of the form of a special knowledge source called scheduler that could be configured to adopt different strategies. A variation of this approach was introduced in the HASP system ([NFAR82]) where, instead of one scheduler, it had three specialized schedulers. These early systems considered the control knowledge to be concentrated only in the scheduler or schedulers. This knowledge, thus, was implicit in the implementation and programming of the schedulers. Stanford's BB1 ([HRH85]), instead, followed a different approach; it allows for the explicit expression of control knowledge. BB1 keeps all control information in a global control plan stored in the blackboard. This allows the development of special knowledge sources (control KS's) that embody strategies and direct heuristics and foci of attention. These knowledge sources can modify the control plan, adapting it to new situations, as the reasoning process proceeds. The BB1 scheduler task then consists on evaluating the KSARs in the agenda against the current control plan, and choosing the one with the highest priority. [GHR89] compares this two control approaches from an empirical point of view. The authors conclude that while the implicit approach is

more efficient, the second is much more flexible for the representation and explanation of control knowledge, as well as for the ease of application development.

A.4 Parallelism on the blackboard model

One of the appeals of the blackboard model is that it has several characteristics that make it a good candidate to be considered as inherently parallel model. The most important characteristic is that the model relies upon semantically distributed conceptions of knowledge. It is based on the interaction of a group of completely independent entities (the knowledge sources) using a very well defined communication media (the blackboard). This simplicity makes it a very clean and seemly useful model for parallel problem solving.

While this is true for the original blackboard model, a number of problems appear when we include the control component. The way this component integrates with the original blackboard model in a parallel environment, is a problem that has not been properly solved. The amount and strength of control knowledge that is optimal for solution formation, is a serious and difficult question. Too much control serializes the computation excessively, therefore loosing any advantages offered by parallel processing. Too little control leads to very inefficient and ineffective problem solving.

The basic blackboard model presented in the previous sections, is an example of "too much control". The scheduler in the model has full control on the problem solving process. At each step, the scheduler allows one knowledge source to solve a problem, completely serializing the system's operation. The main function of this scheduler is to solve the conflict that arises when several knowledge sources try to use the same processing resources at the same time. This is acceptable in sequential systems such as Hearsay-II, BB1, Blondie-I ([LVV86]), etc. In parallel systems, however, this type of scheduler forms a bottleneck and has to be adapted properly.

We can identify two basic approaches into solving the integration of control and the original blackboard model. The first approach, which we will call *distributed* approach, basically avoids the problem completely. It is based on the idea of the spatial decomposition of the problem to be solved. This is; the problem is amenable to be decomposed in very similar problems on restricted areas of the problem domain. For example, the Distributed Vehicle Monitoring Testbed ([LC83]) keeps track of different vehicles within an area of a city. The system solves the problem by dividing the global problem into several smaller problems consisting of keeping track of vehicles on a smaller area (one block). The DVMT system works by having a network of blackboard

systems solving the smaller problems and communicating with each other whenever vehicles cross area boundaries.

The global architecture of the distributed system looks as a number of nodes interconnected. Each one of these nodes containing sequential blackboard systems that work on very similar problems. Other systems developed with this structure are, the distributed version of Hearsay-II ([LE80]), Blondie-III ([FvLV88]) and the Activation Framework ([Gre87]). This approach is specially useful when the parallel processing available is provided by a network of processors on a slow network such as Ethernet or Token-Ring.

The second approach is what we call the *parallel* approach. It is based on the idea that the different blackboard system components can be evaluated in parallel. Systems using this approach have to confront the scheduler problem. The two extremes in the solution of "how much control" should have the scheduler, are evaluated in the CAGE and POLIGON systems ([NAR89]). CAGE takes the Hearsay-II idea of a centralised scheduler that manages the resources of a central processor, and applies it to several processors. The CAGE system has a control cycle that is very similar to that of Hearsay-II, its only difference being that it allows several knowledge sources to execute simultaneously, instead of just one. The CAGE scheduler then waits until all knowledge sources have finished working in order to generate the new agenda and evaluate another set of knowledge sources. The whole system synchronizes on the execution of the scheduler. The POLIGON system, instead, goes to the other extreme; it is an asynchronous system that has no scheduler at all. Whenever a knowledge source trigger, it is executed. The CAGE system did not produce good results ([RAN89]), basically because the central scheduler was a bottleneck, no knowledge source could work while the scheduler worked. Also, fast knowledge sources had to wait for slow ones. The POLIGON system produced better results, loosing, however, the capabilities of expressing control knowledge. Other approach is the *Virtual Framework* ([Jag89]) which is a mixture between the DVMT-distributed and the CAGE-parallel approaches. It works on a unique global blackboard that is shared by several processors. Each processor has its own set of knowledge sources and scheduler that operate, mostly, on its own local area of the blackboard and can access other blackboard areas transparently. Other system following this approach is a later version of DVMT ([DGHL90]). In this thesis, we will present a form of solving the scheduler problem in a way that allows the execution of the system in way similar to that of POLIGON, yet keeping the ability to control the operation of the knowledge sources.

Several other systems have applied the blackboard architecture to distributed environments. [EG86] proposes a distributed blackboard architecture where access to the blackboard is by a mechanism similar to the 'data locks' used for distributed databases ([Ull82]). One feature of the [EG86] system is that knowledge sources not only communicate through the blackboard, but can send direct messages between one another as well. [aT86] describes an environment for distributed problem solving (COPS) based on the blackboard architecture. This system is a group of interconnected nodes, where each node is a production system with its own local memory, production memory and inference engine. There is one special node whose local memory holds the blackboard. The rest of the nodes act as knowledge sources. Other distributed blackboard systems are TRICERO ([Nii86b]), [Elf86], [Dav87] and [CT86].

These systems have become part of what is now called *distributed artificial intelligence (DAI)* or *distributed problem solving* ([Huh87]). These titles are used to label all those systems that use AI techniques within a distributed environment. DAI systems attempt to address the same questions as distributed blackboard systems, but with a different approach. For a good overview of DAI, see [Her88]. Within DAI, some blackboard systems are seen as *shared memory* systems, in contrast with *message passing* systems such as those based on the Actor model ([Hew77]) or on Contract Net ([DS83]). Message passing systems, attempt problem solving as individual processes that interact by sending messages to one another. In the case of the actor model each process is very simple and the system computations are mainly done through communications. In the Contract Net model each process is independent and semi-intelligent; that is, it can operate autonomously. They co-operate through negotiation to perform a given task. A process in the contract net can decide to broadcast the availability of a given task; some other process (or processes) can then bid for it, and the original process will evaluate all bids and award the task to the winner.

Appendix B

A Glossary of the Blackboard Performance Framework Terms

B.1 Basic Terms

Blackboard: The blackboard is the global structure that stores information about hypothesis, global data, partial results, goals and control information.

Level: A level is a partition of the blackboard that holds particular type of information. Levels can also be subdivided in sublevels, each one of them holding a different type of information.

BBel: A bbel or blackboard element, is a complex structure holding a particular instance of a given type of information. This type of information is given by the level in which it is stored.

Field: A field represents a particular characteristic that composes a particular piece of information (bbel). A given bbel is formed by a number of fields, one per property.

Blackboard Depth: The depth of a blackboard system is the number of nested levels it can have.

Blackboard Modification Operations:
Blackboard modification operations change the state of the blackboard. These operations include the creation of bbels, the modification of the information within a bbel or set of bbels and the deletion of bbels from the blackboard.

Blackboard Access Operations: The blackboard access operations are the ones that only read from the blackboard and do not modify it. This includes the retrieval of information stored within BBels and testing for the existence of particular pieces of information.

Knowledge Source: A knowledge source is the embodiment of the knowledge on a particular domain. The use of this knowledge will allow the system to solve particular problems.

Trigger Condition: A trigger condition is the part of the knowledge source that identifies its problem domain. It holds the specification of the problem or problems that the knowledge source knows how to solve.

Knowledge Source Body: The body is the part of the knowledge source that knows how to solve its problem domain. It holds the knowledge needed for achieving a solution of a problem identified by the knowledge source's trigger condition.

KSAR: A KSAR is the representation of a potential knowledge source execution produced by a knowledge source trigger condition. The KSAR can hold information about the the problem that the knowledge source will solve, as well as its its intentions, estimated speed or certainty of arriving to a solution.

Trigger Granularity: The granularity of the KS trigger conditions of a particular system is given by the smallest blackboard structure that they can specify when describing a problem.

The KS trigger granularity can be of the following types:

Blackboard The trigger conditions have blackboard granularity when they specify that they are to be called only when there is a modification to the blackboard (any modification).

BB Event The KS trigger condition is evaluated when a particular blackboard event occurs. The blackboard events are usually *creation*, *modification* and *deletion* of BBels.

Level The trigger condition is evaluated when there is a modification to a particular level of the blackboard.

BBel The trigger condition can restrict its evaluation to modifications performed to a particular BBel.

Field The trigger condition is only evaluated when there is a modification to a particular field of BBels on a particular level.

Scheduler: The scheduler is the part or parts of a blackboard system that decides on the order of execution of the knowledge sources. It is the embodiment of the meta-knowledge not held already in the knowledge sources.

Implicit Control: A blackboard system has implicit control knowledge when this knowledge is stored within a scheduler or group of specialised schedulers. This knowledge is static in the sense that cannot be modified once the system is running.

Explicit Control: A blackboard system has explicit control knowledge when this knowledge is stored in the blackboard. This offers the possibility of changing the current system control knowledge while the problem solving activity is in progress.

Control Structure: Complex Scheduler This is the control structure of those systems in which all the control knowledge is coded in one big scheduler.

Collection of Schedulers This is the usual configuration of implicit control systems. The control knowledge is separated in several control modules each dedicated to take particular control decisions.

Control Plan This is the usual configuration for explicit control systems. The system has a scheduler that decides the execution of the knowledge sources on the basis of control knowledge stored in the blackboard. The set of all public control knowledge is called the *Control Plan*. This control plan is created and updated by a set of control knowledge sources.

Basic Blackboard Modules:

Blackboard Manipulation Module. This module embodies all the procedures that create, modify, retrieve and delete blackboard elements, together with all its support procedures.

Knowledge Source Activation Module. This module holds all the knowledge source trigger conditions, as well as all the procedures needed for evaluating these conditions. It also contains procedures that create agenda elements and the ones that update the agenda.

Scheduler Module. It is formed by the set of procedures that decide which knowledge source body will be evaluated next. This includes all the procedures that perform an ordering of the agenda.

Knowledge Source Execution Module. This module holds all the different knowledge source bodies and all the procedures needed for evaluating them.

Service Modules: These are a number of secondary system modules that provide particular services to the four basic modules. Examples of service modules could be a user interface module, a module for communicating to other systems, a module that provides an interface to a database, etc.

Control Cycle: It is the sequence, or cycle of modules that a blackboard system executes in the process of problem solving.

Type of Architecture:

A blackboard architecture is *sequential* when the interdependencies between the modules make it impossible to evaluate more than one module at a time.

A *parallel* blackboard architecture is the one in which the evaluation of a particular module can be started or continued without waiting for the previous module to finish.

A *distributed* blackboard architecture is the one composed by several clusters of the basic modules having either sequential or parallel sub-architectures that are usually linked by service modules.

B.2 Blackboard System Performance

Building Blocks: All blackboard systems will have a number n of modules: M_1, ..., M_n. Module M_i, will have a number m_i of functions: $f^1_{M_i}$, ..., $f^{m_i}_{M_i}$. The number of calls to function $f^j_{M_i}$ will be called $Nf^j_{M_i}$, and its total execution time will be $Tf^j_{M_i}$

B.2.1 Module Use

N_{M_i} : **Quantitative Module Usage**

G_{M_i}: **Global Module Proportion**

B.2.2 Module Performance

$Af^j_{M_i}$: **Average Module Function Time**

$Pf^j_{M_i}$: **Percentage of Use**

WM_i: **Weighted Module Performance**

PM_i: **Proportional Module Performance**

TM_i: **Total Module Execution Time**

Blackboard Manipulation Module

AC_M: **Average BBel Creation Time**

AR_M: **Average Blackboard Read Time**

AM_M: **Average Blackboard Modification Time**

AD_M: **Average BBel Deletion Time**

N_M: **Percentage of Use**

PC_M, PR_M, PM_M and PD_M: Bbel creation, blackboard read, blackboard modification and deletion percentage of use.

AM: **Proportional Blackboard Manipulation Module Time**

TM: **Total Blackboard Manipulation Module Time**

Knowledge Source Activation Module

TT_T: **Total KS Trigger Time**

AT_T: **Average KS Trigger Time**

TT: **Total KS Activation Module Time**

AT: **Average KS Activation Module Time**

Scheduler Module

TS_S: **Total Scheduler Time**

AS_S: **Average Scheduler Time**

TS: **Total Scheduler Module Time**

AS: **Average Scheduler Module Time**

Knowledge Source Execution Module

TB_B: **Total KS Execution Time**

AB_B: **Average KS Execution Time**

TB: **Total KS Execution Module Time**

AB: **Average KS Execution Module Time**

B.2.3 Global Module Influence

T_{Run}: **Total System Run Time**

T_{Seq}: **Sequential Run Time**

TUa: **Total Unaccounted Time**

P_{M_i}: **Module Proportion**

P_{U_a}: **Proportion of Unaccounted Time**

B.2.4 System Performance

A_{Cy}: **Average Cycle Time (A_{Cy})**

SU: **System Speed Up**

Appendix C

Jigsaw-Puzzle Solver, Sample Implementation: EPBS

```
Dec 30 20:10 1991  README Page 1

How to use JSP:

If there is a KS database already generated:

Start prolog with the prolog bbshell. For example:
%       prolog ../blackboard

Then load the different files and run the system:
?-      [load].
?-      run.                          (or step by step, if you prefer)

The system then will ask two questions. For the first question,
depending on the piece KSes file loaded (see the 'load' file),
may be different. For the second question the answer should
always be 0 unless you want to generate a new puzzle.

Please, enter the name of the file with the JSP data: 'data/jsp-1x4'.
Please, enter the number of KSes to generate (Max:M): 0.

Then the system will begin to describe where it's placing each
piece.

If you want to generate a new KS database:

Start prolog as before:

%       prolog ../blackboard

Then load the different files and run the system:
?-      [load_gen].
and
?-      run.                          (or step by step, if you prefer)

You should have a file prepared with the data corresponding to the
new jigsaw you want to generate (you can see its format in the
ones given as example)

As before, the system will ask two questions. For the first question,
you will answer the name of the file where the data is.
The second question will ask for the number of KSes that
you wish to be generated. It should be a number between 1 and the
maximum expressed in the question. For example:

Please, enter the name of the file with the JSP data: 'data/jsp-3x4'
Please, enter the number of KSes to generate (Max:12): 12.

Once they are generated, you would store those KSes in a file, so
it can be used later. You do this by executing the following predicate
?-      saveKSes('<File Name>').

Now, you can modify the 'load' file and follow the steps as given before

Emilio.
eam, 13/12/89
(modified 22/1/90)
```

```
Dec 30 20:10 1991  jsp-1x4-4.ks Page 1

%
%       FILE:           JSP.EPBS/jsp-1.4-4.ks
%       CREATED BY      Emilio Agustin Molina (eam)
%

%-----PieceKS4
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS3
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS2
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS1
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]
```

```
Dec 30 20:10 1991  jsp-1x4-2.ks Page 1

%
%       FILE:           jsp-1x4-2
%       CREATED BY      Emilio Agustin Molina (eam)
%

%-----PieceKS2
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS1
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]
```

```
Dec 30 20:10 1991  jsp-1x6-6.ks Page 1

%
%       FILE:           JSP.EPBS/jsp-1 6-6.ks
%       CREATED BY      Emilio Agustin Molina (eam)
%

%-----PieceKS6
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS5
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS4
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS3
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS2
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS1
if      # ([jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
```

```
Dec 30 20:10 1991  jsp-1x6-6.ks Page 2

%
set     [pieceKS,jigsaw/position(_294957,_294958)].
```

```
Dec 30 20:10 1991  jsp-2x4-8.ks Page 2

%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS2
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS1
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]
```

```
Dec 30 20:10 1991  jsp-2x4-8.ks Page 1

%
%       FILE:           JSP EPBS/jsp-2x4-8.ks
%       CREATED BY      Emilio Agustin Molina (eam)
%

%-----PieceKS8
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS7
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS6
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS5
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS4
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS3
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
```

```
Dec 30 20:10 1991  jsp-3x4-12.ks Page 1

%
%       FILE:           jsp-3x4-12.ks
%       CREATED BY      Emilio Agustin Molina (eam)
%

%-----PieceKS12
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS11
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)].

%-----PieceKS10
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS9
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS8
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
%
set     [pieceKS,jigsaw/position(_294957,_294958)]

%-----PieceKS7
if      @ []jigsaw/position(_294957,_294958),[piece(active),_294966,links([_2949
%
then    jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to      amend _295061
```

```
Dec 30 20:10 1991  jsp-3x4-12.ks Page 2

%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS6
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS5
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS4
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS3
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS2
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].

%-----PieceKS1
if       # [jigsaw/position(_294957,_294958),[piece(active),_294966,links([_29497
%
then     jsp_update(jigsaw/position(_294957,_294958),_294997,_294998,_295061)
%
to       amend _295061
%
set      [pieceKS, jigsaw/position(_294957,_294958)].
```

```
Dec 30 20:10 1991  jsp-Control.pl Page 1

%       FILE:           jsp-Domain.pl
%       DATE CREATED:   Mon Sep 11 14:48:25 BST 1989
%       AUTHOR:         Emilio Agust'in Molina (eam)
%

%       Function used to determine the KSAR to be executed next.
%       It defines an ordering relation for KSARs the smallest
%       of wich will be executed next. It means LessThanOrEqual.
%
less_rating(Est1,Est2) :-
        number(Est1), number(Est1), !, Est1 =< Est2.
less_rating([T1,Pos1],[T2,Pos2]) :-
        get_rating(T1,Pos1,R1),
        get_rating(T2,Pos2,R2), !, R1 =< R2.

%       Get the rating for the KSAR
%
rating([Type,Pos],[Type,Pos]) :- !.
rating(Est,Rating) :-
        Rating is Est.

%       Get the rating for the KSAR with type piece/misplaced/init
%       for the current cycle.
%
get_rating(initKS,_,1) :- !.
get_rating(pieceKS,Position,Rating) :-
        recorded(current_cycle,CurrentCycle,_),
        ((recorded(jspRating,[pieceKS,Position,Cycle,R],Ref),
          (CurrentCycle = Cycle ->
                Rating = R
                ;(erase(Ref), fail)))
        ;(entry(Position,[_,values(Vals),links(Links),_,_],_),
          ratePosition(Vals,Links,R),
          Rating is 100 - R,
          record(jspRating,[pieceKS,Position,CurrentCycle, Rating],_))),
          !.
get_rating(misplacedKS,Position,Rating) :-
        recorded(current_cycle,CurrentCycle,_),
        ((recorded(jspRating,[misplacedKS,Position,Cycle,R],Ref),
          (CurrentCycle = Cycle ->
                Rating = R
                ;(erase(Ref), fail)))
        ;((entry(agenda,ksar(body(jsp_update(Position,_,_,_),_,_,_),_,_),_),_),
           Rating = 100)
         ; Rating = 1)),
        !.

%       Rate the actual KS producing an 'Est'imate value.
%       This value depends on the position that triggered
%       the KS, given by its Values and Links, and its Type (that is
%       if it is a 'piece' KS or a 'misplaced' KS).
%
rateKS(_, _, misplaced, 1) :- !.            % Misplaced KS.
rateKS(Values, Links, piece, Est) :-        % Piece KS.
        ratePosition(Values,Links,Rating),
        Est is 100 - Rating, !.             % This is because the lowest the
```

```
Dec 30 20:10 1991  jsp-3x4-12.ks Page 3
```

```
Dec 30 20:10 1991  jsp-Control.pl Page 2

                                            % estimate, the highest the priority.
rateKS(_, _, init, 1) :- !.                 % Initialization KS.
rateKS(_, _, _, 100) :- !.                  % Default.
```

```
Dec 30 20:10 1991  jsp-Domain.pl Page 1

%       FILE:           jsp-Domain.pl
%       DATE CREATED:   Mon Sep 11 14:48:25 BST 1989
%       AUTHOR:         Emilio Agust'in Molina (eam)
%

%       Check if a sequence [T,R,B,L] of values belongs to a
%       corner position (or piece).
%
is_corner([T,R,_,_]) :-
        nonvar(T), nonvar(R),
        T = 0, R = 0.                   % Top-right corner
is_corner([T,_,_,L]) :-
        nonvar(T), nonvar(L),
        T = 0, L = 0.                   % Top-left corner
is_corner([_,R,B,_]) :-
        nonvar(R), nonvar(B),
        R = 0, B = 0.                   % Bottom-right corner
is_corner([_,_,B,L]) :-
        nonvar(B), nonvar(L),
        B = 0, L = 0.                   % Bottom-left corner

%       Check if a sequence [T,R,B,L] of values belongs to a
%       position (or piece) in the sides of the puzzle. The
%       corners are included.
%
is_side([T,_,_,_]) :-
        nonvar(T), T = 0.               % Top side
is_side([_,R,_,_]) :-
        nonvar(R), R = 0.               % Right side
is_side([_,_,B,_]) :-
        nonvar(B), B = 0.               % Bottom side
is_side([_,_,_,L]) :-
        nonvar(L), L = 0.               % Left side

%       Check if a piece in PieceList matches the given position.
%
jsp_matches_set(Position,PieceList,Piece,Or) :-
        entry(Position,Record,_), !,
        jsp_matches_set2(Record,PieceList,Piece,Or)

%
jsp_matches_set2(PosRec,[Piece | _], Piece, Or) :-
        jsp_matches(PosRec,Piece,Or), !.
jsp_matches_set2(Position,[_ | PieceList],Piece,Or) :-
        jsp_matches_set2(Position,PieceList,Piece,Or).

%       Check if a piece matches a given position.
%       The position shoud be given by its Record.
%       If it succeeds, it will return the orientation.
%       Note!!: This predicate instantiates variables.
%
jsp_matches([_, values([Tv,Rv,Bv,Lv]),
            _,old_pieces(OldL),_],Piece,Or) :-
        entry(Piece,[not_placed(t),_,
                values([Tval,Rval,Bval,Lval])|_],_),
%
```

```
Dec 30 20:10 1991  jsp-Domain.pl Page 2

        ((jsp_match(Tv,Tval),     % Orientation: t
          jsp_match(Rv,Rval),     %    tt
          jsp_match(Bv,Bval),     % ll   rr
          jsp_match(Lv,Lval),     %    bb
          Or = t)
%       ;(jsp_match(Tv,Rval),     % Orientation: r
%         jsp_match(Rv,Bval),     %    tr
%         jsp_match(Bv,Lval),     % lt   rb
%         jsp_match(Lv,Tval),     %    bl
%         Or = r)
%       ;(jsp_match(Tv,Bval),     % Orientation: r
%         jsp_match(Rv,Lval),     %    tb
%         jsp_match(Bv,Tval),     % lr   rl
%         jsp_match(Lv,Rval),     %    bt
%         Or = b)
%       ;(jsp_match(Tv,Lval),     % Orientation: r
%         jsp_match(Rv,Tval),     %    tl
%         jsp_match(Bv,Rval),     % lb   rt
%         jsp_match(Lv,Bval),     %    br
%         Or = l)
        ),
        not_inList([Piece,Or],OldL).

%       Match a position value with a piece value.
%       If PosVal is a variable and PieVal is not 0 then
%       both will be unified.
%
jsp_match(PosVal,PieVal) :-
        (var(PosVal) ->
                (PieVal \== 0,
                 PosVal = PieVal)
        ;       PosVal == PieVal).

%       Test if an Element is not in a List
%
not_inList(Element, [Element | _]) :- !, fail.
not_inList(Element, [_ | List]) :-
        !, not_inList(Element,List).
not_inList(_,[]).

%       Rate the actual Position, given its values and links
%       to other positions. Api is the number of adjacent
%       positions filled with a piece. Apo is the number of
%       active positions.
%
ratePosition(Values,Links,Rating) :-
        adjPiePos(Links,Api,Apo),
        (is_side(Values) ->
                Co = 5
        ;       Co = 0),
        Rating is 10 * Api + Apo + Co, !.

%       Find out the number of adjacent positions already filled
%       (with a piece, Api) and the number of adjacent active
%       positions (Apo), given the links of the current position.
%
```

```
Dec 30 20:10 1991  jsp-Domain.pl Page 3

adjPiePos([Tpos,Rpos,Bpos,Lpos], Api, Apo) :-
        entry(Tpos,[piece(Teta)|_],_),
        entry(Rpos,[piece(Reta)|_],_),
        entry(Bpos,[piece(Beta)|_],_),
        entry(Lpos,[piece(Leta)|_],_),
        (Teta = in/active ->
                (I1 = 0, O1 = 0)
        ;       (Teta = active ->
                        (I1 = 0, O1 = 1)
                ;       (I1 = 1, O1 = 0))),
        (Reta = in/active ->
                (I2 = 0, O2 = 0)
        ;       (Reta = active ->
                        (I2 = 0, O2 = 1)
                ;       (I2 = 1, O2 = 0))),
        (Beta = in/active ->
                (I3 = 0, O3 = 0)
        ;       (Beta = active ->
                        (I3 = 0, O3 = 1)
                ;       (I3 = 1, O3 = 0))),
        (Leta = in/active ->
                (I4 = 0, O4 = 0)
        ;       (Leta = active ->
                        (I4 = 0, O4 = 1)
                ;       (I4 = 1, O4 = 0))),
        Api is I1 + I2 + I3 + I4,
        Apo is O1 + O2 + O3 + O4, !

%       See if any of the positions in the link list, different to the
%       one provided, is filled with a piece.
%
supp_only_by(links(List),Pos) :-
        jsp_check_sup(List,Pos).

jsp_check_sup([Pos| Rest],Pos) :- !,
        jsp_check_sup(Rest,Pos), !.
jsp_check_sup([jigsaw/position(0,0)| Rest],Pos) :- !,
        jsp_check_sup(Rest,Pos), !.
jsp_check_sup([P|Rest],Pos) :-
        not entry(P,[piece(jigsaw/_)|_],_), !,
        jsp_check_sup(Rest,Pos),
        !.
jsp_check_sup([],_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
%                          BODIES OF KSEs                             %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Update a position and surounding positions if needed.
%       This is the body of all Piece KSes.
%
jsp_misplaced(Position,Adds) :-
        entry(Position,[piece(Piece),
                        values([Tv,Rv,Bv,Lv]),
```

```
Dec 30 20:10 1991  jsp-Domain.pl Page 4

                        links([Tp,Rp,Bp,Lp]),
                        old_pieces(OldP),LastM],_),
        entry(Piece,[_,Or,Vals],_),
        A1 = [Piece,[not_placed(t),Or,Vals],1],
        entry(Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),Tlinks,TOld,TLst],_),
        entry(Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),Rlinks,ROld,RLst],_),
        entry(Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),Blinks,BOld,BLst],_),
        entry(Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),Llinks,LOld,LLst],_),
        (Pt = active ->
                (supp_only_by(Tlinks,Position) ->
                        A2 = ([Tp,[piece(in/active),
                                values([TTv,TRv,_,TLv]),
                                Tlinks,
                                old_pieces([]),
                                last_modified_by(noposition)],1]
                            and A1)
                ;       A2 = ([Tp,[piece(active),
                                values([TTv,TRv,_,TLv]),
                                Tlinks, TOld, TLst],1]
                            and A1))
        ;       (Tval = Tv,
                 (Tv = 0 ->
                        (Add0 = [Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),
                                 Tlinks,TOld,TLst],1],
                         A2 = A1)
                        A2 = ([Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),
                                Tlinks,TOld,TLst],1] and A1)))),
        (Pr = active ->
                (supp_only_by(Rlinks,Position) ->
                        A3 = ([Rp,[piece(in/active),
                                values([RTv,RRv,RBv,_]),
                                Rlinks,
                                old_pieces([]),
                                last_modified_by(noposition)],1]
                            and A2)
                ;       A3 = ([Rp,[piece(active),
                                values([RTv,RRv,RBv,_]),
                                Rlinks, ROld, RLst],1]
                            and A2))
        ;       (Rval = Rv,
                 (Rv = 0 ->
                        (Add0 = [Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),
                                 Rlinks,ROld,RLst],1],
                        A3 = A2)
                        A3 = ([Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),
                                Rlinks,ROld,RLst],1] and A2)))),
        (Pb = active ->
                (supp_only_by(Blinks,Position) ->
                        A4 = ([Bp,[piece(in/active),
                                values([_,BRv,BBv,BLv]),
                                Blinks,
                                old_pieces([]),
                                last_modified_by(noposition)],1]
                            and A3)
                ;       A4 = ([Bp,[piece(active),
                                values([_,BRv,BBv,BLv]),
```

```
Dec 10 20:10 1991  jsp-Domain.pl Page 5

                                Blinks, BOld, BLst],1]
                          and A3))
        ;       (Bval = Bv,
                 (Bv = 0 ->
                        (Add0 = [Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),
                                 Blinks,BOld,BLst],1],
                         A4 = A3)
                 ;      A4 = ([Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),
                                 Blinks,BOld,BLst],1] and A3)))),
        (Pl = active ->
                (supp_only_by(Llinks,Position) ->
                        A5 = ([Lp,[piece(in/active),
                                values([LTv,_,LBv,LLv]),
                                Llinks,
                                old_pieces([]),
                                last_modified_by(noposition)],1]
                         and A4)
                ;       A5 = ([Lp,[piece(active),
                                values([LTv,_,LBv,LLv]),
                                Llinks, LOld, LLst],1]
                          and A4))
        ;       (Lval = Lv,
                 (Lv = 0 ->
                        (Add0 = [Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),
                                 Llinks,LOld,LLst],1],
                         A5 = A4)
                 ;      A5 = ([Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),
                                 Llinks,LOld,LLst],1] and A4)))),
        A6 = ([Position,
                [piece(active),
                 values([Tval,Rval,Bval,Lval]),
                 links([Tp,Rp,Bp,Lp]),
                 old_pieces(OldP),LastM],1]
             and A5),
        (var(Add0) ->
                Adds = A6
        ;       Adds = (Add0 and A6)),
        write('Removing Piece '),write(Piece),
        write(' in Position '),write(Position),nl,
        !.

%       Update a position and surounding positions if needed.
%       This is the body of all Piece KSes.
%
jsp_update(Position,Piece,Or,Adds) :-
        entry(Position,Record,_),
        entry(Piece,[_,_,Values],_),
        jsp_matches(Record,Piece,Or),
        Record = [_,values([Tv,Rv,Bv,Lv]),
                  links([Tp,Rp,Bp,Lp]),
                  old_pieces(OldP),LastM],
        entry(Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),Tlinks,TOld,TLst],_),
        entry(Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),Rlinks,ROld,RLst],_),
        entry(Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),Blinks,BOld,BLst],_),
        entry(Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),Llinks,LOld,LLst],_),
        A1 = ([Position,
```

```
Dec 10 20:10 1991  jsp-Domain.pl Page 6

                [piece(Piece),
                 values([Tv,Rv,Bv,Lv]),
                 links([Tp,Rp,Bp,Lp]),
                 old_pieces([[Piece,Or]|OldP]),
                 LastM],
                 1]
            and [Piece,[not_placed(f),orientation(Or),Values],1]),
        (Tv == 0 ->
                (Add0 = [Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),
                         Tlinks,TOld,TLst],1],
                 A2 = A1)
        ;  ((Pt = in/active ; Pt = active) ->
                A2 = ([Tp,[piece(active),values([TTv,TRv,Tv,TLv]),
                        Tlinks,TOld,last_modified_by(Position)],1] and A1)
        ;       A2 = ([Tp,[piece(Pt),values([TTv,TRv,TBv,TLv]),
                        Tlinks,TOld,TLst],1] and A1))),
        (Rv == 0 ->
                (Add0 = [Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),
                         Rlinks,ROld,RLst],1],
                 A3 = A2)
        ;  ((Pr = in/active ; Pr = active) ->
                A3 = ([Rp,[piece(active),values([RTv,RRv,RBv,Rv]),
                        Rlinks,ROld,last_modified_by(Position)],1] and A2)
        ;       A3 = ([Rp,[piece(Pr),values([RTv,RRv,RBv,RLv]),
                        Rlinks,ROld,RLst],1] and A2))),
        (Bv == 0 ->
                (Add0 = [Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),
                         Blinks,BOld,BLst],1],
                 A4 = A3)
        ;  ((Pb = in/active ; Pb = active) ->
                A4 = ([Bp,[piece(active),values([Bv,BRv,BBv,BLv]),
                        Blinks,BOld,last_modified_by(Position)],1] and A3)
        ;       A4 = ([Bp,[piece(Pb),values([BTv,BRv,BBv,BLv]),
                        Blinks,BOld,BLst],1] and A3))),
        (Lv == 0 ->
                (Add0 = [Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),
                         Llinks,LOld,LLst],1],
                 A5 = A4)
        ;  ((Pl = in/active ; Pl = active) ->
                A5 = ([Lp,[piece(active),values([LTv,Lv,LBv,LLv]),
                        Llinks,LOld,last_modified_by(Position)],1] and A4)
        ;       A5 = ([Lp,[piece(Pl),values([LTv,LRv,LBv,LLv]),
                        Llinks,LOld,LLst],1] and A4))),
        (var(Add0) ->
                Adds = A5
        ;       Adds = (Add0 and A5)),
        write(Piece),write(' placed in '),write(Position),nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                     %
%                       DISPLAY PREDICATES                            %
%                                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Display the contents of the position part of the BB.
%
```

```
Dec 10 20:10 1991  jsp-Domain.pl Page 7

jsp_display_pos :-
        entry(jsp/info,[JspX,JspY|_],_),
        writef("\f"),                   % Clear the screen
        jsp_write_pos(1,JspX,JspY),
        Times is JspX * 10,
        writef("-%r\n",['-',Times]), !.

%       Display the contents of the position part of the BB.
%
jsp_display_pis :-
        entry(jsp/info,[JspX,JspY|_],_),
        writef("\f"),                   % Clear the screen
        jsp_write_pis(1,JspX,JspY),
        Times is JspX * 10,
        writef("-%r\n",['-',Times]), !.

jsp_write_pos(Y,JspX,JspY) :-
        Times is JspX * 10,
        writef("-%r\n",['-',Times]),
        get_data(1,Y,JspX,Tvals,Bvals,Names,LRvSts),
        write('|'),
        writefList('%9c|',Tvals),nl,
        writef("|%r\n",['         |',JspX]),
        write('|'),
        writefList('  %2r-%2l  |',Names),nl,
        write('|'),
        writefList('%2l%5c%2r|',LRvSts), nl,
        writef("|%r\n",['         |',JspX]),
        write('|'),
        writefList('%9c|',Bvals),nl,
        (Y = JspY ->
                true
        ;       (Y0 is Y + 1,
                 jsp_write_pos(Y0,JspX,JspY))),
        !.

jsp_write_pis(Y,JspX,JspY) :-
        Times is JspX * 10,
        writef("-%r\n",['-',Times]),
        get_data2(1,Y,JspX,Tvals,Bvals,Names,LRvSts),
        write('|'),
        writefList('%9c|',Tvals),nl,
        writef("|%r\n",['         |',JspX]),
        write('|'),
        writefList('  %2r-%2l  |',Names),nl,
        write('|'),
        writefList('%2l%2r-%2l%2r|',LRvSts), nl,
        writef("|%r\n",['         |',JspX]),
        write('|'),
        writefList('%9c|',Bvals),nl,
        (Y = JspY ->
                true
        ;       (Y0 is Y + 1,
                 jsp_write_pis(Y0,JspX,JspY))),
        !.
```

```
Dec 10 20:10 1991  jsp-Domain.pl Page 8

%       Get the position data for the line Y
%
get_data(X,Y,JspX,[Tvl|Tvals],[Bvl|Bvals],
                [[X,Y]|Names],[[Lvl,St1,Rvl]|LRvSts]) :-
        entry(jigsaw/position(X,Y),
                [piece(St),
                 values([Tv,Rv,Bv,Lv])|_],_),
        (St = in/active ->
                St1 = '---'
        ;       (St = jigsaw/piece(Xp,Yp) ->
                        (name(Xp,Xn),name(Yp,Yn),
                         append(Xn,[45|Yn],St0),
                         name(St1,St0))
                ;       St1 = '***')),
        (var(Tv) ->
                Tvl = 'X'
        ;       Tvl = Tv),
        (var(Rv) ->
                Rvl = 'X'
        ;       Rvl = Rv),
        (var(Bv) ->
                Bvl = 'X'
        ;       Bvl = Bv),
        (var(Lv) ->
                Lvl = 'X'
        ;       Lvl = Lv),
        (JspX =< X ->
                (Tvals = [],
                 Bvals = [],
                 Names = [],
                 LRvSts = [])
        ;       (X0 is X + 1,
                 get_data(X0,Y,JspX,Tvals,Bvals,Names,LRvSts))),
        !.

%       Get the piece data for the line Y
%
get_data2(X,Y,JspX,[Tvl|Tvals],[Bvl|Bvals],
                [[X,Y]|Names],[[Lvl,Np,Or,Rvl]|LRvSts]) :-
        entry(jigsaw/piece(X,Y),
                [not_placed(Np),
                 orientation(Or),
                 values([Tv,Rv,Bv,Lv])]|_),
        (var(Tv) ->
                Tvl = 'X'
        ;       Tvl = Tv),
        (var(Rv) ->
                Rvl = 'X'
        ;       Rvl = Rv),
        (var(Bv) ->
                Bvl = 'X'
        ;       Bvl = Bv),
        (var(Lv) ->
                Lvl = 'X'
        ;       Lvl = Lv),
        (JspX =< X ->
```

```
Dec 10 20:10 1991   jsp-Domain.pl Page 9

                (Tvals = [],
                 Bvals = [],
                 Names = [],
                 LRvSts = [])
        ;       (X0 is X + 1,
                 get_data2(X0,Y,JspX,Tvals,Bvals,Names,LRvSts))),
        !.

%       Writef each element of L with format F.
%
writefList(F,[H|T]) :-
        writef(F,H), !,
        writefList(F,T).
writefList(_,_).

%       Example:
%
%
%       ------------------------------
%           0    |     0    |     0    |
%                |          |          |
%          1-1   |    2-1   |    3-1   |
%       0  1-1  3|3   ***  X|X   ---  X| ...
%                |          |          |
%           3    |     X    |     X    |
%       ------------------------------
%           3    |
%                |
%          2-1   |
%       0  ***  X|
%                |
%           3    |
%       ----------
%
%           :

%       This is needed, so writef can center things in fields.
%
is(X,div,Y,Z) :-
        X is Y div Z, !.
```

```
Dec 10 20:10 1991   jsp-Gen-ks.pl Page 1

%       FILE:             jsp-gen-kb.pl
%       DATE CREATED:     Fri Sep 8 11:22:26 BST 1989
%       AUTHOR:           Emilio Agust'in Molina (eam)
%

%---- JSP-INIT: Ask for the name of the file with the Jigsaw-Puzzle data
%       The file given should have the format:
%       sizeX sizeY
%       Tval Rval Bval Lval            (for position(1,1))         \
%       Tval Rval Bval Lval            (for position(2,1))         |
%         :    :    :    :                                         | sizeX *
%       Tval Rval Bval Lval            (for position(I,J))         >  sizeY
%         :    :    :    :                                         | pieces
%       Tval Rval Bval Lval            (for position(sizeX,sizeY)) /
%
%
if      notnow [jigsaw/position(0,0),_,true]
%
then    (repeat,
         write('Please, enter the name of the file with the JSP data: '),
         read(File),
         (get_data(File,JspX,JspY,AddPieces) -> true
         ;       (write('Not in the proper format.'),nl,seen)))
%
to      add     [jigsaw/position(0,0),
                 [piece(jigsaw/position(0,0)),
                  values([0,0,0,0]),
                  links([]),
                  old_pieces([]),
                  last_modified_by(jigsaw/position(0,0))],
                 1]
        and     [jsp/info,[JspX,JspY],1]
        and     [generateKSes,0,1]
        and     AddPieces
%
set     [initKS,0].

%-----Generate-KSes: Generate NKSes.
%
if      #[generateKSes,_,true]
%
then    (entry(jsp/info,[JspX,JspY],_),
         Max is JspX * JspY,
         repeat,
         write('Please, Enter the number of KSes to generate (Max:'),
         write(Max), write(') '),
         read(NKSes),
         (number(NKSes) -> true
         ;       (write('Not a number.'),nl,fail)),
         ((0 =< NKSes, NKSes =< Max) -> true
         ;       (write('Not within range.'),nl,fail)),
         (NKSes \== 0 ->
                 generate_KSes(NKSes,Max)
         ;       true))
%
to      delete
```

```
Dec 10 20:10 1991   jsp-Gen-ks.pl Page 2

%
set     [initKS,0].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%                                  PREDICATES                                  %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Save generated KSes to a file
%
saveKSes(File) :-
        tell(File),
        write('%'),nl,
        write('%        FILE:           '),write(File),nl,
        write('%        CREATED BY      Emilio Agustin Molina (eam)'),nl,
        write('%'),nl,nl,
        writeKSes,
        told.

writeKSes :-
        pieceKS(N,(if Cond then Body to Action set Set)),
        write('%-----PieceKS '),write(N),nl,
        write('if       '),writeq(Cond),nl,write('%'),nl,
        write('then     '),writeq(Body),nl,write('%'),nl,
        write('to       '),writeq(Action),nl,write('%'),nl,
        write('set      '),writeq(Set),write(' '),nl,nl,
        fail.
writeKSes.

%       Generate the given number of KSes.
%
generate_KSes(NKSes,Max) :-
        PiPerKS is Max div NKSes,
        ExtraPi is Max mod NKSes,
        findall(Piece,(entry(jigsaw/piece(X,Y),_,_),
                       Piece = jigsaw/piece(X,Y)),PieceList),
        gen_KSes(NKSes,PiPerKS,ExtraPi,PieceList), !.

gen_KSes(NKSes,_,_,_) :- NKSes =< 0, !.
gen_KSes(NKSes,PiPerKS,0,PieceList) :-
        write(['GEN_KSes:', NKSes, PiPerKS, 0, PieceList]),nl,
        get_first(PiPerKS,PieceList,FirstN,RestL),
        templatePieceKS(KS,FirstN),
        assert(pieceKS(NKSes,KS)),
        N is NKSes - 1,
        gen_KSes(N,PiPerKS,0,RestL), !.
gen_KSes(NKSes,PiPerKS,ExtraPi,PieceList) :-
        write(['GEN_KSes:', NKSes, PiPerKS, ExtraPi, PieceList]),nl,
        NPieces is PiPerKS + 1,
        get_first(NPieces,PieceList,FirstN,RestL),
        templatePieceKS(KS,FirstN),
        assert(pieceKS(NKSes,KS)),
        N is NKSes - 1,
        Ex is ExtraPi - 1,
        gen_KSes(N,PiPerKS,Ex,RestL).
```

```
Dec 10 20:10 1991   jsp-Gen-ks.pl Page 3

%
%       Template for Piece KSes.
%
templatePieceKS((
        if      #[jigsaw/position(X,Y),
                  [piece(active),
                   _,
%                  values(Values),
                   links([Tpos,Rpos,Bpos,Lpos]) | _],
                  (PieceList = PIECES,
                   jsp_matches_set(jigsaw/position(X,Y), PieceList,Piece,Or))]
%                  jsp_matches_set(jigsaw/position(X,Y), PieceList,Piece,Or),
%                  rateKS(Values,[Tpos,Rpos,Bpos,Lpos],piece,Set))]
            and #[Tpos, _, true]
            and #[Rpos, _, true]
            and #[Bpos, _, true]
            and #[Lpos, _, true]
            and #[Piece,_, true]
        %
        then    jsp_update(jigsaw/position(X,Y),Piece,Or,Adds)
        %
        to      amend   Adds
        %
        set     [pieceKS, jigsaw/position(X,Y)]),
        PIECES).

%
%       Example Piece KS.
%
exPieceKS(
        if      #[jigsaw/position(X,Y),
                  [piece(active),
                   values(Values),
                   links([Tpos,Rpos,Bpos,Lpos]) | _],
                  (PieceList = [jigsaw/piece(1,1), jigsaw/piece(1,2)],
                   jsp_matches_set(jigsaw/position(X,Y), PieceList),
                   rateKS(Values,[Tpos,Rpos,Bpos,Lpos],piece,Set))]
            and #[Tpos, _, true]
            and #[Rpos, _, true]
            and #[Bpos, _, true]
            and #[Lpos, _, true]
        %
        then    jsp_update(jigsaw/position(X,Y),Adds)
        %
        to      amend   Adds
        %
        set     Set
).

%       Get the N first elements of the List, returning them in
%       the FirstN list and the rest in Rest
%
get_first(N,List,[],List) :- N =< 0, !.
get_first(N,[E|List],[E|FirstN],Rest) :-
        N1 is N - 1,
```

```
Dec 30 20:10 1991   jsp-Gen-ks.pl Page 4


        get_first(N1,List,FirstN,Rest), !.
get_first(_,[],[],[]).

%       Generate enouth adds to generate all potitions
%       and pieces as needed in the jigsaw puzzle stored
%       in the File.
%
get_data(File,JspX,JspY,Adds) :-
        see(File),
        read(JspX),
        read(JspY),
        get_pieces(JspX,JspY, 1, 1, Adds),
        seen, !.

%       Get each piece data and generate adds for it and
%       its position.
%
get_pieces(JspX, JspY,
           JspX, JspY,
           ([jigsaw/position(JspX,JspY),
             [piece(in/active),
              values([Tv,Rv,Bv,Lv]),
              links([Tpos,Rpos,Bpos,Lpos]),
              old_pieces([]),
              last_modified_by(noposition)],
             ]]
            and [jigsaw/piece(JspX,JspY),
                 [not_placed(t),
                  orientation(t),
                  values([Tval,Rval,Bval,Lval])],
                 ]])) :-
        read(Tval),
        (Tval = 0 -> Tv = 0 ; true),
        read(Rval),
        (Rval = 0 -> Rv = 0 ; true),
        read(Bval),
        (Bval = 0 -> Bv = 0 ; true),
        read(Lval),
        (Lval = 0 -> Lv = 0 ; true),
        X0 is JspX - 1,
        Y0 is JspY - 1,
        (Y0 = 0 ->
                Tpos = jigsaw/position(0,0)
        ;       Tpos = jigsaw/position(JspX,Y0)),
        Rpos = jigsaw/position(0,0),
        Bpos = jigsaw/position(0,0),
        (X0 = 0 ->
                Lpos = jigsaw/position(0,0)
        ;       Lpos = jigsaw/position(X0,JspY)), !.
get_pieces(JspX, JspY,
           X,    Y,
           ([jigsaw/position(X,Y),
             [piece(in/active),
              values([Tv,Rv,Bv,Lv]),
              links([Tpos,Rpos,Bpos,Lpos]),
              old_pieces([]),
```

```
Dec 30 20:10 1991   jsp-Gen-ks.pl Page 5


              last_modified_by(noposition)],
             ]]
            and [jigsaw/piece(X,Y),
                 [not_placed(t),
                  orientation(t),
                  values([Tval,Rval,Bval,Lval])],
                 ]]
            and Adds)) :-
        read(Tval),
        (Tval = 0 -> Tv = 0 ; true),
        read(Rval),
        (Rval = 0 -> Rv = 0 ; true),
        read(Bval),
        (Bval = 0 -> Bv = 0 ; true),
        read(Lval),
        (Lval = 0 -> Lv = 0 ; true),
        X0 is X - 1,
        (X = JspX ->
                (X1 = 0, Xnext = 1, Ynext is Y + 1)
        ;       (X1 is X + 1, Xnext = X1, Ynext = Y)),
        Y0 is Y - 1,
        (Y = JspY -> (Y1 = 0)
        ;       (Y1 is Y + 1)),
        (X0 = 0 ->
                Lpos = jigsaw/position(0,0)
        ;       Lpos = jigsaw/position(X0,Y)),
        (X1 = 0 ->
                Rpos = jigsaw/position(0,0)
        ;       Rpos = jigsaw/position(X1,Y)),
        (Y0 = 0 ->
                Tpos = jigsaw/position(0,0)
        ;       Tpos = jigsaw/position(X,Y0)),
        (Y1 = 0 ->
                Bpos = jigsaw/position(0,0)
        ;       Bpos = jigsaw/position(X,Y1)),
        get_pieces(JspX,JspY,Xnext,Ynext, Adds), !.
```

```
Dec 30 20:10 1991   jsp-domain-ks.pl Page 1


%       FILE:             jsp-gen-kb.pl
%       DATE CREATED:     Fri Sep 8 11:22:26 BST 1989
%       AUTHOR:           Emilio Agust'in Molina (eam)
%

%---- Misplaced KS. Special case for the first piece placed.
%                  (There should be a problem!), the puzzle
%                  cannot be solved!)
%
if      #[jigsaw/position(X,Y),
          [piece(active), Values, Links,
           old_pieces(_),
           last_modified_by(noposition)],
           (not entry(agenda,ksar(body(jsp_update(jigsaw/position(X,Y),_,_,_),
                                  _,_,_),_,_),_))
           ]
%
then    true
%
to      amend   [jigsaw/position(X,Y),
                 [piece(active), Values, Links,
                  old_pieces([]),
                  last_modified_by(noposition)],
                 ]]
%
set     [misplacedKS,
         jigsaw/position(X,Y)]

%-      Misplaced KS.
%
if      #[jigsaw/position(X,Y),
          [piece(active), _, _, _,
           last_modified_by(jigsaw/position(Xlast,Ylast))],
           (not entry(agenda,ksar(body(jsp_update(jigsaw/position(X,Y),_,_,_),
                                  _,_,_),_,_),_))
           ]
    and #[jigsaw/position(Xlast,Ylast),
          [piece(Piece),
           .
           links([Tpos,Rpos,Bpos,Lpos]) | _],
          true]
    and #[Tpos, _, true]
    and #[Rpos, _, true]
    and #[Bpos, _, true]
    and #[Lpos, _, true]
    and #[Piece,_, true]
%
then    jsp_misplaced(jigsaw/position(Xlast,Ylast),Adds)
%
to      amend   Adds
%
% set   Ks
set     [misplacedKS,
         jigsaw/position(X,Y)].

%---    Activate Corners KS.
```

```
Dec 30 20:10 1991   jsp-domain-ks.pl Page 2


%
if      #[jigsaw/position(X,Y),
          [piece(in/active),
           values(Values),
           links(Links),
           old_pieces(Old),
           last_modified_by(Last)],
          is_corner(Values)]
%
then    true
%
to      amend   [jigsaw/position(X,Y),
                 [piece(active),
                  values(Values),
                  links(Links),
                  old_pieces(Old),
                  last_modified_by(Last)],
                 ]]
%
% set   Ks
set     [initKS,
         jigsaw/position(X,Y)]
```

```
Dec 30 20:10 1991  load Page 1

%
%       FILE:           load
%       DATE CREATED:   Wed Dec 13 14:46:15 GMT 1989
%       CREATED BY:     Emilio Agust'in Molina (eam)
%

%       Assimilate the Generator Knowledge Base.
%
:- assimilate('jsp-Gen-ks.pl').

%       Load the domain predicates.
%
  [-'jsp-Domain.pl'].

%       Load the control predicates.
%
  [-'jsp-Control.pl'].

%       Assimilate the Domain Knowledge Base.
%
  assimilate('jsp-domain-ks.pl').

%       Load the piece KBes. NOTE: change this to run other puzzle.
%
  assimilate('jsp-1x4-4.ks').

%       Initialize the blackboard.
%
  start.
```

```
Dec 30 20:10 1991  load_gen Page 1

%
%       FILE:           jsp-setup.pl
%       DATE CREATED:   Mon Sep 11 14:39:36 BST 1989
%       CREATED BY:     Emilio Agust'in Molina (eam)
%

%       Assimilate the Generate Knowledge Base.
%
:- assimilate('jsp-Gen-ks.pl').

%       Load the domain predicates.
%
  [-'jsp-Domain.pl'].

%       Load the control predicates.
%
  [-'jsp-Control.pl'].

%       Assimilate the Domain Knowledge Base.
%
  assimilate('jsp-domain-ks.pl').

%       Initialize the blackboard.
%
  start
```

Appendix D

An Introduction to Z Terms and Expressions

In the present appendix, we will introduce the Z specification language as defined by [Spi89]. We will provide descriptions for the notation used in this thesis. Most of the examples used here are taken from [Spi89]. For the formal definition of the notation, we advise the reader to consult [Spi89] directly.

The Z specification language begins by defining a number of *basic types*. For example, when we are dealing with names, and dates, we may define them as the basic types of the specification. This allows the definition of sets without specifying the type of objects they contain.

$$[NAME, DATE]$$

For the use within a specification, we can use *axiomatic descriptions*, mostly used of function definitions. For example, we can define the *square* function:

$$square : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall n : \mathbb{N} \bullet \quad square(n) = n * n$$

We can also use predicates on their own paragraph. These predicates act as independent *constraints*. For example, we could define that all the dates in the system should be greater than the first of September of 1971. This is assuming we have defined the ">" relation for dates.

$$\forall d : DATE \bullet d > 1/11/1971$$

This translates to "for all d in $DATE$, d is greater than 1/11/1971".

The main specification tool within the Z language is the *schema* definition. This allows the definition of the state space of the system to be specified. For example, the specification of the active blackboard system is:

ABBSystem

$$BBoard : BlackBoard$$
$$KSources : \mathbb{F}\ KS$$
$$PElements : \mathbb{F}\ Desk$$

$$\forall k_1, k_2 : KSources \bullet k_1.Name = k_2.Name \Rightarrow k_1 = k_2$$
$$\forall p_1, p_2 : PElements \bullet p_1.Name = p_2.Name \Rightarrow p_1 = p_2$$

This may also be defined in horizontal mode:

$ABBSystem \mathrel{\widehat=} [BBoard : BlackBoard, KSources : \mathbb{F}\ KS, PElements : \mathbb{F}\ Desk \mid ...]$

This is, an ABB system is made of a *Blackboard*, a finite set ($\mathbb{F}$) of *KSs* and a finite set of *Desks*. All KSs must have different manes (if they have the same name, then they are the same knowledge source), and the desks likewise.

We can also use *abbreviations*. These are a form of constant definition. They use the abbreviation definition symbol ==. For example, in the ABB system, we define a composite name called $PATH$, formed of a sequence of $NAME$s:

$PATH == \text{seq}\ NAME$

In Z, tuples are represented by $(X_1, ..., X_n)$, and sets by $\{Y_1, ..., Y_m\}$. This set representation is by enumeration. Z also allows the definition of sets by comprenhension. For example, the set of all even naturals can be defined as:

$Even == \{n : \mathbb{N} \mid \exists\, m : \mathbb{N} \bullet n = m * 2\}$

The power set of a given set X is defined as $\mathbb{P}\, X$. Also, the cartesian product between two sets X and Y is expressed as $X \times Y$.

We have already used a number of predicates in the previous examples. Z defines the following predicates:

$=, \in$	Equality, membership
$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$	Propositional connectives.
	Not, and, or, implication, equivalence
$\forall, \exists, \exists_1$	Quantifiers
	For all, exists, exists one

The propositional connectives, and quantifiers can also be applied to schemas. This allows the composition of different schemas to produce more complex specifications. Other operations that can be applied to schemas are:

$\setminus$	Schema hiding Hides the identifiers explicitly listed as its second argument
Δ	Schema state change specification
Ξ	Schema state test
$\fatsemi$	Sequential composition

Schemas are defined with an internal structure formed by a number of identifiers and their types. The $\setminus$ operator hides some of this structure. Several of the internal identified can be decorated or marked as input or output identifiers. The $\fatsemi$ operator composes two schemas whose input and output identifiers match both in name and in type.

The Z language allows the definition of generic constants. For example, instead of defining an empty set for every set, we can define the generic empty set by:

$$[X]$$

$$\varnothing : \mathbb{P}\, X$$

$$\varnothing = \{x : X \mid \mathit{false}\}$$

In Z, *true* and *false* are the usual boolean constants.

Besides the definitions and operations presented above, the Z language provides a mathematical toolkit for the operation with sets, relations, functions, numbers, and sequences. It also provides tools for delaing with iteration and finiteness.

The set operations are:

$\neq, \notin$	Inequality, non-membership
$\varnothing, \subseteq, \subset, \mathbb{P}_1$	Empty set, subsets, non-empty sets
$\cup, \cap, \setminus$	Union, intersection, difference
$\bigcup, \bigcap$	generalized union and intersection.
first, second	Projection functions for ordered pairs.

The relation operations are:

$\leftrightarrow, \mapsto$	Binary relations, "maps to" operation
dom, ran	Domain and range of a relation
id, $\fatsemi$, $\circ$	Identity relation, composition, backward composition

The maps to operation is equivalent to ordered pairs, and can be used to define

partial functions by enumeration.

The function operations are:

$\rightarrowtail\!\!\!\!\!\!+$, $\rightarrow$	Partial and total functions
$\rightarrowtail\!\!\!\!\!\!+$, $\rightarrowtail$	Partial and total injections
$\oplus$	Functional overriding

The operations and definitions on numbers are:

$\mathbb{N}$, $\mathbb{Z}$	Natural numbers, integers
$+$, $-$, $*$, div, mod, $<$, $\leq$, $>$, $\geq$	Usual operations on numbers
$\mathbb{N}_1$, succ	Positive numbers, successor function
R^k	iteration
$\mathbb{F}$, $\mathbb{F}_1$	Finite set, non empty finite set
#	number of members
min, max	minimum, maximum numbers

The operations on sequences are:

seq, seq_1	Finite and non empty sequences
$\frown$	Concatenation
head, last, tail, front	Sequence decompositions
rev	Reverse
$\upharpoonright$	Sequence filtering

Appendix E

The ABB Model Detailed Specification

E.1 Introduction

In Chapter 4, we already presented a guide to the Z specification of the ABB model. In this appendix we will provide the detailed ABB Model specification. We will also include the information presented in Chapter 4 as it helps to explain the full specification. For a complete description of the Z language, used in this appendix, please read [Spi89].

The specification of the system as a whole will be called *ABBSystem*. This will be composed, by a global blackboard, a set of knowledge sources and a set of desks. These three elements will be described by the *BlackBoard*, *KS*, and *Desk* specifications. The form and structure of these elements will be defined later. For the moment, we will treat them as Z basic types. This just means that they are to be regarded as sets of generic objects or elements.

$[ABBSystem, BlackBoard, KS, Desk]$

Within the system there is the need to provide names to the different elements, be they blackboard elements, knowledge sources or desks. We will assume the existence of a set of names called *NAME*. The ABB system components will have composite names formed by sequences of *NAME* elements ($\text{seq}\ NAME$). These sequences will be called *PATH*'s, defined using the Z abbreviation definition ($==$). They will be used, for example, to name the different blackboard object in a way that reflect the hierarchical structure of the blackboard. We can also define the absence of a path, or the null path, to be an empty sequence ($\langle\rangle$).

$[NAME]$
$PATH == \text{seq}\ NAME$
$NullPATH == \langle\rangle$

The information and knowledge needed by the different elements of the model is symbolised by the *INFO* set. The *NullINFO* element represents the absence of information.

$[INFO]$
$NullINFO \in INFO$

Within the ABB system, there is also the need to manipulate information associated with the different system components. This is useful for controlling blackboard access and knowledge source execution, for example. Each system component will have some information that may change with time or in certain situations. We will associate a set of information with each system component. We represent this by a function ($\rightarrow$) that associates information sets (*finsetINFO*) to system component names (*PATH*). The expression $\mathbb{F}\, X$ represents the set of all finite subsets of X. The only restriction of such functions, is that all information comming from different system elements must be different, or as we express on the Z expression below, if the information of two system elements is equal, then they are the same element. This can be easily accomplished by including the element path on all its information. The symbol $\bullet$ is the Z notation for "*Such That*".

$$InfoOf : PATH \rightarrow \mathbb{F}\, INFO$$
$$\forall p_1, p_2 : PATH \bullet \exists\, i_1 : InfoOf(p_1);\ i_2 : InfoOf(p_2) \bullet (i_1 = i_2 \Rightarrow p_1 = p_2)$$

The ABB model contemplates the interaction, via a blackboard, of a set of experts working on desks. The experts are represented by knowledge sources executing on desks, and interact via a global data structure called the blackboard. When performing their tasks, the knowledge sources and the other ABB system components perform operations on the system. These operations can query about the status of the system, or can actually produce a change on it. We specify them using the Z generic constant syntaxis. A system operation (*SystemOp*) is a partial function that receives the currentr state of the system and returns some response.

$$[X]$$
$$SystemOp : ABBSystem \rightarrow\!\!\!\!\!\!\;\; X$$

This response can be boolean (the result of a test), a system (when its state is modified), or any other information.

$$TestOp == SystemOp[\{true, false\}]$$
$$ModifyOp == SystemOp[ABBSystem]$$

We will represent any given procedure involving system operations as a sequence of those operations. These procedures will serve to specify the actions of the different system elements.

$Procedure == \mathrm{seq}\, SystemOp$

E.2 Knowledge Sources

The ABB knowledge sources perform three basic functions. These are:

Identify: The recognition of problems that fall within the area of expertise of the knowledge source.

Prioritise: The ordering of the pending problems, according to their importance in a given moment. This is part of the control structure of the ABB model.

Solve: The actual resolution of a problem.

The knowledge about how to identify suitable problems to be solved by the knowledge source, is encoded within a condition. Given the current state of the blackboard, this condition succeeds if there is one of such problems. The knowledge about how to solve a particular problem is encoded into the body of the knowledge source. Given some information of the problem that has been identified by the trigger condition, the knowledge source body solves it and changes the state of the blackboard to reflect its solution.

The communication between the trigger condition and the body on the knowledge source is done via a *KSAR*. This *KSAR* will contain the PATH of the blackboard element where the problem was identified (*Path*), some information about the knowledge source that it activates (*KsInfo*), and a finite set of fields specifying the information about the problem needed by the knowledge source body (*Fields*). The actual contents of the information specified by each field is also kept in the KSAR. We can specify this by a partial function ($\rightarrowtail$) that associates each field name with its content (*Content*). We specify the KSAR using the Z schema syntaxis. This defines the object structure and a set of conditions that this structure must satisfy. In the case of *KSAR*, the domain (dom) of the content function must be the set of fields of the KSAR. This means that each field must have a content and that there will be no more contents than fields.

KSAR

$Path : PATH$
$KsInfo : INFO$
$Fields : \mathbb{F}\ NAME$
$Content : NAME \twoheadrightarrow INFO$

$\text{dom } Content = Fields$

The initial state of a *KSAR* is one in which it has no contents, or rather, in which its fields contain *NullINFO*. Whenever a new ksar is generated, it starts up in this initial state.

NullKSAR

$KSAR$

$Content = \{\ n \mapsto NullINFO \mid n \in Fields\ \}$

Within the ABB model, control is performed by a set of functions that operate mainly on sets of ksars. We can identify two kinds of such functions. First, there are a number of (possibly partial) functions that take a finite set of ksars and return a modified subset of it. These functions will be called *FilterCFn*. Second, there are the functions that take a set of ksars and return one of them. We will call these functions, *SelectCFn*.

$$FilterCFn == \mathbb{F}\ KSAR \twoheadrightarrow \mathbb{F}\ KSAR$$

$$SelectCFn == \mathbb{F}\ KSAR \rightarrow KSAR$$

The individual problem identification is made by the trigger condition via a set of activation conditions (*ActivationCond*). Each one of these conditions contains a blackboard operation (*ActivationOp*) that tests for a specific type of problem and, when it succeeds, returns an activation record (*Test*).

$$ActivationOp == BlackBoard \rightarrow KSAR$$

The activation conditions are targeted to a specific blackboard level, represented by its *PATH* (*BBlevel*). It also defines the structure of the KSARs produced by its *Test* by specifying the set of fields that they should have (*Fields*), and the information

about the knowledge source (*KsInfo*).

ActivationCond

$Test : ActivationOp$
$BBlevel : PATH$
$Fields : \mathbb{F}\ NAME$
$KsInfo : INFO$

$\forall k : KSAR \mid k \in \operatorname{ran} Test \bullet$
$\quad last\ k.Path = BBlevel \land k.Fields = Fields \land k.KsInfo = KsInfo$

There are, also, a number of special activation conditions that are satisfied when, and only when, the system starts. These are called *StartCond*'s, and are characterised by having an empty path. Note that the ksars produced by these conditions also have an empty path. The *StartCond*'s are used to specify the initial knowledge sources that will begin the problem solving activity of the system.

StartCond

$ActivationCond$

$BBLevel = NullPATH$

A given trigger condition will contain a set of activation conditions, an internal structure, and a reference to its knowledge source. All the activation conditions will have information about the knowledge source. This information will be used to build ksars, when necessary. The structure of those ksars will be inherited from the trigger condition. The trigger condition schema is then:

TriggerCond

$Conditions : \mathbb{F}_1\ ActivationCond$
$Fields : \mathbb{F}\ NAME$
$KsName : PATH$

$\forall c : Conditions \bullet$
$\quad c.KsInfo \in InfoOf(KsName) \land c.Fields = Fields$

When an *ActivationCond* within a *TriggerCond* succeeds, a *KSAR* is produced and submitted to the system for consideration. In a particular moment, the system may

decide to process it. The processing of the ksars is performed by a procedure called the *Body* of the knowledge source. This procedure is where the knowledge source actually solves the problems within its domain. The body procedure takes the ksar to be processed and as a result, it can modify the state of the system (add or modify information in the blackboard, for example).

$$KSBodyProc == KSAR \nrightarrow Procedure$$

The knowledge source, then, is defined by an unique name, a trigger condition that identifies its problem domain, a knowledge source control function that prioritises its work, and the body that actually solves the problems. The list of ksars, representing the problems that remain to be solved is kept in a local agenda (*KSAgenda*). The trigger condition keeps a reference to the knowledge source, represented by its name. The control function must be defined for all possible local agendas. The body of the knowledge source must only accept ksars produced by it. Finally, the knowledge source local agenda is a set of ksars produced by its trigger condition.

KnowledgeSource

$Name : PATH$
$Trigger : TriggerCond$
$KSCf : FilterCFn$
$Body : KSBodyProc$
$KSAgenda : \mathbb{F}\ KSAR$

$KsInfo \in InfoOf(Name)$
$Trigger.KsName = Name$
$\exists K : \mathbb{F}\ KSAR \mid K = \{ k : KSAR \mid k.KsInfo \in InfoOf(Name) \land k.Fields = Trigger.Fields \} \bullet$
$(\forall s : \mathbb{F}\ K \bullet s \in \mathrm{dom}\ KSCf) \land$
$(\forall k : \mathrm{dom}\ Body \bullet k \in K) \land$
$(KSAgenda \subseteq K)$

The knowledge sources that have one *StartCond* within its trigger conditions will be called initialisation knowledge sources (*InitKS*'s). They are activated when the system starts, and are specially suited for performing all the initialisations that may be needed by it.

$InitKS$
$KnowledgeSource$

$\exists c : StartCond \bullet c \in Trigger.Conditions$

In a given ABB system there will be a set of knowledge sources, each of which will have a set of activation conditions. Whenever the blackboard is modified, the conditions interested in those changes are evaluated. Let A be the set of activation records resulting from the above evaluation, we can define the *AddKSARs* function to include them into the knowledge source agenda. The result of this function is a modified knowledge source. The ksars being added, of course, must correspond to the given knowledge source.

$AddKSARs : KS \times \mathbb{F}\, KSAR \rightarrow KS$

$\forall K : KS;\ A : \mathbb{F}\, KSAR \mid (\forall k : A \bullet k.KsInfo \in InfoOf(K.Name)) \bullet$
$\quad \exists K' : KS \mid K' = AddKSARs(K, A) \bullet$
$\quad\quad K' \setminus KSAgenda = K \setminus KSAgenda \wedge$
$\quad\quad K'.KSAgenda = K.KSAgenda \cup A$

E.3 Blackboard

The ABB blackboard is a structure used to store information of global interest. This information is structured in blackboard elements (bbels) stored in different levels within a level hierarchy. A bbel is a structure with a name, and a set of fields with associated contents. The type of a particular bbel is defined by the number and type of its fields.

$BBel$
$Name : PATH$
$Fields : \mathbb{F}\, NAME$
$Contents : NAME \twoheadrightarrow INFO$

$\mathrm{dom}\ Contents = Fields$

The initial state of the bbels of a given type is defined by the *NullBBel* schema. This defines a bbel whose fields have no information defined.

$$\begin{array}{|l}
\textit{NullBBel} \\
\hline
\textit{BBel} \\
\hline
\textit{Contents} = \{\ n \mapsto \textit{NullINFO} \mid n \in \textit{Fields}\ \}
\end{array}$$

There are a number of procedures associated to all the bbels of a given type. These procedures are, an initialisation procedure (*InitBBel*), a set of *Guard*'s, and a set of *Trigger*'s. The initialisation procedure is used to define a number of default values for bbels. It is evaluated whenever a new bbel is created. The default values should be defined for some (maybe all) of the fields of the bbel. These values will substitute the values already in the bbel.

$$\begin{array}{|l}
\textit{InitBBel} \\
\hline
\Delta \textit{BBel} \\
\textit{Defaults?} : \textit{NAME} \twoheadrightarrow \textit{INFO} \\
\hline
\mathrm{dom}\ \textit{Defaults?} \subseteq \textit{Fields} \\
\textit{Contents}' = \textit{Contents} \oplus \textit{Defaults?}
\end{array}$$

When a new bbel is created, a *NullBBel* is generated and then initialised with *InitBBel*. We can define the initial bbel as follows.

$$\textit{InitialBBel} \mathrel{\widehat=} \textit{NullBBel} \mathbin{;} \textit{InitBBel}$$

The guards are operations that are evaluated whenever a system component tries to access a bbel within the blackboard. They consist on a test about the status of the blackboard, upon which they decide whether to grant or deny the access to a bbel. This facility can be used to hide some areas of the blackboard to specific blackboard system elements. This allows the implementation of security schemes.

$$\textit{Guard} == \textit{INFO} \rightarrow (\textit{BlackBoard} \rightarrow \{\ \textit{true}, \textit{false}\ \})$$

The triggers are operations evaluated after a change has been made to the blackboard. Given the bbel modified, the trigger may change the state of the system.

$$\textit{Trigger} == \textit{BBel} \rightarrow \textit{ModifyOp}$$

Both guards and triggers have associated names that identify them unequivocally.

The set of guards and triggers of a particular type of bbel can be seen as partial functions that map the different names to the actual guards or triggers.

$$Guards_{bbel} == NAME \pfun Guard$$

$$Triggers_{bbel} == NAME \pfun Trigger$$

Bbels of the same type are grouped together within a level. The level itself can be seen as the definition of a particular type of bbels. An ABB level is defined by a structure that holds its name, a set of fields that define the type of the bbels it holds, a bbel initialisation procedure, a set of triggers, a set of guards, the actual set of bbels stored in it, the blackboard control function, and the set of activated triggers, waiting to be evaluated.

The name of all the bbels within a level is formed by the composition of the name of the level and a bbel name, unique within it. The level also defines the structure of the bbels stored in it. This is, the fields of the bbels are defined to be the same fields as its level. The level initialisation procedure must define the contents of those fields defined by the level. It can define defaults for all or part of the level's fields. The blackboard control function operates on ksars generated in this level. The default blackboard control function is the identity function.

LevelStructure

$$\begin{array}{l}
Name : PATH \\
Fields : \mathbb{F}\ NAME \\
Init : InitialBBel \\
Triggers : NAME \pfun Trigger \\
Guards : NAME \pfun Guard \\
BBels : \mathbb{F}\ BBel \\
BBCf : FilterCFn \\
Eval : \mathbb{F}\ ModifyOp \\
\hline
\forall b : BBels \bullet \exists n : NAME \bullet b.Name = \langle n \rangle \frown Name \\
\forall b_1, b_2 : BBels \bullet (b_1.Name = b_2.Name) \Rightarrow (b_1 = b_2) \\
\forall b : BBels \bullet b.Fields = Fields \\
Init.Fields = Fields \\
\mathrm{dom}\ Init.Defaults? \subseteq Fields \\
\forall k : KSAR \mid k \in \mathrm{dom}\ BBCf \cup \mathrm{ran}\ BBCf \bullet last\ k.Path = Name \\
\exists f : FilterCFn \bullet BBCf = \mathrm{id}(\mathbb{F}\ KSAR) \oplus f
\end{array}$$

Note that there is no level-wide agenda. This is because the blackboard agenda is specific to each bbel that has been modified. A given level may have several agendas. Furthermore, we consider not necessary to store these agendas. When a bbel is modified, the relevant activation conditions can be evaluated. The set of ksars they return can be processed, then, by the level control function and the resulting set of ksars can be sent directly to the appropriate knowledge sources. This is possible because the blackboard agendas are produced by knowledge source activation conditions, and they do not modify the state of the blackboard.

An ABB level is then a tree-like structure that has a level structure on each node and a finite set of sub-levels.

$$Level ::= LevelStructure \times \mathbb{F}\ Level$$

We define two functions to access both the level structure and its sub-levels.

$$\begin{array}{l} Struct : Level \rightarrow LevelStructure \\ SubLevels : Level \rightarrow \mathbb{F}\ Level \\ \hline \forall\, l : Level \bullet \\ \quad Struct(l) = first\ l \land \\ \quad SubLevels(l) = second\ l \end{array}$$

In order to be properly defined, the level must satisfy a set of laws that regulate the form of its components.

The name formation rule for the sublevels is identical to the one for the bbels within the level's structure. This insures that the name of any particular level or bbel reflects its position within the blackboard level hierarchy.

$$\begin{array}{l} \forall\, l : Level \bullet \\ \quad (\forall\, l' : SubLevels(l) \bullet \exists\, n : NAME \bullet \\ \quad\quad Struct(l').Name = \langle n \rangle \frown Struct(l).Name) \land \\ \quad (\forall\, l_1, l_2 : SubLevels(l) \bullet \\ \quad\quad Struct(l_1).Name = Struct(l_2).Name \Rightarrow l_1 = l_2) \end{array}$$

The names of the bbels and sublevels within a level must be all unique.

$$\begin{array}{l} \forall\, l : Level \bullet \\ \quad \forall\, b : Struct(l).BBels;\ l' : SubLevels(l) \bullet b.Name \neq Struct(l').Name \end{array}$$

A given level partially defines the structure of its sub-levels. They inherit the fields of of their parent level. This is, the fields of each sub-level are the fields of the parent level plus some other specific to itself.

$$\forall\, l : Level \bullet \forall\, l' : SubLevels(l) \bullet Struct(l).Fields \subseteq Struct(l').Fields$$

The structure of each sub-level initialisation procedure is partially defined, also, by its parent level. The defaults defined in the level initialisation procedure will be used unless overridden by the sub-level.

$$\begin{array}{l} \forall\, l : Level \bullet \forall\, l' : SubLevels(l) \bullet \\ \quad \exists\, i : InitialBBel \mid i.Fields = Struct(l').Init.Fields \bullet \\ \quad\quad Struct(l').Init.Defaults? = Struct(l).Init.Defaults? \oplus i.Defaults? \end{array}$$

The blackboard control function is also inherited, unless the sub-levels override it. This is, the sub-level control function is defined by the parent's function in all of the elements of the domain, except where defined locally.

$$\begin{array}{l} \forall\, l : Level \bullet \forall\, l' : SubLevels(l) \bullet \exists\, f : FilterCFn \bullet \\ \quad Struct(l').BBCf = Struct(l).BBCf \oplus f \end{array}$$

Finally, the guards and triggers are also inherited. The sub-levels can redefine the parent's guards and triggers by defining new procedures with the same associated names.

$$\begin{array}{l} \forall\, l : Level \bullet \forall\, l' : SubLevels(l) \bullet \\ \quad \exists\, t : NAME \twoheadrightarrow Trigger;\ g : NAME \twoheadrightarrow Guard \bullet \\ \quad\quad (Struct(l').Triggers = Struct(l).Triggers \oplus t) \wedge \\ \quad\quad (Struct(l').Guards = Struct(l).Guards \oplus g) \end{array}$$

Given the previous level definition, any given level can be seen as a set of level structures. These structures are embedded within a tree-like organization. The set of all the level structures held within a level can be defined as follows.

$$
\begin{array}{|l}
LevelsOf : Level \rightarrow \mathbb{F}\ LevelStructure \\
\hline
\forall\, l : Level \bullet \\
\quad LevelsOf(l) = \\
\qquad \{\ Struct(l)\ \} \cup \\
\qquad \{\ s' : LevelStructure \mid \exists\, l' : SubLevels(l) \mid s' \in LevelsOf(l')\ \}
\end{array}
$$

Given the level naming convention used, it is possible to reconstruct the full level structure from the level's structure set. However, not all sets of level structures can form a proper level structure. They must satisfy all the level laws.

Besides the usual equality, we will need a slightly relaxed form of equality for level comparison. This is the static structure equality, and it is based of the fact that levels, within a blackboard system, are only modified in the *BBels* and *Eval* sets. Two levels will have identical static structure if they are equal in every component, except in those two.

$$
\begin{array}{|l}
EqStaticStruct : Level \times Level \\
\hline
\forall\, l_1, l_2 : Level \bullet \\
\quad EqStaticStruct(l_1, l_2) \Leftrightarrow \\
\qquad Struct(l_1) \setminus (BBels, Eval) = Struct(l_2) \setminus (BBels, Eval) \wedge \\
\qquad (\forall\, l_1' : SubLevels(l_1) \bullet \exists\, l_2' : SubLevels(l_2) \bullet EqStaticStruct(l_1', l_2')) \wedge \\
\qquad (\forall\, l_2' : SubLevels(l_2) \bullet \exists\, l_1' : SubLevels(l_1) \bullet EqStaticStruct(l_1', l_2'))
\end{array}
$$

The ABB blackboard is a level identical to any other. Its only particularity is that it is the top of the level hierarchy. As such, its associated path is a sequence of only one name.

$$
\begin{array}{l}
BlackBoard \subset Level \\
\forall\, l : Blackboard \bullet \exists\, n : NAME \bullet Struct(l).Name = \langle n \rangle
\end{array}
$$

When defining a blackboard for a particular application, we will place the common global structure and defaults in the blackboard level. Given the problem domain information dependencies, this blackboard will have as many sub-levels as necessary. The sub-levels will inherit the global structure which will be extended with particular specialisations. These sub-levels can be further subdivided to suit the problem structure.

The ABB blackboard defines a data and knowledge hierarchy based on structure

and procedure inheritance. This is similar to the hierarchy found in most object oriented languages and systems. The ABB root object class is the blackboard and all other classes of objects are derived from it.

E.4 Blackboard Operations

When any ABB system element needs to access or modify the blackboard structure, it must make it using the blackboard operations available. The ABB system defines four basic blackboard operations that allow the creation, access, modification and deletion of blackboard elements.

Before any of these functions can be performed, the guards corresponding to the affected bbels must be executed. The set of guards of the bbels touched by a given operation is defined as follows.

$$GuardsOf : \mathbb{F}\, PATH \times Level \rightarrow \mathbb{F}\, Guard$$

$$\begin{aligned}
&\forall P : \mathbb{F}\, PATH;\ l : Level \bullet \\
&\quad GuardsOf(P, l) = \{\ g : Guard \mid \exists\, b : Struct(l).BBels \mid b.Name \in P \bullet \\
&\qquad\qquad g \in \operatorname{ran} Struct(l).Guards\ \} \cup \\
&\qquad\qquad \{\ g : Guard \mid \exists\, l' : SubLevels(l) \mid g \in GuardsOf(P, l')\ \}
\end{aligned}$$

In order to find out if the guards allow a given operation, we need information about the system element that is performing it, the set of bbels that may be affected by it, and the actual blackboard being operated.

$$CheckGuards : INFO \times \mathbb{F}\, PATH \times Level$$

$$\begin{aligned}
&\forall i : INFO;\ P : \mathbb{F}\, PATH;\ l : Level \bullet \\
&\quad (l \notin BlackBoard) \vee \\
&\quad ((l \in Blackboard) \wedge (\forall g : GuardsOf(P, l) \bullet g(i)(l)))
\end{aligned}$$

After the operations are executed, the state of the blackboard may have changed. If this is the case, the triggers of the bbels that took part on the operation must be activated. The operations will place these triggers in the level evaluation list.

The creation of bbels is performed via the *LCreateBBels* operation. It takes information about the system element performing the operation (needed for guard evaluation), a set of bbel names (paths) and a level. If the guards allow the modification, the operation returns a modified level with the new bbels added. The modified level

will be equal to the initial one except in its *SubLevels*, *BBels* and *Eval* attributes. The new sub-levels will be the the old ones, with the relevant bbels created. The new bbels will be the old bbels, plus the new bbels corresponding to this level. These new bbels are duplicates of the *InitialBBel*, with the appropriate names, and they must not be already created. The new *Eval* set will be the previous one, plus all the activated triggers.

$$
\begin{array}{l}
LCreateBBels : INFO \times \mathbb{F}\ PATH \times Level \rightarrow Level \\
\hline
\forall i : INFO;\ P : \mathbb{F}\ PATH;\ l : Level \bullet \exists l' : Level \mid LCreateBBels(i, P, l) = l' \bullet \\
\quad ((\neg\ CheckGuards(i, P, l)) \wedge l' = l) \vee \\
\quad (CheckGuards(i, P, l) \wedge EqStaticStruct(l, l') \wedge \\
\quad\ (\exists B : \mathbb{F}\ BBel \mid (B = \{\ b : BBel \mid (b.Name \in P) \wedge \\
\qquad\qquad (b \setminus Name = Struct(l).Init \setminus (Defaults?, Name)) \wedge \\
\qquad\qquad (Struct(l).Name = tail\ b.Name) \wedge \\
\qquad\qquad (\forall b' : Struct(l).BBels \bullet b'.Name \neq b.Name) \\
\qquad\qquad (\forall s' : SubLevels(l') \bullet s'.Name \neq b.Name)\ \}) \bullet \\
\qquad (Struct(l').BBels = Struct(l).BBels \cup B) \wedge \\
\qquad (Struct(l').Eval = Struct(l).Eval \cup \\
\qquad\qquad \{\ \odot : ModifyOp \mid \exists b : B \mid \exists \triangle : \mathrm{dom}\ Struct(l).Triggers \mid \\
\qquad\qquad\quad \odot = \triangle(b)\ \})) \wedge \\
\quad\ (SubLevels(l') = \{\ l_2 : Level \mid \\
\qquad\qquad \exists l_1 : SubLevels(l) \mid l_2 = LCreateBBels(i, P, l_1)\ \}))
\end{array}
$$

The *LReadBBels* function allows the access to the blackboard elements. It takes a set of bbel names (paths) and a level, and if the guards succeed, it returns the set of bbels with the indicated names.

$$
\begin{array}{l}
LReadBBels : INFO \times \mathbb{F}\ PATH \times Level \rightarrow \mathbb{F}\ BBel \\
\hline
\forall i : INFO;\ P : \mathbb{F}\ PATH;\ l : Level \bullet \\
\quad ((\neg\ CheckGuards(i, P, l)) \wedge LReadBBels(i, P, l) = \varnothing) \vee \\
\quad (CheckGuards(i, P, l) \wedge \\
\quad\ LReadBBels(P, l) = \{\ b : Struct(l).BBels \mid b.Name \in P\ \} \cup \\
\qquad\qquad \{\ b : BBel \mid \exists l' : SubLevels(l) \mid b \in LReadBBels(i, P, l')\ \})
\end{array}
$$

The modification of bbels is performed via the *LModifyBBels* function. The modifications are represented by a bbel with the same name as the one we want to

modify. Its fields must be a subset of those of the target bbel. Its contents will replace those on the target bbel. In a way similar to the creation function, *LModifyBBels* takes information about the element performing the modifications, the set of bbels to be modified and a level. If the guards succeed, it returns the given level with the modifications done. The return level will be equal to the initial one except in its *BBels*, *SubLevels* and *Eval* attributes. The new bbel set will be the previous one with the affected bbels modified. The new sub-levels also change. They will be the the old ones, with the relevant bbels modified. The new evaluation set will be the previous one with all the triggers activated by the operation.

$$LModifyBBels : INFO \times \mathbb{F}\ BBel \times Level \rightarrow Level$$

$$\begin{array}{l}
\forall\, i : INFO;\ B : \mathbb{F}\ BBel;\ l : Level \bullet \\
\quad \exists\, l' : Level \mid LModifyBBels(i, B, l) = l' \bullet \\
\qquad ((\neg\ CheckGuards(i, P, l)) \wedge l' = l) \vee \\
\qquad (CheckGuards(i, P, l) \wedge EqStaticStruct(l, l') \wedge \\
\qquad\ (\exists\, B' : \mathbb{F}\ BBel \mid (B' = \{\ b : BBel \mid \exists\, b' : Struct(l).BBels \mid \exists\, b'' : B \mid \\
\qquad\qquad\quad (b''.Name = b'.Name) \wedge (b''.Fields \subseteq b'.Fields) \wedge \\
\qquad\qquad\quad (b \setminus Contents = b' \setminus Contents) \wedge \\
\qquad\qquad\quad (b.Contents = b'.Contents \oplus b''.Contents)\ \}) \bullet \\
\qquad\qquad (Struct(l').BBels = Struct(l).BBels \\
\qquad\qquad\quad \setminus \{\ b : Struct(l).BBels \mid \exists\, b' : B \mid b.Name = b'.Name\ \} \\
\qquad\qquad\quad \cup B') \wedge \\
\qquad\qquad (Struct(l').Eval = Struct(l).Eval \cup \\
\qquad\qquad\quad \{\ \odot : ModifyOp \mid \exists\, b : B \mid \exists\, \triangle : \mathrm{dom}\ Struct(l).Triggers \mid \\
\qquad\qquad\qquad \odot = \triangle(b)\ \})) \wedge \\
\qquad\ (SubLevels(l') = \{\ l_2 : Level \mid \\
\qquad\qquad\quad \exists\, l_1 : SubLevels(l) \mid l_2 = LModifyBBels(i, B, l_1)\ \}))
\end{array}$$

Finally, the *LDeleteBBels* function allows the deletion of bbels from the blackboard. It takes a list of bbel names and a level, and returns the level with the bbels with those names removed.

$$
\begin{array}{|l}
LDeleteBBels : INFO \times \mathbb{F}\, PATH \times Level \rightarrow Level \\
\hline
\forall i : INFO;\ P : \mathbb{F}\, PATH;\ l : Level \bullet \\
\quad \exists\, l' : Level \mid LDeleteBBels(i, P, l) = l' \bullet \\
\qquad ((\neg\ CheckGuards(i, P, l)) \land l' = l) \lor \\
\qquad (CheckGuards(i, P, l) \land EqStaticStruct(l, l') \land \\
\qquad\ (SubLevels(l') = \{\ l_2 : Level \mid \\
\qquad\qquad \exists\, l_1 : SubLevels(l) \mid l_2 = LDeleteBBels(i, P, l_1)\ \}) \land \\
\qquad\ (Struct(l') \setminus (BBels, Eval) = Struct(l) \setminus (BBels, Eval)) \land \\
\qquad\ (\exists\, B : \mathbb{F}\, BBel \mid B = \{\ b : Struct(l).BBels \mid (b.Name \in P)\ \} \bullet \\
\qquad\quad (Struct(l').BBels = Struct(l).BBels \setminus B) \land \\
\qquad\quad (Struct(l').Eval = Struct(l).Eval \cup \\
\qquad\qquad \{\ \odot : ModifyOp \mid \exists\, b : B \mid \exists\, \triangle : \mathrm{dom}\ Struct(l).Triggers \mid \\
\qquad\qquad\quad \odot = \triangle(b)\ \})))
\end{array}
$$

We can also describe the set of levels that were modified by any given system operation. Let l_1 be a level (maybe the blackboard) before the operation and l_2 this level after the operation. The names of the modified levels is formed by the names of those levels with the same name in l_1 and l_2 that are also different. This is, they must be different due to some modification to the *BBels* or *Eval* sets. Furthermore, it can be seen in the previous operations that the *Eval* set is modified only when the *BBels* set changes.

$$
\begin{array}{|l}
ModifiedLevels : Level \times Level \rightarrow \mathbb{F}\, PATH \\
\hline
\forall\, l_1, l_2 : Level \bullet \\
\qquad ModifiedLevels(l_1, l_2) = \\
\qquad\qquad \{\ p : PATH \mid \exists\, s_1 : LevelsOf(l_1) \mid \exists\, s_2 : LevelsOf(l_2) \mid \\
\qquad\qquad\quad s_1.Name = S_2.Name \land s_1 \neq s_2 \land s_1.Name = p\ \}
\end{array}
$$

E.5 Desks

The desks are the ABB system components that represent the place where the experts (knowledge sources) perform their work. They will hold knowledge about how to administer their processing resources. This knowledge is embodied in the desk control function. This function decides when a given knowledge source should execute.

A desk is formed by a name that identifies it uniquely, a set of the names of the knowledge sources that may execute in it, the list of ksars of those knowledge sources

ready to be evaluated, and the local control function. Each knowledge source will be able to execute in at least one of the desk of the system.

Desk

$Name : PATH$
$KSnames : \mathbb{F}\ PATH$
$DeskAgenda : \mathbb{F}\ KSAR$
$DeskCf : SelectCFn$
$ActualKS : PATH$
$Eval : Procedure$

$\forall k : DeskAgenda \bullet \exists n : KSnames \bullet k.KsInfo \in InfoOf(n)$
$DeskAgenda \in \mathrm{dom}\ DeskCf$

E.6 The ABB System

The ABB system, as we described in the model, is conformed by a global blackboard, a set of knowledge sources, and a set of desks. These system elements interact with each other in a way that the system as a whole works toward the solution of a problem in the domain.

There are a number of conditions that the system elements must satisfy, in order to insure the proper execution of the system. In order to avoid possible confusion, we define that all knowledge sources and all desks must have different names. All knowledge sources must be able to be evaluated in, at least, one desk. Finally, there must be at least one initialisation knowledge source.

ABBSystem

$BBoard : BlackBoard$
$KSources : \mathbb{F}\ KS$
$Desks : \mathbb{F}\ Desk$

$\forall k_1, k_2 : KSources \bullet k_1.Name = k_2.Name \Rightarrow k_1 = k_2$
$\forall p_1, p_2 : Desks \bullet p_1.Name = p_2.Name \Rightarrow p_1 = p_2$
$\bigcup \{ X : \mathbb{F}\ PATH \mid (\exists p : Desks \bullet X = p.KSnames) \}$
$\quad = \{ x : PATH \mid (\exists k : KSources \bullet x = k.Name) \}$
$\exists k : KSources \bullet k \in InitKS$

When the system starts its operation only the initialisation knowledge sources are triggered. Their activation records are stored in their respective desks. The knowledge sources then have empty agendas. The blackboard, in this initial state, is empty and totally inactive. This is, it has no triggers waiting to be evaluated. The ABB system initial state is then:

$$
\begin{array}{l}
\textit{InitialABBSystem} \\
\hline
\textit{ABBSystem} \\
\hline
\forall E : Desks \bullet \\
\quad E.DeskAgenda = \{\ k : KSAR \mid \exists K : KSources \mid \\
\qquad\qquad K \in InitKS \land k.Path = NullPATH \land \\
\qquad\qquad k.KsInfo \in InfoOf(K.Name)\ \} \land \\
\quad E.Eval = \langle\rangle \land E.ActualKS = NullPATH \\
\forall K : KSources \bullet K.KSagenda = \varnothing \\
\forall s : LevelsOf(BBoard) \bullet s.Eval = \varnothing \land s.BBels = \varnothing \\
\hline
\end{array}
$$

Note that it is possible to obtain the initial state from any *ABBSystem*. This initial state is identical to the given *ABBSystem*, with its bbels, evaluation sets, and agendas emptied and reinitialised.

$$
\begin{array}{l}
InitialSystem : ABBSystem \rightarrow InitialABBSystem \\
\hline
\forall S : ABBSystem \bullet \exists S_i : InitialABBSystem \mid InitialSystem(S) = S_i \bullet \\
\quad EqStaticStruct(S.BBoard, S_i.BBoard) \land \\
\quad (\forall K : S.KSources \bullet \exists K_i : S_i.KSources \bullet \\
\quad\ \ K.Name = K_i.Name \land E \setminus KSAgenda = K_i \setminus KSAgenda) \land \\
\quad (\forall K : S_i.KSources \bullet \exists K : S.KSource \bullet K.Name = K_i.Name) \land \\
\quad (\forall E : S.Desks \bullet \exists E_i : S_i.Desks \bullet E.Name = E_i.Name \land \\
\quad\ \ E \setminus (Eval, ActualKS, DeskAgenda) = \\
\qquad\qquad\qquad E_i \setminus (Eval, ActualKS, DeskAgenda)) \land \\
\quad (\forall E : S_i.Desks \bullet \exists E : S.Desks \bullet E.Name = E_i.Name)
\end{array}
$$

From the initial state, the ABB system will begin operation. One of the desks holding ksars will choose one to be evaluated. The body of the appropriate knowledge source will be taken by it and it will begin executing its operations. Meanwhile another desk can be doing the same. When the initial knowledge sources execute, they will

be changing the state of the blackboard. This will provoke the activation of other knowledge sources, as well as the evaluation of blackboard triggers. All this execution will produce further knowledge source and blackboard trigger activations.

This process will continue until the final state is reached. We define this final state as the one in which there are no more KSAR's queued in any of the agendas, and all evaluations are finished.

StoppedABBSystem

ABBSystem

$\forall p : Desks \bullet p.DeskAgenda = \varnothing \land p.Eval = \langle\rangle$

$\forall k : KSources \bullet k.KSagenda = \varnothing$

$\forall s : LevelsOf(BBoard) \bullet s.Eval = \varnothing$

This final state can be reached naturally by the system, by exhausting all possible solution paths, or it can be forced by a knowledge source or blackboard trigger, when it identifies the domain problem termination condition.

E.7 Global System Operations

The ABB system defines a number of operations on the system as a whole that are available to the user. These operations are used to form the knowledge source body procedures, as well as blackboard triggers. These operations implement the different blackboard manipulation functions, as well as a global stop function.

The general procedure is to modify the blackboard using the appropriate level operation, obtain the ksars generated and add them to the appropriate knowledge source agenda. The guard checking, and trigger generation is performed by the level operation. In order to do this, the level operation needs information about the blackboard system element that originated the operation. The name of this element can be obtained via the *LocName* function. Given an operation and the system, this function returns the name of the element where it appears.

$$LocName : ModifyOp \times ABBSsystem \rightarrow PATH$$

$$\begin{array}{l}
\forall \odot : ModifyOp;\ S : ABBSystem \bullet \exists\, p : PATH \mid LocName(\odot, S) = p \\
\quad (\exists\, s : LevelsOf(S.BBoard) \mid \odot \in S.Eval \bullet p = S.Name) \lor \\
\quad ((\forall\, s : LevelsOf(S.BBoard) \bullet \odot \notin s.Eval) \land \\
\quad\ (\exists\, E : S.Desks \mid \odot = first\ E.Eval \bullet p = E.AcualKS)) \lor \\
\quad ((\forall\, s : LevelsOf(S.BBoard) \bullet \odot \notin s.Eval) \land \\
\quad\ (\forall\, E : S.Desks \mid \odot \neq first\ E.Eval) \land (p = NullPATH))
\end{array}$$

Once the blackboard has been modified, there will be a number of knowledge sources that trigger by those changes. In order to find the set of ksars generated by the modifications, we begin with the set of activation operations interested in a given blackboard level (*KSTriggers*). Whenever a level changes, this is the set of activations that must be evaluated for a given knowledge source.

$$KsTriggers : PATH \times KS\, \mathbb{F}\ ActivationOp$$

$$\begin{array}{l}
\forall\, p : PATH;\ K : KS \bullet \\
\quad KsTriggers(p, K) = \{\ \triangle : ActivationOp \mid \exists\, C : K.Trigger.Conditions \mid \\
\qquad C.BBlevel = p \land \triangle = C.Test\ \}
\end{array}$$

The set of ksars generated by a modification to a given blackboard level is the result of the evaluation of all *KsTriggers* for that level, filtered by the level's control function. Let p be the path of the modified level, Ks the set of knowledge sources of the system, and B the modified blackboard, *BBActivations* is the set of ksars generated by the modification.

$$BBActivations : PATH \times \mathbb{F}\ KS \times BlackBoard\, \mathbb{F}\ KSAR$$

$$\begin{array}{l}
\forall\, p : PATH;\ Ks : \mathbb{F}\ KS;\ B : BlackBoard \bullet \\
\quad ((\forall\, s : LevelsOf(B) \bullet s.Name \neq p) \land \\
\quad\ BBActivations(p, Ks, B) = \varnothing) \lor \\
\quad (\exists\, s : LevelsOf(B) \mid s.Name = p \bullet \\
\quad\ \exists\, A : \mathbb{F}\ KSAR \mid \\
\qquad A = \{\ k : KSAR \mid \exists\, K : Ks \mid \exists \triangle : KsTriggers(p, K) \mid \\
\qquad\quad k = \triangle(S.BBoard)\ \} \bullet \\
\qquad BBActivations(p, Ks, B) = s.BBCf(A))
\end{array}$$

Finally, the set of activation records for a particular knowledge source, is formed

by all the ksars that belong to it, generated in all the modified levels. Let K be the given knowledge source, Ks the set of knowledge sources of the system, and B and B' the blackboard before and after the modification. *KsActivations* is the set of ksars of K generated.

$$KsActivations : KS \times \mathbb{F}\, KS \times BlackBoard \times BlackBoard \rightarrow \mathbb{F}\, KSAR$$

$$\begin{array}{l}
\forall K : KS;\ Ks : \mathbb{F}\, KS;\ B : BlackBoard;\ B' : BlackBoard \bullet \\
\quad ((K \notin Ks) \wedge KsActivations(K, Ks, B, B') = \varnothing) \vee \\
\quad ((K \in Ks) \wedge \exists A : \mathbb{F}\, KSAR \mid A = KsActivations(K, Ks, B, B') \bullet \\
\qquad A = \{\ k : KSAR \mid \exists p : ModifiedLevels(B, B') \mid \\
\qquad\qquad k \in BBActivations(p, Ks, B') \wedge \\
\qquad\qquad k.KsInfo \in InfoOf(K.Name)\ \}
\end{array}$$

There will be five global system operations available to the user. These are *BBCreateOp*, *BBModifyOp*, *BBReadOp*, *BBDeleteOp*, and *BBStopOp*. The first one allows the creation of new blackboard elements. All *BBCreateOp* will have exactly one set of paths associated. These paths indicate where the new bbels must be created.

$$BBCreateOp : ModifyOp$$

$$\begin{array}{l}
\forall \odot : BBCreateOp \bullet \exists_1 P : \mathbb{F}\, PATH \bullet \\
\ \ \forall S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet \\
\quad \exists i : INFO \mid i = InfoOf(LocName(\odot, S)) \bullet \\
\quad\ \ S'.BBoard = LCreateBBels(i, P, S.BBoard) \wedge \\
\quad\ \ S'.KSources = \{\ K : S.KSources \bullet \\
\qquad\quad AddKSARs(K, KsActivations(K, S.BBoard, S'.BBoard))\ \} \wedge \\
\quad\ \ S'.Desks = S.Desks
\end{array}$$

The *BBModifyOp* operation allows the modification of blackboard elements already in the blackboard. All of these operations will have exactly one set of associated bbels. These bbels represent the modifications. Their structure is defined by the level bbel modification function (*LModifyBBels*). This is, they must have names of bbels already present in the blackboard, and their fields must be a subset of those bbels they modify.

$BBModifyOp : ModifyOp$

$$
\begin{array}{l}
\forall \odot : BBModifyOp \bullet \exists_1 B : \mathbb{F}\, BBel \bullet \\
\quad \forall S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet \\
\qquad \exists i : INFO \mid i = InfoOf(LocName(\odot, S)) \bullet \\
\qquad\quad S'.BBoard = LModifyBBels(i, B, S.BBoard) \wedge \\
\qquad\quad S'.KSources = \{\ K : S.KSources \bullet \\
\qquad\qquad\quad AddKSARs(K, KsActivations(K, S.BBoard, S'.BBoard))\ \} \wedge \\
\qquad\quad S'.Desks = S.Desks)
\end{array}
$$

The third operation permits the access to the information held in the blackboard. This operation does not change the contents of the blackboard. It does, however, change the state of the system. It may influence the procedure where it originated by changing some operations, or simply instantiating relevant information. This means that usually there is some sort of choice made, based on the result of the access. This choice can be seen as a function that takes such a result (a set of *BBels*) and a procedure, and returns the modified procedure.

$$Choice == \mathbb{F}\, BBel \times Procedure \rightarrow Procedure$$

All blackboard access operation (*BBReadOp*) will have a specific set of paths associated that indicate the bbels it will look for. It will also have a particular choice attached. As a result of the access, the system will be unchanged, except for the desk where the operation originated. The procedure being evaluated within that desk will be changed by the choice according to the result of the access to the blackboard.

$BBReadOp : ModifyOp$

$$
\begin{array}{l}
\forall \odot : BBReadOp \bullet \exists_1 P : \mathbb{F}\, PATH;\ C : Choice \bullet \\
\quad \forall S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet \\
\qquad S' \setminus Desks = S \setminus Desks \wedge \\
\qquad (((\forall E : S.esk \bullet \odot \neq first\ E.Eval) \wedge \\
\qquad\qquad S'.Desks = S.Desks) \vee \\
\qquad\ (\exists E : S.Desks \mid \odot = first\ E.Eval \bullet \\
\qquad\qquad (\exists i : INFO \mid i = InfoOf(E.ActualKS) \bullet \\
\qquad\qquad\ S'.Desks = S.Desks \setminus \{\ E\ \} \cup \\
\qquad\qquad\qquad \{\ E' : Desk \mid E' \setminus Eval = E \setminus Eval \wedge \\
\qquad\qquad\qquad\ E'.Eval = C(LReadBBels(i, P, S.BBoard), E.Eval)\ \})))
\end{array}
$$

The next operation allows the elimination of information held in the blackboard. For each one of these operations there is one particular set of paths referencing to the bbels that will be removed by it. The resulting system will have the given bbels removed (if the guards allow it). There may be a number of knowledge sources that are activated by these changes. The resulting knowledge sources will have the ksars generated added to their respective agendas.

$$
\begin{array}{l}
BBDeleteOp : ModifyOp \\
\hline
\forall \odot : BBDeleteOp \bullet \exists_1 P : \mathbb{F}\, PATH \bullet \\
\quad \forall S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet \\
\quad\quad (\exists i : INFO \mid i = InfoOf(LocName(\odot, S)) \bullet \\
\quad\quad\quad S'.KSources = \{\ K : S.KSources \bullet \\
\quad\quad\quad\quad AddKSARs(K, KsActivations(K, S.BBoard, S'.BBoard))\ \} \wedge \\
\quad\quad\quad S'.BBoard = LDeleteBBels(i, P, S.BBoard) \wedge \\
\quad\quad\quad S'.Desks = S.Desks)
\end{array}
$$

The last user operation allows to stop the system at any moment. Usually, this will be done whenever the termination condition is detected by a knowledge source body or blackboard trigger operation. This condition will be met whenever the problem domain is solved, or it is known that no solution can be found.

$$
\begin{array}{l}
BBStopOp : ModifyOp \\
\hline
\forall \odot : BBStopOp;\ S : ABBSystem \bullet \exists S' : StoppedABBSystem \mid S' = \odot(S) \bullet \\
\quad EqStaticStruct(S'.BBoard, S.BBoard) \wedge \\
\quad (\forall K' : S'.KSources \bullet \exists K : S.KSources \bullet \\
\quad\quad K' \setminus KSAgenda = K \setminus KSAgenda) \wedge \\
\quad (\forall E' : S.Desks \bullet \exists E : S.Desks \bullet \\
\quad\quad E' \setminus (Eval, DeskAgenda) = E \setminus (Eval, DeskAgenda))
\end{array}
$$

We can now define the set of operations available to the user as the union of the operations described above.

$$
\begin{array}{l}
UserOp == BBCreateOp \cup BBModifyOp \cup BBReadOp \\
\qquad\qquad \cup\ BBDeleteOp \cup BBStopOp
\end{array}
$$

In all ABB blackboard systems, the knowledge source bodies will be formed only by user operations. The blackboard trigger operations will be any user operations,

except read operations. This is because the blackboard triggers are not procedures, and the result of such access would be lost.

$$\forall S : ABBSystem \bullet$$
$$\quad \forall s : LevelsOf(S.BBoard) \bullet \operatorname{ran} s.Trigger \subset UserOp \setminus BBReadOp \land$$
$$\quad \forall K : S.KSources \bullet K.Body \in \operatorname{seq} UserOp$$

We now have the functional specification of the different elements of the active blackboard architecture, and of the operations between them. These operations are stored in different parts of the system, and when they are evaluated, produce changes to the general system state.

However, we still need global system operations that actually evaluate the different user operations in the different levels of the system. We will need four such operations.

The *ExecBBOp* one evaluates user operations held in the blackboard. These operations are the result of triggers activated by any of the blackboard modification operations. It does nothing if there are no user operations anywhere on the blackboard. If there is at least one operation, then it evaluates it. The new system will be the actual system with the operation applied to it, and with the operation taken out of the evaluation list.

ExecBBOp : *ModifyOp*

$$\forall \odot : ExecBBOp;\ S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet$$
$$\quad ((\forall s : LevelsOf(S.BBoard) \bullet s.Eval = \varnothing) \land S' = S) \lor$$
$$\quad (\exists s : LevelsOf(S.BBoard) \mid s.Eval \neq \varnothing \bullet$$
$$\qquad \exists \odot' : S.Eval \bullet$$
$$\qquad\quad \exists s' : LevelsOf(\odot'(S).BBoard) \mid s'.Name = s.Name \bullet$$
$$\qquad\qquad S' \setminus BBoard = S \setminus BBoard \land$$
$$\qquad\qquad EqStaticStruct(S'.BBoard, S.BBoard) \land$$
$$\qquad\qquad LevelsOf(S'.BBoard) = LevelsOf(\odot'(S).BBoard) \setminus \{\ s'\ \}$$
$$\qquad\qquad\qquad \cup \{\ s'' : LevelStruct \mid s'' \setminus Eval = s' \setminus Eval \land$$
$$\qquad\qquad\qquad\quad s''.Eval = s'.Eval \setminus \{\ \odot'\ \}\ \})$$

The second operation selects and prepares a knowledge source body to be evaluated. This operation will do nothing if there are no free desks. If there is at least one free desk, select one and choose one KSAR within its agenda, using its control

function. Take the body of the associated knowledge source, instanciate it with the chosen KSAR and place it in the evaluation list.

$$
\begin{array}{l}
\mathit{DeskprepareOp} : \mathit{ModifyOp} \\
\hline
\forall \odot : \mathit{DeskprepareOp};\ S : \mathit{ABBSystem} \bullet \exists S' : \mathit{ABBSystem} \mid S' = \odot(S) \bullet \\
\quad ((\forall E : S.\mathit{Desks} \bullet E.\mathit{Eval} \neq \langle\rangle) \land S' = S) \lor \\
\quad (\exists E : S.\mathit{Desks} \mid E.\mathit{Eval} = \langle\rangle \bullet \\
\quad\; (E.\mathit{DeskAgenda} = \varnothing \land S' = S) \lor \\
\quad\; (\exists k : \mathit{KSAR} \mid k = E.\mathit{DeskCf}(E.\mathit{DeskAgenda}) \bullet \\
\quad\quad \exists K : S.\mathit{KSources} \mid k.\mathit{KsInfo} \in \mathit{InfoOf}(K.\mathit{Name}) \bullet \\
\quad\quad\; S' \setminus \mathit{Desks} = S \setminus \mathit{Desks} \land \\
\quad\quad\; S'.\mathit{Desks} = S.\mathit{Desks} \setminus \{\ E\ \} \cup \\
\quad\quad\quad \{\ E' : \mathit{Desk} \mid E' \setminus (\mathit{DeskAgenda}, \mathit{Eval}, \mathit{ActualKS}) = \\
\quad\quad\quad\quad\quad\quad\quad\quad E \setminus (\mathit{DeskAgenda}, \mathit{Eval}, \mathit{ActualKS}) \land \\
\quad\quad\quad\quad E'.\mathit{DeskAgenda} = E.\mathit{DeskAgenda} \setminus \{\ k\ \} \land \\
\quad\quad\quad\quad E'.\mathit{Eval} = K.\mathit{Body}(k) \land \\
\quad\quad\quad\quad E'.\mathit{ActualKS} = K.\mathit{Name}\ \}))
\end{array}
$$

The third operation evaluates operations held in the desks, product of the previous operation. This operation will do nothing if there are no user operations waiting to be evaluated in any desk. If there is one such operation, it evaluates it. The new system will be the actual system with the operation applied to it, and with the operation taken out of the evaluation list.

$$
\begin{array}{l}
\mathit{ExecDeskOp} : \mathit{ModifyOp} \\
\hline
\forall \odot : \mathit{ExecDeskOp};\ S : \mathit{ABBSystem} \bullet \exists S' : \mathit{ABBSystem} \mid S' = \odot(S) \bullet \\
\quad ((\forall E : S.\mathit{Desks} \bullet E.\mathit{Eval} = \langle\rangle) \land S' = S) \lor \\
\quad (\exists E : S.\mathit{Desks};\ \odot' : \mathit{ModifyOp} \mid \odot' = \mathit{first}\ E.\mathit{Eval} \bullet \\
\quad\; \exists E' : \odot'(S).\mathit{Desks} \mid E'.\mathit{Name} = E.\mathit{Name} \bullet \\
\quad\quad S' \setminus \mathit{Desks} = \odot'(S) \setminus \mathit{Desks} \land \\
\quad\quad S'.\mathit{Desks} = \odot'(S).\mathit{Desks} \setminus \{\ E\ \} \cup \\
\quad\quad\quad\quad \{\ E'' : \mathit{Desk} \mid E'' \setminus \mathit{Eval} = E' \setminus \mathit{Eval} \land \\
\quad\quad\quad\quad\quad\quad E''.\mathit{Eval} = \mathit{last}\ E'.\mathit{Eval}\ \})
\end{array}
$$

Finally, the forth operation schedules KSAR's from the knowledge source agendas to the appropriate desk agenda. The other movements of KSAR's between the different

agendas is already performed by the blackboard user operations. This operation will do nothing if there is no KSAR's queued in any of the knowledge source agendas. If there is at least one non-empty agenda, it will choose a subset of its KSAR's using the knowledge source control function, and will place them within the agenda of its associated desk.

$$MoveKSARsOp : ModifyOp$$

$$
\begin{array}{l}
\forall \odot : MoveKSARsOp;\ S : ABBSystem \bullet \exists S' : ABBSystem \mid S' = \odot(S) \bullet \\
\quad ((\forall K : S.KSources \bullet K.KSAgenda = \varnothing) \wedge S' = S) \vee \\
\quad (\exists K : S.KSources \mid K.KSAgenda \neq \varnothing \bullet \\
\quad\ \exists E : S.Desks \mid k.Name \in E.KSnames \bullet \\
\quad\quad \exists A : \mathbb{F}\, KSAR \mid A \subseteq K.KSCf(K.KSAgenda) \bullet \\
\quad\quad\ S'.BBoard = S.BBoard \wedge \\
\quad\quad\ S'.KSources = S.KSources \setminus \{\ K\ \} \cup \\
\quad\quad\quad\quad \{\ K' : KS \mid K' \setminus KSAgenda = K \setminus KSAgenda \wedge \\
\quad\quad\quad\quad\ K'.KSAgenda = K.KSAgenda \setminus A\ \} \\
\quad\quad\ S'.Desks = S.Desks \setminus \{\ E\ \} \cup \\
\quad\quad\quad\quad \{\ E' : Desk \mid E' \setminus DeskAgenda = E \setminus DeskAgenda \wedge \\
\quad\quad\quad\quad\ E'.DeskAgenda = E.DeskAgenda \cup A\ \})
\end{array}
$$

The global system operations are, then, the union of those four operations.

$$ABBSystemOp = ExecBBOp \cup ExecDeskOp \cup DeskprepareOp \cup MoveKSAROp$$

E.8 ABB System Operation

The global system operation, then, results in the application of successive *ABBSystemOp*'s. We can define a transition function $\rightharpoondown$ that relates one system state with the one resulting from the application of a given system operation.

$$\rightharpoondown : ABBSystem \times ABBSystemOp \times ABBSystem$$

$$
\begin{array}{l}
\forall S, S' : ABBSystem;\ \odot : ABBSystemOp \bullet \\
\quad S \overset{\odot}{\rightharpoondown} S' \Leftrightarrow S' = \odot(S)
\end{array}
$$

By convention, we will write $S \overset{\odot_1}{\rightharpoondown} S' \overset{\odot_2}{\rightharpoondown} S''$, meaning $S \overset{\odot_1}{\rightharpoondown} S' \wedge S' \overset{\odot_2}{\rightharpoondown} S''$. In general,

$$S \xrightarrow{\odot_1} \ldots \xrightarrow{\odot_n} S_n \Leftrightarrow S \xrightarrow{\odot_1} S_1 \wedge \ldots \wedge S_{n-1} \xrightarrow{\odot_n} S_n$$

Let S be any blackboard system state. This state should be the result of applying several *ABBSystemOp*'s to a corresponding initial state. Let S_i be it's initial state. S will be a consistent state of the system if there is a sequence of *ABBSystemOp*'s that, after being applied, result in S.

Coherent : *ABBSystem*
$\forall S : ABBSystem \bullet \exists S_i : InitialABBSystem \mid S_i = InitialSystem(S) \bullet$ $Coherent(S) \Leftrightarrow (\exists \odot_1, \ldots, \odot_n : ABBSystemOp \bullet S_i \xrightarrow{\odot_1} \ldots \xrightarrow{\odot_n} S$

This property says that a given system state is coherent if there exists at least one sequential ordering of its operations with which we can reach it from the system's initial state. This is, it forces some sort of serialisation of its operations.

A system may be in a incoherent state when there is a modification of the blackboard and there are blackboard triggers or knowledge source activation procedures that fail to activate.

A system may also be incoherent if there were two consecutive blackboard modifications that were interleaved. For example, suppose two blackboard operations, $\odot_1$ and $\odot_2$, modify bbels b_1 and b_2. We will denote b_1^1, b_2^1, and b_1^2, b_2^2 the result of the $\odot_1$, and $\odot_2$ modifications, respectively. If we apply $\odot_1$, and then $\odot_2$, we will end up with b_1^2, and b_2^2. If we do it the other way around, we obtain b_1^2, and b_2^2.

Suppose, now, that we are working in a parallel environment, and that while $\odot_1$ is modifying b_1, $\odot_2$ is modifying b_2. Once they modify the first bbel they go and modify the other. When the modifications finish, we may end up with b_1^2, and b_2^1. This combination is impossible to duplicate using $\odot_1$ and $\odot_2$ sequentially.

A given implementation of the ABB specification must provide the functionality described here, and must comply with its properties. This includes the coherence property. The implementation must insure that it does not fall in an incoherent state. The way this is achieved is irrelevant to the specification. Note that the implementation could be sequential or parallel. Those considerations are purely implementation issues.

Appendix F

ABB Communication Mechanism

During the implementation of the ABB prototype, we used the CSTools message passing routines from the Meiko computing surface. In order to generate a system that could be ported to other platforms, we decided to implement a very small and simple set of message passing routines that will be sufficient for the ABB prototype implementation. The number of those routines should be very small, and yet allow multiple channel asynchronous communication. We call them the *SAMP* (Simple Asynchronous Message Passing) library.

The SAMP library is built by eight procedures that could be grouped in four categories. The first one is formed by global system operations for initialising (`samp_init`) and exiting gracefully from the system (`samp_exit`). The `samp_init` routine receives, as its arguments, pointers to the usual C main procedure arguments.

```
STATUS samp_init(int * argc, char *** argv);
```

The `samp_exit` routine receives an integer that will be used as the program return code.

```
STATUS samp_exit(int code);
```

Any given process, using the SAMP library, can access any number of communication channels. The two following routines, `samp_open` and `samp_close`, allow for the opening and closing of those channels. This is done in a fashion very similar to opening and closing of external files.

```
CHANNEL samp_open(char * chan_name, char * mode);
```

The open operation will receive a name for the channel, and a mode that can be either `"r"` (read) or `"w"` (write). The SAMP channels are unidirectional. The name given in the open operation must identify a channel univocally. A given channel can be opened for reading by only one process. When a process opens a channel for writing, this channel must be defined already by some other process for reading. If it does not exist, the procedure will block the process until it becomes available. This provides a first level of synchronisation, however, note that this may produce a deadlock. All processes using the SAMP library must open all reading channels before any writing channel. All communication channels are connected when they are opened.

The SAMP library also provides the `samp_close` routine for closing a particular channel. All the messages written on a closed channel will disappear.

```
STATUS samp_close(CHANNEL chan);
```

The next two procedures provide the basic message passing primitives. If **tochan** is open for writing, then we can use **samp_write** to write into it **datasz** number of bytes from the **data** area. This communication primitive is synchronous. This is, it will block until the receiving process is ready to read the message.

```
STATUS samp_write(CHANNEL tochan, size_t datasz, char * data);
```

If **mychan** is a channel open for reading, then we can use the **samp_read** routine to read a message from it. One should specify, besides the input channel, a pointer to the area where the message should be stored. The procedure will return the number of bytes read, or an error code in case of failure. This procedure implements the synchronous read function. If there is no message available, the operation will block the process until it arrives.

```
size_t samp_read(CHANNEL mychan, char * data);
```

The next two routines provide the SAMP library the capability of performing some asynchronous communication. The SAMP library allows the queuing of input buffers through the use of the **samp_qbuf** procedure. These buffers will be used to store incoming messages until the reader process decides to retrieve it. If a buffer is available, the reader process will be always ready for receiving a message. This means that the writing process can send the message without waiting for the reader process to be completely ready. There is still some level of synchronisation, but this is minimal. Also, the reader process can perform other tasks while waiting for a message, and can receive messages while it is doing some other work.

```
STATUS samp_qbuf(CHANNEL mychan, size_t datasz, char * data);
```

The other procedure in the group is the **samp_test** routine. It allows for questioning the input message queue in an non-blocking fashion. The status it returns will indicate if a message was received, or not. When there is one message waiting to be received, the **samp_test** routine will return the pointer to it in its **data** argument. This **data** argument can be **NULL**, in which case **samp_test** will only return if there was or not a message received, without retrieving it. It can be referencing a data pointer with the value **NULL**, in which case it will return the pointer to the received message. Alternatively, it can be pointing to defined memory area. This area is supposed to be a previous buffer. In this case, the routine will return a status indicating that a message was received in the specified buffer, or not.

```
STATUS samp_test(CHANNEL mychan, char ** data);
```

Usually, communication between processes will involve some interaction. One process sends a message, and it knows that it must receive some sort of answer. These processes can then set up enough buffers for receiving the answer and continue with some other task. The answer can then arrive, regardless of whether the process is ready to receive it or not.

The previous routines do have some influence from the type of facilities CSTools provides. This made the implementation of the above routines under CSTools very simple, adding almost no delays to it. However, some of the routine features, such as the possibility of having several communication channels per process, are not present in some other vendor's libraries, and would have to be inplemented.

Any other behaviour, except the one described here, will surely be inherited from the vendor transport layer chosen for SAMP implementation. For example, CSTools does not guarantee the order of messages within a channel. It we send message A and the message B, it is not guaranteed that A arrives before B. We ask the SAMP user not to make any assumptions about the library behaviour, except for the one presented here.

We implemented two versions of the SAMP library. One used CSTools as the vendor transport layer, the other was a sequential version of it. This sequential SAMP was implemented using C lists of structures, and allows the generation of one executable program with all the processes linked together. The inter-process communication is simulated by a sort of co-routining based on the design of the ABB processes. As we described earlier, all ABB processes are command-driven. They wait for a command, they process it and they wait for another command. This "waiting" is performed by calling the **samp_test** routine. The sequential version keeps a list of all the "processes", allowing the **samp_test** routine to cycle through them. Communication is done by merely moving data pointers to the different channel queues.

The sequential SAMP library allowed us to generate sequential ABB prototypes, without using external communications, in a variety of computer systems, such as Sun Sparc, Sequent, and Macintosh.

Appendix G

Jigsaw-Puzzle Solver, ABB Implementation

```
Oct 19 19:01 1994  BB-Lvl.ac Page 1

/*********************************************************/
/* Blackboard Method Definition.                         */
/* LEVEL: BB                                             */
/*********************************************************/

/*>
 * GetName Method:
 * Return the namestring of the given BBel.
 */
BOOL BB::
     GetName(BB::BBEL * self, char * name)
{
     strcpy(name,self->name);
     return(TRUE);
}

/*>
 * NameInit Method:
 * BB Initialization Method. Initializes the name to null.
 */
BOOL BB::
     NameInit(BB::BBEL * self)
{
     strcpy(self->name, self->bbelPH.name);

     return(TRUE);
}

/*>
 * NotifyAll Method:
 * Default Scheduling function. This may be overridden by
 * sublevel scheduling functions.
 */
BOOL BB::
     NotifyAll(BB::BBEL * self, LIST * BBelAgenda, LIST ** ToNotify)
{
     *ToNotify = BBelAgenda;
     return(TRUE);
}
```

```
Oct 19 19:01 1994  BB.JSP-Lvl.ac Page 1

/*********************************************************/
/* Blackboard Method Definition.                         */
/* LEVEL: BB.JSP                                         */
/*********************************************************/

/*>
 * setShp Method:
 * Modifies the given shape.
 */
BOOL BB.JSP::
     setShp(BB.JSP::BBEL * self, int * which, int * value)
{
     self->shape[*which] = *value;
     return(TRUE);
}

/*>
 * getShp Method:
 * Returns a particular side's shape.
 */
BOOL BB.JSP::
     getShp(BB.JSP::BBEL * self, int * which, int * value)
{
     *value = self->shape[*which];
     return(TRUE);
}

/*>
 * initShape Method:
 * Initializes the shape field for all new BB.JSP bbels
 * (or bbels on its sublevels).
 */
BOOL BB.JSP::
     initShape(BB.JSP::BBEL * self)
{
     self->shape[0] = 0;
     self->shape[1] = 0;
     self->shape[2] = 0;
     self->shape[3] = 0;
     return(TRUE);
}
```

```
Oct 19 19:01 1994  BB-Lvl.ah Page 1

/*********************************************************/
/* Blackboard Structure Definition.                      */
/* LEVEL: BB                                             */
/*********************************************************/

/*>
 * Top Blackboard Level Structure.
 * All elements will have:
 *      name:   A name
 */
BBLVLdefine BB {       /* Define top Object Type */
     char        name[NAMELENGTH];

     BOOL        GetName(BB::BBEL * self, char ** name);
     BOOL        INIT::NameInit(BB::BBEL * self);
     BOOL        BBELCF::NotifyAll(BB::BBEL * self,
                               LIST * BBelAgenda, LIST ** ToNotify);
}
```

```
Oct 19 19:01 1994  BB.JSP-Lvl.ah Page 1

/*********************************************************/
/* Blackboard Structure Definition.                      */
/* LEVEL: BB.JSP                                         */
/*********************************************************/

/*>
 * BB.JSP Level Structure.
 * This is the game structure proper. Any JSP element
 * can be seen as a squared piece (or position) with a
 * given shape on each side.
 *
 * All elements will have:
 *      shape:  A shape array. One shape for each of the four sides.
 */
BBLVLdefine BB.JSP {
     int         shape[4];

     BOOL        setShp(BB.JSP::BBEL * self, int which, int value);
     BOOL        getShp(BB.JSP::BBEL * self, int which, int * value);
     BOOL        INIT::initShape(BB.JSP::BBEL * self);
}
```

```
Oct 19 19:01 1994  BB.JSP.Piece-Lvl.ac Page 1

/**********************************************************/
/* Blackboard Method Definition.                          */
/* LEVEL: BB.JSP.Piece                                    */
/**********************************************************/

/*>
 * setKsName Method:
 * Set the name of the KS that holds the Piece
 */
BOOL BB.JSP.Piece::
      setKsName(BB.JSP.Piece::BBEL * self, char * ksName)
{
      strcpy(self->ksName,ksName);
      return(TRUE);
}

/*>
 * place Method:
 * Set the piece status as placed.
 */
BOOL BB.JSP.Piece::
      place(BB.JSP.Piece::BBEL * self)
{
      self->placed = TRUE;
      return(TRUE);
}

/*>
 * remove Method:
 * Set the piece status as placed.
 */
BOOL BB.JSP.Piece::
      remove(BB.JSP.Piece::BBEL * self)
{
      self->placed = FALSE;
      return(TRUE);
}

/*>
 * notPlaced Method:
 * Test if the piece is already placed.
 */
BOOL BB.JSP.Piece::
      notPlaced(BB.JSP.Piece::BBEL * self)
{
      if (self->placed)
            return(FALSE)
      return(TRUE);
}

/*>
 * initPiece Method:
 * Initialise a Piece. (Set it to be Non-placed.)
 */
BOOL BB.JSP.Piece::
      initPiece(BB.JSP.Piece::BBEL * self)
```

```
Oct 19 19:01 1994  BB.JSP.Piece-Lvl.ah Page 1

/**********************************************************/
/* Blackboard Structure Definition.                       */
/* LEVEL: BB.JSP.Piece                                    */
/**********************************************************/

/*>
 * BB.JSP.Piece Level Structure.
 * This level will hold all JSP pieces. They will only
 * have an extra flag (placed) indicating if the piece is
 * already placed.
 */
BBLVLdefine BB.JSP.Piece {
      int         placed;
      char        ksName[NAMELEN];

      BOOL        setKsName(BB.JSP.Piece::BBEL * self, char * ksName);
      BOOL        place(BB.JSP.Piece::BBEL * self);
      BOOL        remove(BB.JSP.Piece::BBEL * self);
      BOOL        notPlaced(BB.JSP.Piece::BBEL * self);
      BOOL        INIT::initPiece(BB.JSP.Piece::BBEL * self);
}

/*
 * Define Level Instances.
 * Each instance will be a BBel store
 */
defineBBLVL BB.JSP.Piece Piece_Lvl_1;
defineBBLVL BB.JSP.Piece Piece_Lvl_2;
defineBBLVL BB.JSP.Piece Piece_Lvl_3;
```

```
Oct 19 19:01 1994  BB.JSP.Piece-Lvl.ac Page 2

{
      self->placed=FALSE;
      return(TRUE);
}
```

```
Oct 19 19:01 1994  BB.JSP.Position-Lvl.ac Page 1

/**********************************************************/
/* Blackboard Method Definition.                          */
/* LEVEL: BB.JSP.Position                                 */
/**********************************************************/

#include "BB.JSP.Position-Lvl.ah"
#include "jsp.h"

BOOL BB.JSP.Position::
      activate(BB.JSP.Position::BBEL * self)
{
      if (self->piece[0] == '\001')
            self->piece[0]='\000';
      return(TRUE);
}

BOOL BB.JSP.Position::
      disable(BB.JSP.Position::BBEL * self)
{
      if (self->piece[0] == '\000') {
            self->piece[0]='\001';
            self->piece[1]='\000';
      }
      return(TRUE);
}

BOOL BB.JSP.Position::
      isActive(BB.JSP.Position::BBEL * self)
{
      if (self->piece[0] == '\000')
            return(TRUE);
      return(FALSE);
}

BOOL BB.JSP.Position::
      match(BB.JSP.Position::BBEL * self, char * piece, int * ori)
{
      int t,r,b,l;
      int pieceSh[4];
      int i;
      int st;

      t=TOP;      r=RIGHT;
      b=BOTTOM;   l=LEFT;
      st = readSBele(1, self->PiecePath, piece,
                     "getShp(%d,%d),getShp(%d,%d),"
                     "getShp(%d,%d),getShp(%d,%d) ",
                      &t,&pieceSh[TOP], &r,&pieceSh[RIGHT],
                      &b,&pieceSh[BOTTOM], &l,&pieceSh[LEFT]);
      if (st != OK)
            return(FALSE);

      for (i=0;i<4; i++)
            if (self->matchShapes(self, &(self->shape[TOP]),
                                  &(pieceSh[ADJUST(TOP,i)])) &&
                self->matchShapes(self, &(self->shape[RIGHT]),
```

```
Oct 19 19:01 1994  BB_JSP_Position-Lvl.ac Page 2

                              &(pieceSh[ADJUST(RIGHT,1)])) &&
            self->matchShapes(self, &(self->shape[BOTTOM]),
                              &(pieceSh[ADJUST(BOTTOM,1)])) &&
            self->matchShapes(self, &(self->shape[LEFT]),
                              &(pieceSh[ADJUST(LEFT,1)]))  &&
            self->notOldPiece(self, piece, &i)) {
             *ori = i;
             return(TRUE);
        }
    return(FALSE);
}

BOOL BB_JSP_Position::
    matchShapes(BB_JSP_Position::BBEL * self,
                int * posSh, int * pieSh)
{
    if (*posSh == UNDEF) {
        if (*pieSh == BORDER)
            return(FALSE);
        return(TRUE);
    }

    if (*posSh == *pieSh)
        return(TRUE);
    return(FALSE);
}

BOOL BB_JSP_Position::
    notOldPiece(BB_JSP_Position::BBEL * self,
                char * piece, int * ori)
{
    LIST * tmpL = self->oldPieces;
    PieceOri * pce;

    while (tmpL != NULLlist) {
        pce = (PieceOri *) headLIST(tmpL);
        if ((pce->orientation == *ori) &&
            (strcmp(pce->piece, piece) == 0))
            return(FALSE);
        tmpL=tailLIST(tmpL);
    }
    return(TRUE);
}

BOOL BB_JSP_Position::
    setLnk(BB_JSP_Position::BBEL * self, int * which, char * position)
{
    strcpy(self->link[*which],position);
    return(TRUE);
}

BOOL BB_JSP_Position::
    getLnk(BB_JSP_Position::BBEL * self, int * which, char * position)
{
    strcpy(position, self->link[*which]);
    return(TRUE);
```

```
Oct 19 19:01 1994  BB_JSP_Position-Lvl.ac Page 3

}

BOOL BB_JSP_Position::
    updateShape(BB_JSP_Position::BBEL * self, int * which)
{
    int         st;
    int         which2 = COMPL(*which);

    if (self->shape[*which] == BORDER)
        return(TRUE);

    st = ReadBBele(1,
                self->PositPath, self->link[*which],
                    "getShp(%d,%d)",&which2, &(self->shape[*which]))
    if (st != OK)
        return(FALSE);
    return(TRUE);
}

BOOL BB_JSP_Position::
    modifyPosition(BB_JSP_Position::BBEL * self, int * which)
{
    modifL *    mdf;

    if ((mdf = malloc(sizeof(modifL))) == NULL)
        return(FALSE);
    mdf->which = *which;
    mdf->next = self->modifiedBy;
    self->modifiedBy = mdf;
    return(TRUE);
}

BOOL BB_JSP_Position::
    resetOld(BB_JSP_Position::BBEL * self)
{
    freeLIST(self->oldPieces);
    self->oldPieces = NULLlist;
    return(TRUE);
}

BOOL BB_JSP_Position::
    placePiece(BB_JSP_Position::BBEL * self, char * piece, int * ori)
{
    int         t=ADJUST(TOP,*ori);
    int         r=ADJUST(RIGHT,*ori);
    int         b=ADJUST(BOTTOM,*ori);
    int         l=ADJUST(LEFT,*ori);
    int         ts,rs,bs,ls;
    int         st;
    PieceOri * pce;

    if ((pce = malloc(sizeof(PieceOri))) == NULL)
        return(FALSE);

    st = ReadBBele(1, self->PiecePath, piece,
                        "getShp(%d,%d),getShp(%d,%d),"
```

```
Oct 19 19:01 1994  BB_JSP_Position-Lvl.ac Page 4

                        "getShp(%d,%d),getShp(%d,%d) ",
                         &t,&ts, &r,&rs, &b,&bs, &l,&ls);
    if (st != OK) {
        free(pce);
        return(FALSE);
    }

    self->shape[TOP]    = ts;
    self->shape[RIGHT]  = rs;
    self->shape[BOTTOM] = bs;
    self->shape[LEFT]   = ls;

    strcpy(self->piece,piece);

    strcpy(pce->piece,piece);
    pce->orientation = *ori;
    self->oldPieces = consLIST(sizeof(PieceOri),pce,self->oldPieces);
    return(TRUE);
}

BOOL BB_JSP_Position::
    getPiece(BB_JSP_Position::BBEL * self, char * piece)
{
    strcpy(piece,self->piece);
    return(TRUE);
}

BOOL BB_JSP_Position::
    removePiece(BB_JSP_Position::BBEL * self)
{
    char pce[NAMELEN];
    int  i;

    self->piece[0]='\000';
    for (i=0;i<4;i++)
        if (self->shape[i] != BORDER) {
            ReadBBele(1,
                self->PositPath, self->link[i],"getPiece(%s)",&pce);
            if ((pce[0] == '\000') || (pce[0] == '\001'))
                self->shape[i] = UNDEF;
        }
    return(TRUE);
}

BOOL BB_JSP_Position::
    getModifBy(BB_JSP_Position::BBEL * self, char * position)
{
    strcpy(position,self->link[self->modifiedBy->which]);
    return(TRUE);
}

BOOL BB_JSP_Position::
    removeModifBy(BB_JSP_Position::BBEL * self, int * which)
{
    modifL * tmpL1;
    modifL * tmpL2;
```

```
Oct 19 19:01 1994  BB_JSP_Position-Lvl.ac Page 5

    tmpL2=NULL;
    for(tmpL1=self->modifiedBy; tmpL1!=NULL; tmpL1=tmpL1->next) {
        if (*which == tmpL1->which) {
            if (tmpL2 == NULL) {
                self->modifiedBy=tmpL1->next;
                free(tmpL1);
            } else {
                tmpL2->next = tmpL1->next;
                free(tmpL1);
            }
            return(TRUE);
        }
        tmpL2 = tmpL1;
    }
    return(TRUE);
}

BOOL BB_JSP_Position::
    initPosition(BB_JSP_Position::BBEL * self)
{
    self->PositPath = idPATH("BB_JSP_Position");

    self->link[0][0]='\000';
    self->link[1][0]='\000';
    self->link[2][0]='\000';
    self->link[3][0]='\000';

    self->PiecePath = idPATH("BB_JSP_Piece");

    self->piece[0]='\000';
    self->Usable(self);

    self->modifiedBy = NULL;

    self->oldPieces = NULLlist;
    return(TRUE);
}

/*
 * Rate the given position.
 * Priority is given to those positions with most sides defined.
 * Also, those on the borders of the game board have also more
 * priority
 *
 * The rating is computed as follows:
 *       - Each defined side is valued as 10.
 *       - A bonus of 2 is given if the side is on the border.
 *
 * A piece with two sides defined will be given smaller rating
 * than a corner, say, but a corner with no other sides defined
 * will have a lower rating than a piece with three sides defined.
 */
BOOL BB_JSP_Position::
    ratePosition(BB_JSP_Position::BBEL * self, int * rating)
{
```

```
Oct 19 19:01 1994  BB.JSP.Position-Lvl.ac Page 6

	int i;

	*rating = 0;
	for (i=0;i<4;i++) {
		if (self->shape[i] != UNDEF)
			*rating += 10;
		if (self->shape[i] == BORDER)
			*rating += 2;
	}
	return(TRUE);
}

BOOL BB.JSP.Position::
	PreferPieceKSs(BB.JSP.Position::BBEL * self,
			LIST * BbelAgenda, LIST ** ToNotify)
{
	KSAR * ksar;
	LIST * tmpL1;
	LIST * tmpL2;
	int    rating;

	if (BbelAgenda == NULLlist) {
		*ToNotify = NULLlist;
		return(TRUE);
	}

	/*
	 * For any given modification to the blackboard, the Misplaced
	 * KS will trigger at most once. (Actually, it will trigger always
	 * once.) Therefore, if the agenda has more than one element then
	 * it will necessarily have a KSAR for a PieceKS. This means that
	 * the MisplacedKS should not trigger, and we take it out of the
	 * agenda.
	 */

	self->ratePosition(self,&rating);

	ksar = (KSAR *) headLIST(BbelAgenda);
	if (tailLIST(BbelAgenda) != NULLlist) {
		tmpL2=NULLlist;
		for (tmpL1=BbelAgenda; tmpL1!=NULLlist; tmpL1=tailLIST(tmpL1)) {
			ksar = (KSAR *) headLIST(tmpL1);
			ksar->rating = rating;
			if (ksar->ksName[0] == 'M')
				if (tmpL2 == NULLlist) {
					BbelAgenda = tailLIST(tmpL1);
				} else
					addToLIST(tmpL2,tailLIST(tmpL1));
		}
	}
	*ToNotify = BbelAgenda;
	return(TRUE);
}

/*
STATUS BB.JSP.Position::
```

```
Oct 19 19:01 1994  BB.JSP.Position-Lvl.ac Page 7

	checkShapes(BB.JSP.Position::BBEL * self, CMD * command)
{

}

BOOL BB.JSP.Position::
	unifyShapes(BB.JSP.Position::BBEL * self, CMD * command)
{

}
*/
```

```
Oct 19 19:01 1994  BB.JSP.Position-Lvl.ah Page 1

/*******************************************************/
/* Blackboard Structure Definition.                    */
/* LEVEL: BB.JS.Position                               */
/*******************************************************/

#include "jsp.h"

/**
 * BB.JSP.Position Level Structure.
 * This level will hold the JSP positions.
 * A position will represent the place where pieces will
 * be placed. Initially all positions will be empty and
 * their shapes will be undefined. (Except for the ones
 * on the sides; these will have one or more shapes defined.)
 *
 * Each position will have the following fields:
 *	link:		References to all the surrounding positions.
 *			These are needed as adjacent shapes must be
 *			maintained equal.
 *	piece:		Reference to the piece occupying the position
 *	orientation:	The orientation of the occupying piece.
 */
BBLVLdefine BB.JSP.Position {
	PATH		PositPath;
	char		link[NAMELEN][4];
	PATH		PiecePath;
	char		piece[NAMELEN];
	M-III *		modifiedBy;
	LIST *		oldPieces;

	BOOL		activate(BB.JSP.Position::BBEL * self);
	BOOL		disable(BB.JSP.Position::BBEL * self);
	BOOL		isActive(BB.JSP.Position::BBEL * self);
	BOOL		match(BB.JSP.Position::BBEL * self,
				char * piece, int * ori);
	BOOL		matchShapes(BB.JSP.Position::BBEL * self,
				int * posSh, int * pieSh);
	BOOL		notOldPiece(BB.JSP.Position::BBEL * self,
				char * piece, int * ori);
	BOOL		setLnk(BB.JSP.Position::BBEL * self,
				int which, char * position);
	BOOL		getLnk(BB.JSP.Position::BBEL * self,
				int * which, char * position);
	BOOL		updateShape(BB.JSP.Position::BBEL * self, int which);
	BOOL		modifyPosition(BB.JSP.Position::BBEL * self, int which);
	BOOL		ratePosition(BB.JSP.Position::BBEL * self, int * which);
	BOOL		placePiece(BB.JSP.Position::BBEL * self,
				char * piece, int ori);
	BOOL		getPiece(BB.JSP.Position::BBEL * self, char * piece);
	BOOL		removePiece(BB.JSP.Position::BBEL * self);
	BOOL		resetOld(BB.JSP.Position::BBEL * self);
	BOOL		getModifBy(BB.JSP.Position::BBEL * self, char * position);
	BOOL		removeModifBy(BB.JSP.Position::BBEL * self, int which);
	BOOL		NEW::initPosition(BB.JSP.Position::BBEL * self);
	BOOL		BBELCF::PreferPieceKSs(BB.JSP.Position::BBEL * self,
					LIST * BbelAgenda, LIST ** ToNotify);
```

```
Oct 19 19:01 1994  BB.JSP.Position-Lvl.ah Page 2

}

/*
 * Define level instances.
 * Each instance will be a Bbel store.
 */
-defineBBLVL BB.JSP.Position Position_Lvl_1;
-defineBBLVL BB.JSP.Position Position_Lvl_2;
-defineBBLVL BB.JSP.Position Position_Lvl_3;
```

```
Oct 19 19:01 1994  BB.KS-Lvl.ac Page 1

/*********************************************************/
/* Blackboard Method Definition.                         */
/* LEVEL: BB.KS                                          */
/*********************************************************/

#include "BB.KS-Lvl.ah"

/*>
 * include Method:
 */
BOOL BB.KS::
     include(BB.KS::BBEL * self, char * pieceName)
{
     char * tmp;

     if ((tmp = malloc((size_t) NAMELEN)) == NULL)
          return(FALSE);
     self->Pieces = consLIST((size_t) NAMELEN, tmp, self->Pieces);
     return(TRUE);
}

/*>
 * place Method:
 */
BOOL BB.KS::
     place(BB.KS::BBEL * self, char * pieceName)
{
     LIST * tmplL;
     LIST * tmpL;

     for(tmplL=tmpL=self->Pieces; tmpL != NULLlist; tmpL=tailLIST(tmpL)) {
          if (strcmp((char *) headLIST(tmpL), pieceName) == 0) {
               tmplL->next=tmpL->next;
               self->Placed = addToLIST(tmpL,self->Placed);
               return(TRUE);
          }
          tmplL = tmpL;
     }
     return(FALSE);
}

/*>
 * remove Method:
 */
BOOL BB.KS::
     remove(BB.KS::BBEL * self, char * pieceName)
{
     LIST * tmplL;
     LIST * tmpL;

     tmplL = NULL;
     for(tmpL=self->Placed; tmpL != NULLlist; tmpL=tailLIST(tmpL)) {
          if (strcmp((char *) headLIST(tmpL), pieceName) == 0) {
               if (tmplL == NULL)
                    self->Placed = tmpL->next;
               else tmplL->next=tmpL->next;
```

```
Oct 19 19:01 1994  BB.KS-Lvl.ac Page 2

               self->Pieces = addToLIST(tmpL,self->Pieces);
               return(TRUE);
          }
          tmplL = tmpL;
     }
     return(FALSE);
}

/*>
 * allPlaced Method:
 */
BOOL BB.KS::
     allPlaced(BB.KS::BBEL * self)
{
     if (self->Pieces == NULLlist)
          return(TRUE);
     return(FALSE);
}

/*>
 * getNumber Method:
 */
BOOL BB.KS::
     getPieces(BB.KS::BBEL * self, PieceArray * pieces, int * number)
{
     LIST * tmpL;

     *number = 0;
     for(tmpL=self->Pieces; tmpL != NULLlist; tmpL=tailLIST(tmpL)) {
          strcpy(pieces[*number], (char *)headLIST(tmpL));
          (*number)++;
     }
     return(TRUE);
}

/*>
 * initNumber Method:
 * Initializes the number of PieceKS's for all new
 * BB.KS bbels (or bbels on its sublevels).
 */
BOOL BB.KS::
     initKS(BB.KS::BBEL * self)
{
     self->Pieces = NULLlist;
     self->Placed = NULLlist;
     return(TRUE);
}
```

```
Oct 19 19:01 1994  BB.KS-Lvl.ah Page 1

/*********************************************************/
/* Blackboard Structure Definition.                      */
/* LEVEL: BB.KS                                          */
/*********************************************************/

/*>
 * BB.KS Level Structure.
 * This is the structure that stores the piece details
 * (their names) of a particular KS. The KS becomes
 * inactive when all its pieces have been placed.
 *
 */
BBLVLdefine BB.KS {
     LIST *     Pieces;
     LIST *     Placed;

     BOOL       include(BB.JSP::BBEL * self, char * pieceName);
     BOOL       place(BB.JSP::BBEL * self, char * pieceName);
     BOOL       remove(BB.JSP::BBEL * self, char * pieceName);
     BOOL       allPlaced(BB.JSP::BBEL * self);
     BOOL       getPieces(BB.JSP::BBEL * self,
                          PieceArray * pieces, int * number);
     BOOL       INIT::initKS(BB.JSP::BBEL * self);
}

/*
 * Define level instances.
 * Each instance will be a bbel store.
 */
defineBBLVL BB.KS KS_Lvl_1;
defineBBLVL BB.KS KS_Lvl_2;
```

```
Oct 19 19:01 1994  InitKS-KS.ac Page 1

/*********************************************************/
/* Knowledge Source Method Definition.                   */
/*********************************************************/

BOOL BB.JSP.KS::
     AllPiecesPlaced(BBEL::BB.JSP.Control * trigBBel,
                     InitKS::KSAR * self)
{
     int  st;
     PATH ksLvl;

     if (trigBBel->Pieces == NULLlist) {
          ksLvl = idPATH("BB.KS");
          st = ReadBBels(5, ksLvl, "PieceKS_1", "allPlaced()",
                            ksLvl, "PieceKS_2", "allPlaced()",
                            ksLvl, "PieceKS_3", "allPlaced()",
                            ksLvl, "PieceKS_4", "allPlaced()",
                            ksLvl, "PieceKS_5", "allPlaced()");
          if (st == OK)
               return(TRUE);
     }
     return(FALSE);
}

void InitKS::
     initFn(InitKS::KS * self)
{

}

void InitKS::
     AllKSAre(InitKS::KS * self,
              LIST ** agenda, LIST ** toSched)
{
     *toSched = *agenda;
     *agenda = NULLlist;
}

void InitKS::
     InitKS(InitKS::KS * self, InitKS::KSAR * ksar)
{
     PATH tmpPath;

     if (pathPATHNAME(&(ksar->bbel)) == NULLpath) {
          /*
           * Create and Initialize the KS's piece list.
           */
          tmpPath=idPATH("BB.KS");
          createBBels(5,
               tmpPath, "PieceKS_1", "include(%s).include(%s).include(%s)",
                    "1 1", "2 2", "3 3",
               tmpPath, "PieceKS_2", "include(%s).include(%s).include(%s)",
                    "1 2", "2 3", "3 4",
               tmpPath, "PieceKS_3", "include(%s).include(%s)",
                    "1 3", "2 4",
```

```
Oct 19 19:01 1994  InitKS-KS.ac Page 2

            tmpPath, "PieceKS_4", "include(%s).include(%s)",
                "1 4", "3 1",
            tmpPath, "PieceKS_5", "include(%s).include(%s)",
                "2 1", "3 2");

        /*
         * Create and Initialise all Pieces.
         */
        tmpPath = idPATH("BB.JSP.Piece");
        CreateBBels(4,
            tmpPath, "1 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP, BORDER, RIGHT,      3, BOTTOM,      5, LEFT, BORDER,
            tmpPath, "1 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP, BORDER, RIGHT,      3, BOTTOM,      3, LEFT,      3,
            tmpPath, "1 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP, BORDER, RIGHT,      4, BOTTOM,      1, LEFT,      1,
            tmpPath, "1 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP, BORDER, RIGHT, BORDER, BOTTOM ,     3, LEFT,      4);
        CreateBBels(4,
            tmpPath, "2 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                 TOP,      5, RIGHT,      2, BOTTOM,      1, LEFT, BORDER,
            tmpPath, "2 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                 TOP,      3, RIGHT,      2, BOTTOM,      3, LEFT,      2,
            tmpPath, "2 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                 TOP,      1, RIGHT,      4, BOTTOM,      3, LEFT,      2,
            tmpPath, "2 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP,       3, RIGHT, BORDER, BOTTOM,      5, LEFT,      4)
        CreateBBels(4,
            tmpPath, "3 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP,       1, RIGHT,      5, BOTTOM, BORDER, LEFT, BORDER,
            tmpPath, "3 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP,       3, RIGHT,      3, BOTTOM, BORDER, LEFT,      %,
            tmpPath, "3 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP,       3, RIGHT,      2, BOTTOM, BORDER, LEFT,      3,
            tmpPath, "3 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d)",
                TOP,       5, RIGHT, BORDER, BOTTOM, BORDER, LEFT,      2);

        /*
         * Create and Initialise all Positions.
         */
        tmpPath = idPATH("BB.JSP.Position");
        CreateBBels(4,
            tmpPath, "0 0",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
```

```
Oct 19 19:01 1994  InitKS-KS.ac Page 3

                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP, BORDER, RIGHT, BORDER, BOTTOM, BORDER, LEFT, BORDER,
                TOP,  "0 0", RIGHT,  "0 0", BOTTOM,  "0 0", LEFT,  "0 0");
        CreateBBels(4,
            tmpPath, "1 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP, BORDER, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT, BORDER,
                TOP,  "0 0", RIGHT,  "1 2", BOTTOM,  "2 1", LEFT,  "0 0",
            tmpPath, "1 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP, BORDER, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "0 0", RIGHT,  "1 3", BOTTOM,  "2 2", LEFT,  "1 1",
            tmpPath, "1 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP, BORDER, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "0 0", RIGHT,  "1 4", BOTTOM,  "2 3", LEFT,  "1 2",
            tmpPath, "1 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP, BORDER, RIGHT, BORDER, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "0 0", RIGHT,  "0 0", BOTTOM,  "2 4", LEFT,  "1 3");
        CreateBBels(4,
            tmpPath, "2 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT, BORDER,
                TOP,  "1 1", RIGHT,  "2 2", BOTTOM,  "3 1", LEFT,  "0 0",
            tmpPath, "2 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "1 2", RIGHT,  "2 3", BOTTOM,  "3 2", LEFT,  "2 1",
            tmpPath, "2 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "1 3", RIGHT,  "2 4", BOTTOM,  "3 3", LEFT,  "2 2",
            tmpPath, "2 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT, BORDER, BOTTOM,  UNDEF, LEFT,  UNDEF,
                TOP,  "1 4", RIGHT,  "0 0", BOTTOM,  "3 4", LEFT,  "2 3");
        CreateBBels(4,
            tmpPath, "3 1",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM, BORDER, LEFT, BORDER,
                TOP,  "2 1", RIGHT,  "3 2", BOTTOM,  "0 0", LEFT,  "0 0",
            tmpPath, "3 2",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM, BORDER, LEFT,  UNDEF,
                TOP,  "2 2", RIGHT,  "3 3", BOTTOM,  "0 0", LEFT,  "3 1",
```

```
Oct 19 19:01 1994  InitKS-KS.ac Page 4

            tmpPath, "3 3",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT,  UNDEF, BOTTOM, BORDER, LEFT,  UNDEF,
                TOP,  "2 3", RIGHT,  "3 4", BOTTOM,  "0 0", LEFT,  "3 2",
            tmpPath, "3 4",
                "setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),setShp(%d,%d),"
                "setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s),setLnk(%d,%s)",
                TOP,  UNDEF, RIGHT, BORDER, BOTTOM, BORDER, LEFT,  UNDEF,
                TOP,  "2 4", RIGHT,  "0 0", BOTTOM,  "0 0", LEFT,  "3 3");
        /*
         * Activate the four corners. This will initiate the system.
         */
        writeBBels(4,
            tmpPath, "1 1", "activate()",
            tmpPath, "1 4", "activate()",
            tmpPath, "3 1", "activate()",
            tmpPath, "3 4", "activate()");
    } else {
        StopSystem();
    }
}
```

```
Oct 19 19:01 1994  InitKS-KS.ah Page 1

/*********************************************************/
/* Knowledge Source Structure Definition.               */
/* KS: PieceKS                                          */
/*********************************************************/

KSdefine InitKS {

    void        INIT::initFn(InitKS::KS * self);
    void        KSCF::AllKSARs(InitKS::KS * self,
                        LIST ** agenda, LIST ** toShed);
    void        KSBODY::InitKS(InitKS::KS * self,
                        InitKS::KSAR * ksar);
};

TriggerDefine InitKS {
    SYSINIT;

    BOOL BB_KS::
            AllPiecesPlaced(BB_KS::BBEL * trigBBel,
                        InitKS::KSAR * self);
}

/*
 * Define KS instances
 */
DefineKS InitKS InitKS_1;
```

```
Oct 19 19:01 1994  JSP-Sys.sc Page 1

/*******************************************************/
/* JSP System Method Definitions.                      */
/*******************************************************/

/*>
 * JSP System initialisation function.
 * This function only shows the ability to use the system
 * internal structure.
 */
void SYS::
      SysInit(SYS * self)
{
      self->sys_var = 1;
}

/*>
 * This is the system-wide scheduling function.
 */
void SYS::
      Highest(SYS * self, LIST ** agenda, KSAR ** toRun)
{
      if (*agenda != NULLlist)
            *agenda = removeMaxKSAR(toRun, *agenda);
}
```

```
Oct 19 19:01 1994  JSP-Sys.sh Page 1

/*******************************************************/
/* JSP System Structure Definition.                    */
/*******************************************************/

#include "jsp.h"

/*>
 * JSP System Supervisor definition.
 *
 * Structure and methods.
 */
SYSdefine JSP {
      int   sys_var;

      void INIT::SysInit(SYS * self);
      void SYSCF::Highest(SYS * self, LIST ** agenda, KSAR ** toRun);
};

/*
 * Define System Supervisor instances.
 * Each will manage a group of KS instances.
 */
defineSYS JSP PE_1 (PieceKS_1);
defineSYS JSP PE_2 (PieceKS_2);
defineSYS JSP PE_3 (PieceKS_3);
defineSYS JSP PE_4 (PieceKS_4);
defineSYS JSP PE_5 (PieceKS_5);
defineSYS JSP PE_6 (InitKS_1, MisplacedKS_1);
```

```
Oct 19 19:01 1994  MisplacedKS-KS.sc Page 1

/*******************************************************/
/* Knowledge Source Method Definition.                 */
/*******************************************************/

void BB_JSP_Position::
      Misplaced(BB_JSP_Position::BBEL * trigBbel,
              MisplacedKS::KSAR * self)
{
      if (trigBbel->isActive(trigBbel)) {
            if (trigBbel->modifiedBy == NULL)
                  self->toRemove[0]='\000';
            else
                  strcpy(self->toRemove,
                        trigBbel->link(trigBbel->modifiedBy->which));
            return(TRUE);
      }
      return(FALSE);
}

void MisplacedKS::
      initFn(MisplacedKS::KS * self)
{
      self->runNumber=0;
}

void MisplacedKS::
      AllKSARs(MisplacedKS::KS * self,
              LIST ** agenda, LIST ** toSched)
{
      *toSched = *agenda;
      *agenda = NULLlist;
}

void MisplacedKS::
      MisplacedKS(MisplacedKS::KS * self, MisplacedKS::KSAR * ksar)
{
      int         t,r,b,l;
      char        link[4][NAMELEN];
      int         st;

      /* Check if the position is still active, if the piece      *
       * get its links.                                            */
      t=TOP; r=RIGHT; b=BOTTOM; l=LEFT;
      st = ReadRules(2,
              ksar->bbel.path, ksar->bbel.name,
                  "isActive()",
              ksar->bbel.path, ksar->toRemove,
                  "getLnk(%d,%s).getLnk(%d,%s).getLnk(%d,%s).getLnk(%d,%s)",
                  &t,&link[0], &r,&link[1], &b,&link[2], &l,&link[3]);
      if (st != OK)
            return;

      /* NULLify the link names to the border Piece. This          *
       * will make the system to skip it. This is needed as        *
       * two separate writes to the same Bbel, within one          *
```

```
Oct 19 19:01 1994  MisplacedKS-KS.sc Page 2

       * transaction will block the transaction.                  */
      if (link[0][0] == '0')      link[0][0] == '\000';
      if (link[1][0] == '0')      link[1][0] == '\000';
      if (link[2][0] == '0')      link[2][0] == '\000';
      if (link[3][0] == '0')      link[3][0] == '\000';

      WriteRules(1,
            ksar->bbel.path, ksar->toRemove, "removePiece().disable()");
      WriteRules(5,
            ksar->bbel.path, ksar->toRemove, "activate()",
            ksar->bbel.path, link[TOP],
                  "removeModifBy(%d).updateShape(%d)", BOTTOM, BOTTOM,
            ksar->bbel.path, link[RIGHT],
                  "removeModifBy(%d).updateShape(%d)", LEFT,   LEFT,
            ksar->bbel.path, link[BOTTOM],
                  "removeModifBy(%d).updateShape(%d)", TOP,    TOP,
            ksar->bbel.path, link[LEFT],
                  "removeModifBy(%d).updateShape(%d)", RIGHT,  RIGHT);
}
```

Oct 19 19:01 1994 MisplacedKS-KS.ah Page 1

```
/**********************************************************/
/* Knowledge Source Structure Definition.                 */
/* KS: MisplacedKS                                        */
/**********************************************************/

KSdefine MisplacedKS {
     long           runNumber;

     void           INIT::initFn(MisplacedKS::KS * self);
     void           KSCP::AllKSARs(MisplacedKS::KS * self,
                              LIST ** agenda, LIST ** toShed);
     void           KSBODY::MisplacedKS(MisplacedKS::KS * self,
                                   MisplacedKS::KSAR * ksar);
};

TRIGGERdefine MisplacedKS {
     char           toRemove[NAMELEN];

     BOOL BB.JSP.Position::
                    Misplaced(BB.JSP.Position::BBEL * trigBBel,
                              MisplacedKS::KSAR * self);
};


/*
 * Define KS instances.
 */
defineKS MisplacedKS MisplacedKS_1;
```

Oct 19 19:01 1994 PieceKS-KS.ac Page 1

```
/**********************************************************/
/* Knowledge Source Method Definition.                    */
/**********************************************************/


BOOL BB.JSP.Position::
     matchPiece(BB.JSP.Position::BBEL * trigBBel,
                PieceKS::KSAR * self)
{
     PATH          tmpPath = idPATH("BB.KS");
     PieceArray    Pieces;
     int           pieceNo;
     int           st;
     int           i;

     st = ReadBBels(1,
               tmpPath, self->ksName,
                    "getPieces(%j,%d)",sizeof(PieceArray),&Pieces,&pieceNo);
     if (st != OK)
          return(FALSE);

     for (i=0;i<pieceNo;i++)
          if (trigBBel->match(trigBBel,&Pieces[i],&(self->orientation))) {
               strcpy(self->pieceName,Pieces[i]);
               return(TRUE);
          }
     return(FALSE);
}

void PieceKS::
     initFn(PieceKS::KS * self)
{
     self->runNumber=0;
}

LIST * PieceKS::
     orderKSARs(PieceKS::KS * self, LIST * agenda)
{
     KSAR  ksar;

     return(agenda);
}

void PieceKS::
     PreferSoundPos(PieceKS::KS * self,
                    LIST ** agenda, LIST ** toShed)
{
     *toShed = *agenda;
     *agenda = NULLlist;
}

void PieceKS::
     PieceKS(PieceKS::KS * self, PieceKS::KSAR * ksar)
{
     int           ori;
     int           t,r,b,l;
```

Oct 19 19:01 1994 PieceKS-KS.ac Page 2

```
     char          link[4][NAMELEN];
     int           st;

     /* Check if the position is still active, if the piece    *
      * still matches the position and get its links.          */
     t=TOP; r=RIGHT; b=BOTTOM; l=LEFT;
     st = ReadBBels(1,
               ksar->bbel.path, ksar->bbel.name,
                    "isActive(),match(%s,%d),"
                    "getLink(%d,%s),getLink(%d,%s),getLink(%d,%s),getLink(%d,%s)",
                    ksar->pieceName,&ori,
                    &t,&link[0], &r,&link[1], &b,&link[2], &l,&link[3]);
     if ((st != OK) || (ori != ksar->orientation))
          return;

     /* NULLify the link names to the border Piece. This       *
      * will make the system to skip it. This is needed as     *
      * two separate writes to the same BBel, within one       *
      * transacction will block the transaction.               */
     if (link[0][0] == '0')     link[0][0] == '\000';
     if (link[1][0] == '0')     link[1][0] == '\000';
     if (link[2][0] == '0')     link[2][0] == '\000';
     if (link[3][0] == '0')     link[3][0] == '\000';

     /* If there is one, modify position and piece accordingly  */

     /* First, disable the piece and its surrounding pieces, so *
      * nobody will modify them until we finish with them.      */
     st = WriteBBels(5,
               ksar->bbel.path, ksar->bbel.name, "disable()",
               ksar->bbel.path, link[0], "disable()",
               ksar->bbel.path, link[1], "disable()",
               ksar->bbel.path, link[2], "disable()",
               ksar->bbel.path, link[3], "disable()");
     if (st != OK) return;

     /* Second, place the Piece in the Position and modify the  *
      * piece structure and the KS piece list.                  */
     WriteBBels(3,
          ksar->bbel.path, ksar->bbel.name,
               "placePiece(%s,%d)",ksar->pieceName,ksar->orientation,
          idPATH("BB.JSP.Piece"), ksar->pieceName,
               "place()",
          idPATH("BB.KS"), ksar->ksName,
               "place(%s)",ksar->pieceName);

     /* Finally, update and activate the surrounding positions  */
     WriteBBels(4,
          ksar->bbel.path,  link[0],
               "activate(),updateShape(%d),modifyPosition(%d),resetOld()",
               COMPL(0),COMPL(0),
          ksar->bbel.path,  link[1],
               "activate(),updateShape(%d),modifyPosition(%d),resetOld()",
               COMPL(1),COMPL(1),
          ksar->bbel.path,  link[2],
               "activate(),updateShape(%d),modifyPosition(%d),resetOld()",
```

Oct 19 19:01 1994 PieceKS-KS.ac Page 3

```
               COMPL(2),COMPL(2),
          ksar->bbel.path,  link[3],
               "activate(),updateShape(%d),modifyPosition(%d),resetOld()",
               COMPL(3),COMPL(3));
}
```

```
Oct 19 19:01 1994  PieceKS-KS.ah Page 1

/*********************************************************/
/* Knowledge Source Structure Definition.               */
/* KS: PieceKS                                           */
/*********************************************************/

KSdefine PieceKS {
     long          runNumber;

     LIST *        orderKSARs(PieceKS::KS * self, LIST * agenda);
     void          INIT::initFn(PieceKS::KS * self);
     void          KSCP::PreferSoundPos(PieceKS::KS * self,
                                  LIST ** agenda, LIST ** toShed);
     void          KSBODY::PieceKS(PieceKS::KS * self,
                              PieceKS::KSAR * ksar);
};

TRIGGERdefine PieceKS {
     int           orientation;
     char          pieceName[NAMELEN];

     BOOL BB.JSP.Position::
               matchPiece(BB.JSP.Position::BBEL * trigBBel,
                              PieceKS::KSAR * self);
};


/*
 * Define KS instances.
 */
defineKS PieceKS PieceKS_1;
defineKS PieceKS PieceKS_2;
defineKS PieceKS PieceKS_3;
defineKS PieceKS PieceKS_4;
defineKS PieceKS PieceKS_5;
```