The Engineering of Emergence in Complex Adaptive Systems

by

Anna Elizabeth Gezina Potgieter

submitted in partial fulfilment of the requirements for the degree of

Philosophiae Doctor

(Computer Science)

in the

Faculty of Engineering, Built Environment and Information Technology

University of Pretoria

Pretoria

2004

Acknowledgements

This thesis is dedicated to my family. To my husband Herman for his help during this quest and for his daily joy in experiencing the emergence of life, and to my dear children, Thys, Jessie and Annie, who emerged from our love.

Thanks to my supervisor, Professor Judith Bishop, for her assistance and guidance.

Thanks to my parents, for starting me off on this journey.

Thanks to the National Research Foundation for their financial assistance.

The Engineering of Emergence in a Complex Adaptive System

Student: Anna Elizabeth Gezina Potgieter
Study Leader: Professor Judith Bishop
Department: Computer Science
Degree: PhD (Computer Science)

SYNOPSIS

Agent-oriented software engineering is a new software engineering paradigm that is ideally suited to the analysis and design of complex systems. Open distributed environments place a growing demand on complex systems to be adaptive as well. Complex systems that can learn from and adapt to dynamically changing environments are called complex adaptive systems. These systems are characterized by emergent behaviour caused by interactions between system components and the environment. Agent-oriented software engineering methodologies attempt to control emergence during analysis and design by engineering the complex system in such a way that the correct emergent behaviour results during run-time. In a complex adaptive system however, emergent behaviour cannot be predicted during analysis and design, as it evolves only after implementation. By restricting emergent behaviour, as is done in most agent-oriented software engineering approaches, a complex system cannot be fully adaptive as well.

We propose the BaBe methodology that will enable a complex system to be adaptive by learning from its environment and modifying its behaviour during run-time. This methodology adds a run-time emergence model consisting of distributed Bayesian behaviour networks to the agent-oriented software engineering lifecycle. These networks are initialised by the human software engineer during analysis and design and deployed by Bayesian agencies (also complex adaptive systems). The Bayesian agents are simple, and collectively they implement distributed Bayesian behaviour networks. These networks, being specialized Bayesian networks, enable the Bayesian agents to collectively mine relationships between emergent behaviours and the interactions that caused them to emerge, in order to adapt the behaviour of the system. The agents are organized into heterarchies of agencies, where each agency activates one or more component behaviour depending on the inference in the underlying Bayesian behaviour network. These agencies assist the human software engineer to bridge the gap between the implementation and the understanding of emergent behaviour

in complex adaptive systems. Due to the simplicity of the agents and the minimal communication amongst them, they can be implemented using a commercially available component architecture. We describe a prototype implementation of the Bayesian agencies using Sun's Enterprise JavaBeans™ component architecture.

Keywords: complex adaptive systems, emergence, hyperstructures, Bayesian networks, agents, multi-agent systems, agencies, heterarchies, agent-oriented software engineering, component-based systems.

Table of Contents

List of Figures

List of Tables

It is heart-of-watermelon red
Mellow like an amber slice of the moon
As it emerges from high rock and low cloud
Suspended near the blueless sky
A spectre between nothing and nothing
Without a single ray of light
As if simple to say
Don't you know the world is remarkable

- Keorapetse Kgositsile

Chapter 1
Introduction

1.1 Complex Adaptive Systems

We live in a remarkable world, full of uncertainties to which we constantly adapt without really thinking about it. We are examples of systems called *complex adaptive systems,* which are complex systems that can learn from and adapt to their dynamically changing environments. In the computer world, open distributed environments, such as the Internet, place a growing demand on complex computer systems to be able to adapt to their environments. The uncertainty in these environments is mostly due to the behaviour of other complex adaptive systems such as users browsing web pages, the behaviour of buyers and sellers on the Internet and the behaviour of autonomous agents bidding on behalf of persons in Internet auctions.

The ability to adapt implies emergent behaviour. *Adaptive agents* are the basic building blocks of a complex adaptive system. These agents act together, interact with each other and the environment, and collectively adapt to changing environmental conditions. The interactions between the agents and the environment and the interactions between the agents themselves comprise a complex set of causal relationships.

All complex adaptive systems maintain *internal models*, consisting of *hyperstructures* representing "regularities" in the information about the system's environment and its own interaction with that environment. Hyperstructures are higher-order structures that emerge from the collective behaviour of the agents. Complex adaptive systems use these hyperstructures to act in the real world (Gell-Mann, 1994) (Holland, 1995).

1.2 Adaptive Agent Architectures

Agent architectures are software engineering models of agents (Wooldridge & Jennings, 1995). These architectures provide a new and natural way to analyse, design and implement complex software systems (Jennings, Sycara & Wooldridge, 1998). An agent architecture can be a *single-agent system* or a *multi-agent system*, composed of agents, coordinated through their relationships with one another. There are three types of agent architectures, namely *deliberative*, *reactive* and *adaptive* agent architectures. Fundamentally, deliberative agent architectures differ from reactive and adaptive agent architectures with respect to the presence or absence of emergence. Emergence is one of

the main characteristics of reactive and adaptive agent architectures, but is absent from deliberative agent architectures.

Deliberative agent architectures have internal world models that they manipulate. These world models are explicitly represented symbolic models of the world and decisions (for example what actions to perform next) are usually made using logical reasoning, based on pattern matching and symbolic manipulation (Wooldridge & Jennings, 1995). An example is the well-known and popular Belief-Desire-Intention (BDI) architecture (Rao & Georgeff, 1995). Deliberative architectures cannot function as complex adaptive systems as time constraints within complex, dynamic environments render them incapable of responding in time. The translation of the world into an accurate symbolic representation in time for it to be useful cannot realistically be achieved. Complex real-world processes are difficult to represent symbolically and to then reason about them using extensive deliberative processes cannot be done within the time constraints imposed by complex, dynamic environments (Wooldridge & Jennings, 1995). Deliberative agent architectures *cannot* handle emergence and can therefore not function as complex adaptive systems.

Reactive agent architectures are agent architectures that are situated in the world and that are embodied in that their actions are part of a dynamic interaction with the world (Brooks, 1991). These architectures do not include any kind of symbolic world model and they do not use complex symbolic reasoning (Wooldridge & Jennings, 1995). Examples include the Subsumption Architecture (Brooks, 1985) and autonomous adaptive agents developed by Maes (1990). Emergence is the most important characteristic of these agent architectures.

Adaptive agent architectures are agent architectures that can function as complex adaptive systems. These architectures maintain internal models consisting of hyperstructures in order to learn from and adapt to their dynamically changing environments. These architectures are usually reactive agent architectures that can learn from experience.

1.3 Agent-oriented software engineering

Agent-oriented software engineering refers to the software engineering approach followed in an agent architecture. This software engineering approach commonly focuses on two aspects, namely the design of the individual agents and the design of the interactions between different agents. These interactions are usually restricted to symbolic communication protocols. For example, in the Gaia methodology (Wooldridge, Jennings & Kinny, 2000), a system is analysed in terms of roles and interactions between

roles and then designed in terms of complex agents, the services to be provided by each agent and the lines of communication between different agents (acquaintances). The coordination-oriented methodology (Zambonelli, Jennings, Omicini & Wooldridge, 2000) added global laws that agents in an agency must obey when interacting with other agents.

Agent-oriented software engineering is commonly viewed as an extension to conventional component-based software engineering approaches (Griss & Pour, 2001). As most agent architectures are deliberative (Wooldridge & Jennings, 1995), agent-oriented software engineering is mostly concerned with the analysis and design of agents that are complex due to their complex reasoning capabilities. These agents are commonly viewed as "next-generation" components (Griss & Pour, 2001), as the current component architectures are too limited to implement components that exhibit flexible (reactive, pro-active, social) behaviour. These limitations are due to the fact that components and objects are passive in nature and the patterns of interactions between them are rigid and predetermined (Jennings, 2001).

No principled software engineering methodology exists for reactive and adaptive agent architectures (Jennings et al., 1998). These architectures are usually based on *ad hoc* principles as it is not clear how one should reason about them or what their underlying theory is (Wooldridge & Jennings, 1995) – the side effects of emergence. In most of these architectures, the engineering of emergence is a manual iterative process as in Brooks (1985), Maes (1994) and Bryson (2001) in which a human observer places the agent in its environment and observes its behaviour using a laborious process of experimentation, trial and error (Jennings et al., 1998). In a complex adaptive system, this becomes an impossible task for a human observer to perform.

1.4 Goal of this Thesis

The goal of this thesis is to define an adaptive agent architecture and to propose a methodology to engineer emergence in such an architecture. This thesis will prove that it is possible to use a commercially available component architecture to implement such an adaptive agent architecture. In this research, we defined the BaBe (Bayesian Behaviour Networks) adaptive agent architecture, as well as the BaBe agent-oriented software engineering methodology that is followed in this architecture. This methodology modifies and extends current agent-oriented software engineering approaches in order to include the engineering of emergence.

Our BaBe agent architecture is adaptive through the use of specialized Bayesian networks, which we call Bayesian behaviour networks, as hyperstructures. Bayesian

networks are ideally suited for probabilistic reasoning in uncertain environments and can be used by agent architectures to evolve and adapt in response to environmental changes.

Our BaBe methodology includes analysis and design models similar to other agent-oriented engineering approaches, as well as an additional run-time emergence model. This model consists of Bayesian behaviour networks, initialised by the software engineer during the analysis and design phases, and maintained by Bayesian agencies during the execution phase. The emergence model functions as the internal model of the adaptive agent architecture.

1.5 Research Contribution

The contribution of this research effort is twofold, namely:

1. A component-based implementation of adaptive hyperstructures in complex adaptive (software) systems. We defined an *adaptive agent architecture*, namely the BaBe agent architecture, and its implementation using *re-usable components*. In this architecture, the hyperstructures in the internal model are specialized Bayesian networks, which we call Bayesian behaviour networks. Bayesian learning in these networks can incrementally discover regularities in the information about the system's environment and its interaction with that environment. We call the components that are responsible for inference and learning in these networks, Bayesian agents. These simple agents are organized into heterarchies of agencies, which we call Bayesian agencies. A Bayesian agency can activate one or more component behaviours depending on the inference in the underlying Bayesian behaviour network in response to environmental states. Each component behaviour is a re-usable component that implements one or more actions. To our knowledge, adaptive hyperstructures in complex adaptive (software) systems have not yet been constructed from re-usable components.
2. The *automated engineering* of *emergence* in adaptive agent architectures. To our knowledge, the engineering of emergence in existing reactive and adaptive agent architectures involves a laborious manual engineering process. We defined the BaBe methodology that includes the automated engineering of emergence during the execution phase. This methodology includes the use of Bayesian behaviour networks as adaptive hyperstructures. This methodology also associates sets of actions, which we call competence sets, to Bayesian behaviour network node sets. Each competence set defines one or more actions that must be taken depending on the states of nodes due to inference in the Bayesian behaviour network.

1.6 Thesis Organization

Chapter 2 provides a background on complex adaptive systems and the emergence that characterizes these systems. Common to all these systems are *internal models*, consisting of *hyperstructures*. This chapter defines and discusses the role of hyperstructures in the engineering of emergence in complex adaptive systems.

Chapter 3 gives a theoretical overview of Bayesian networks and describes inference and learning in these networks. It then explains why these networks are suitable to be used as hyperstructures in complex adaptive systems.

Adaptive agent architectures are agent architectures that can function as complex adaptive systems. Agent architectures are software engineering models of agents. Chapter 4 gives an overview on agent architectures and underlying concepts of agents, agencies, heterarchies and hierarchies and discusses the difference between deliberative, reactive and adaptive agent architectures.

Agent-oriented software engineering refers to the software engineering methodologies applied in agent architectures. These methodologies are currently applied mostly in deliberative agent architectures. Chapter 5 compares object-oriented, component-based and agent-oriented software engineering methodologies and gives a brief overview of two agent-oriented software engineering methodologies, namely the Gaia methodology (Wooldridge et al., 2000) and the coordination-oriented methodology (Zambonelli et al., 2000).

Chapter 6 describes the BaBe adaptive agent architecture defined in this research. This includes a detailed description of our Bayesian agents and agencies, and how they collectively achieve belief propagation and learning in distributed Bayesian networks. A specialized class of Bayesian networks, which we call Bayesian behaviour networks, is presented and the application of these networks as hyperstructures in the BaBe architecture is explained.

In Chapter 7, we describe our BaBe methodology as a new methodology that addresses the absence of principled design methodologies for reactive and adaptive agent architectures. As a background, the Gaia methodology and the coordination-oriented methodology are described in more detail. This chapter compares the BaBe analysis, design and emergence models with the Gaia models and coordination models.

Chapter 8 describes a prototype implementation of the BaBe adaptive agent architecture and Chapter 9 concludes this thesis.

Chapter 2

Complex Adaptive Systems, Emergence and Engineering: The Basics

Atoms go their many ways –
each his own path lives.
For every way creates and makes but a moment,
that seems to us who know,
a moment that will die.
But yet, a flower's atoms will create (or be)
a flower's form – as it is, when it is
Forever

- Thys Potgieter

2.1 Introduction

A *complex adaptive system* is characterized by complex behaviours that *emerge* as a result of interactions among individual system components (or agents) and among system components (or agents) and the environment.

Examples of complex adaptive systems include:

- Users "foraging" for information, navigating from web page to web page along web links;
- The behaviour of consumers in a retail environment;
- The interactions between companies, consumers and financial markets in the modern capitalist economy;
- Intelligent autonomous agents bidding on peoples' behalf in Internet marketplaces;
- Bayesian networks, neural networks, genetic algorithms and artificial life systems.

The next section describes the properties and mechanisms common to all complex adaptive systems. Of these mechanisms, the internal model mechanism is instrumental in the engineering of emergence in these systems. All complex adaptive systems maintain internal models consisting of structures called *hyperstructures*, representing regularities in the

information about the system's environment and its own interaction with that environment (the input stream).

Emergent software engineering refers to the continuous process of adapting the hyperstructures to accommodate new regularities in the input stream. The hyperstructures can be adapted *manually* through human intervention, or *automatically*. Manual emergent engineering is a very cumbersome process requiring constant involvement from the human software engineer. In automatic emergent engineering, the hyperstructures are adaptive and able to evolve, eliminating the need for human intervention.

In this chapter, we describe the properties and mechanisms common to all complex adaptive systems. We give the background on hyperstructures and emergent engineering, and describe the difference between manual and automatic emergent engineering in more detail.

2.2 Properties and Mechanisms of Complex Adaptive Systems

A complex adaptive system consists of many system components acting together by interacting with each other and mutually affecting each other. Holland (1995) refers to a system component as an *agent* and describes four properties and three mechanisms common to all complex adaptive systems. We summarize Holland's description below:

1. *Aggregation* (Property) – simple agents are organized into adaptive aggregates, which in turn can form part of a higher level aggregation and so forth, yielding a hierarchical organization.
2. *Tagging* (Mechanism) – Tags facilitate selective interaction. They allow agents to select among agents or objects that would otherwise be indistinguishable.
3. *Non-linearity* (Property) – In a complex adaptive system, the whole does not equal the sum of the parts. Non-linear interactions amongst agents almost always make their collective behaviour more complicated than would be predicted from summing or averaging their individual behaviours.
4. *Flows* (Property) – In general terms, this property refers to flows over a network of nodes and connectors, where the nodes are agents, and the connectors designate the possible interactions between the agents. The flows in these networks vary over time.
5. *Diversity* (Property) – Diversity in a complex adaptive system is a dynamic pattern, often persistent and coherent and the product of progressive adaptations. Each new

adaptation opens the possibility for further interactions and opportunities for new interactions.

6. *Internal Models* (Mechanism) – Internal models are maintained in complex adaptive systems. These models are used for anticipation and prediction. Agents select patterns from the input stream and integrate these patterns into the structure of the internal model. The modified internal model must then enable the system to anticipate the consequences that follow when a similar pattern is encountered.
7. *Building Blocks* (Mechanism) – The re-use of basic building blocks to generate internal models is a pervasive feature of complex adaptive systems. As an example, the quarks of Gell-Mann (1994) are combined to yield nucleons, nucleons are combined into atoms, atoms are combined into molecules, and so forth.

2.3 The Engineering of Emergence

2.3.1 What is Emergence?

Emergence, the most important characteristic of a complex adaptive system, is the collective behaviour of interacting system components. Emergence leads to *holism* (Baas & Emmeche, 1997). A complex adaptive system is *holistic*, which means that the collective behaviour of the system components is more than the sum of the behaviours of the individual system components, for example a flock is more than a collection of birds and a traffic jam is more than a collection of cars (Odell, 1998).

Minsky (1988) describes holism as a "lack of understanding" (of an observer) due to the unexpected emergence of a phenomena that had not seemed inherent in the system components, showing that "a whole is more than the sum of its parts".

In the spirit of the Turing test, Ronald, Sipper & Capcarrěre (1999) formulated an emergence test based on this lack of understanding, which they call "amazement", as follows:

> *Our emergence test centers on an observer's avowed incapacity (amazement) to reconcile his perception of an experiment in terms of a global world view with his awareness of the atomic nature of the elementary interactions.*

Ronald et al. described the emergence test in terms of the following three conditions, namely *design*, *observation* and *surprise* of a system designer and a system observer (which could be the same), as follows:

1. *Design: The system has been constructed by the designer, by describing local interactions between components (e.g., artificial creatures and elements of the environment), in a language* $\mathcal{L}_1$.

2. *Observation: The observer is fully aware of the design, but describes global behaviours and properties of the running system, over a period of time, using a language* $\mathcal{L}_2$.

3. *Surprise: The language of design* $\mathcal{L}_1$ *and the language of observation* $\mathcal{L}_2$ *are distinct, and the causal link between the elementary interactions programmed in* $\mathcal{L}_1$ *and the behaviours observed in* $\mathcal{L}_2$ *is non-obvious to the observer - who therefore experiences surprise. In other words, there is a cognitive dissonance between the observer's mental image of the system's design stated in* $\mathcal{L}_1$ *and his contemporaneous observation of the system's behaviour stated in* $\mathcal{L}_2$.

The shuffling back and forth between $\mathcal{L}_1$ and $\mathcal{L}_2$, changing things on the one side and checking the effects on the other side, forms the basis of the engineering process (Ronald & Sipper, 2000). This can be either a *classical* engineering process in which surprises are not tolerated, or an *emergent* engineering process in which surprises are managed.

Classical software engineering is characterized by the intolerance of surprises. Ronald & Sipper refer to this absence of surprise as "unsurprise". For example, in the software engineering "waterfall model" (Sommerville, 1995) the requirement specifications are the language of (desired) observation $\mathcal{L}_2$, formulated in consultation with the client. The language $\mathcal{L}_2$ is then translated into a system and software design – the language of design $\mathcal{L}_1$, which is then used to implement and test the system during the implementation and unit-testing phase. During the integration and the system-testing phase, the client measures the

system performance against the observation language $\mathcal{L}_2$. During this phase, all surprises must be eliminated. This is done by either changing the observation language $\mathcal{L}_2$ in order to accommodate the surprises, or by changing the design language $\mathcal{L}_1$ and integrating the changes into the system.

Emergent software engineering refers to a software engineering approach that manages surprises as part of the software engineering cycle. Ronald & Sipper view emergent engineering as the management of a persistent $\mathcal{L}_1 - \mathcal{L}_2$ understanding gap, and they refer to the form of surprise that is managed as "unsurprising surprise".

2.3.2 Hyperstructures

Emergent phenomena are instances of some emergent higher-order structure that may be explained by the lower-level dynamics generating the collective behaviour (Baas & Emmeche, 1997). Baas & Emmeche refer to these structures as hyperstructures or "emergent explanations".

Hyperstructures are used for explanation and understanding. Gell-Mann (1994) refers to hyperstructures as "schemas", which he describes as follows:

> *a complex adaptive system acquires information about its environment and its own interaction with that environment, identifying regularities in that information, condensing those regularities into a kind of "schema" or model, and acting in the real world on the basis of that schema.*

Gell-Mann refers to the information about the environment of a complex adaptive system and the system's interaction with the environment as the "*input stream*" of the system. A complex adaptive system creates and maintains its hyperstructures by separating "regularities from randomness" in its input stream (Gell-Mann 1994). The set of hyperstructures in a complex adaptive system constitute the internal model of such a system. All complex adaptive systems maintain internal models (Holland, 1995). Emergence occurs as soon as the regularities identified in the input stream deviate from what is expected from the internal model maintained by the complex adaptive system. Cariani (1991) calls this "Emergence Relative to a Model".

According to Minsky (1988), emergence can usually be explained completely, once the interactions between the system components are taken into account. The internal model, consisting of hyperstructures, facilitates the explanation of emergence.

Baas & Emmeche defined the following formal framework for emergence and hyperstructures:

Let $\{\boldsymbol{S}_i\}\ \boldsymbol{i} \in \boldsymbol{I}$ *be a collection of general systems or "agents".*

Let $\boldsymbol{Obs}^1$ *be "observation" mechanisms and* $\boldsymbol{Int}^1$ *be interactions between agents.*

The observation mechanism measures the properties of the agents to be used in the interactions. The interactions then generate a new kind of structure

$$\boldsymbol{S}^2 = \boldsymbol{R}(\boldsymbol{S}_i^1, \boldsymbol{Obs}^1, \boldsymbol{Int}^1)$$

which is the result of the interactions. This could be a stable pattern or a dynamically interacting system. We call $\boldsymbol{S}^2$ *an emergent structure, which may be subject to new observational mechanisms* $\boldsymbol{Obs}^2$*. This leads to*

Definition:

P is an emergent property

$$\Updownarrow$$

$$\boldsymbol{P} \in \boldsymbol{Obs}^2(\boldsymbol{S}^2) \text{ and } \boldsymbol{P} \notin \boldsymbol{Obs}^2(\boldsymbol{S}_i^1)$$

The observational mechanism may be internal or external. Hyperstructures are multi-level emergent structures.

Definition:

A hyperstructure of order N is given by

$$\boldsymbol{S}^N = \boldsymbol{R}(\boldsymbol{S}_{i_{N-1}}^{N-1}, \boldsymbol{Obs}^{N-1}, \boldsymbol{Int}^{N-1}, \boldsymbol{S}_{i_{N-2}}^{N-2}, \boldsymbol{Obs}^{N-2}, \boldsymbol{Int}^{N-2})$$

extending the construction in the definition of emergence. This is a cumulative structure, not necessarily purely recursive.

In the framework above, $\boldsymbol{I}$ denotes the number of general systems or agents in the system.

The *observation mechanism* of a complex adaptive system is responsible for the identification of regularities in its input stream, as well as for the adaptation of the hyperstructures to accommodate these regularities. The process of adaptation in a complex adaptive system relies heavily on its observation mechanism. In terms of the $\mathcal{L}_1$ and $\mathcal{L}_2$ languages of Ronald et al., hyperstructures can be viewed as constituting the language of design $\mathcal{L}_1$ and the observed properties or regularities of a system are described in an observation language $\mathcal{L}_2$. For example, the emergent property $\boldsymbol{P}$ in Baas & Emmeche's framework above will be included in $\mathcal{L}_2$.

The process of adaptation in a complex adaptive system involves a progressive modification of the hyperstructures (Holland, 1975) in order to bridge the $\mathcal{L}_1$-$\mathcal{L}_2$ understanding gap. For example in Baas & Emmeche's framework, $\boldsymbol{S}^N$ can be viewed as the language of design $\mathcal{L}_1$ and $\boldsymbol{S}^{N+1}$ can be viewed as the language of observation $\mathcal{L}_2$, if there exists an emergent property $\boldsymbol{Q}$ such that

$$\boldsymbol{Q} \in \boldsymbol{Obs}^{N+1}(\boldsymbol{S}^{N+1}) \text{ and } \boldsymbol{Q} \notin \boldsymbol{Obs}^{N+1}(\boldsymbol{S}^{N})$$

To bridge the $\mathcal{L}_1$-$\mathcal{L}_2$ understanding gap, $\boldsymbol{S}^N$ must be adapted to

$$\boldsymbol{S}^{N+1} = \boldsymbol{R}(\boldsymbol{S}^{N}_{i_N}, \boldsymbol{Obs}^{N}, \boldsymbol{Int}^{N}, \boldsymbol{S}^{N-1}_{i_{N-1}}, \boldsymbol{Obs}^{N-1}, \boldsymbol{Int}^{N-1}, \boldsymbol{S}^{N-2}_{i_{N-2}}, \boldsymbol{Obs}^{N-2}, \boldsymbol{Int}^{N-2}).$$

$\boldsymbol{S}^{N+1}$ then becomes the language of design, $\mathcal{L}_1$.

Observation mechanisms can be either *internal* or *external*. Complex adaptive systems in nature have internal observation mechanisms, observing their own behaviours in order to maintain the hyperstructures in their internal models. Most computer-based complex adaptive systems have external observation mechanisms. In these systems, the observer is usually the software engineer, maintaining the hyperstructures manually.

2.3.3 External Observation Mechanisms

In a complex adaptive system with an external observation mechanism, the internal model consists of *static* hyperstructures, usually hand-built or compiled from specifications. We refer to hyperstructures such as these as "static" because once implemented, their structure can only be changed by re-engineering them. In most of these systems, the external observation mechanism is the human software engineer, observing the behaviour of his or her

"creation", operating in its environment. The engineer identifies regularities in the input stream of the system and adapts the hyperstructures manually to include these regularities.

The Subsumption Architecture (Brooks, 1990) is an example of a system that uses an internal model consisting of static hyperstructures. In this architecture, the hand-wired arbitration network is a hyperstructure, consisting of a set of suppression and inhibition wires as well as simple arbitration circuitry. These wires form a network connecting behaviour modules with each other. The behaviours are augmented finite state machines (AFMS) mapping perceptual input to action output, and behaviour modules can inhibit or subsume the behaviour of other modules. Rosenblatt & Payton (1989) implemented a more sophisticated version of the Subsumption Architecture in which the behaviours do not completely subsume other behaviours, but bias decisions in favour of different alternatives. Static hyperstructures, as described above, have to be handcrafted until the system exhibits the desired behaviour when operating in its environment.

Maes (1990) used static hyperstructures, called behaviour networks, to represent regularities in the input stream and to control action selection in response to environmental states. These networks model the relationships between "competence modules" that react to states of the environment, and that spread activation energy along links in the behaviour network. The structure of these networks is specified by the system designer (also the external observer), and then compiled into circuit diagrams, which are then implemented. Once implemented, the structure of the behaviour networks can only be changed by recompilation and re-implementation.

In the systems described above, the human software engineer is an external observation mechanism that observes emergence relative to the internal model of the system. In these systems, emergent engineering is a manual process, involving extensive simulation, and testing in order to make sure that the system meets its requirements. This process involves manual modification of the hyperstructures, either by specification and subsequent compilation or by manually modifying the wiring structures in the implementation itself. We call this form of emergent engineering *manual emergent engineering*.

2.3.4 Internal Observation Mechanisms

A complex adaptive system with an internal observation mechanism has hyperstructures that are *adaptive* and able to *evolve*. The internal observation mechanism identifies regularities in the input stream and automatically modifies the hyperstructures in the internal model to include these regularities. Two examples of adaptive hyperstructures include the K-lines

constituting Minsky's Society of Mind, and the genetic structures defined by Holland (1975). We call this form of emergent engineering *automatic emergent engineering*.

The next two sections give a brief overview of Minsky and Holland's internal observation mechanisms.

2.3.4.1 Minsky's Society of Mind

The mind is capable of self-observation and self-interaction. Consciousness may be seen as an internal model maintained by the mind. In Minsky's Society of Mind, internal observation mechanisms called *A-Brains* and *B-Brains* maintain internal models consisting of hyperstructures called K-Lines. Each K-Line is a wire-like structure that attaches itself to whichever mental agents are active when a problem is solved or a good idea is formed (Minsky, 1988).

Minsky describes how a system can watch itself, using a *B-Brain*. In Figure 1, the A-Brain has inputs and outputs that are connected to the real word, and the B-Brain is connected to the A-Brain. The A-Brain can sense and influence what is happening in the world, and the B-Brain can see and influence what is happening inside the A-Brain.

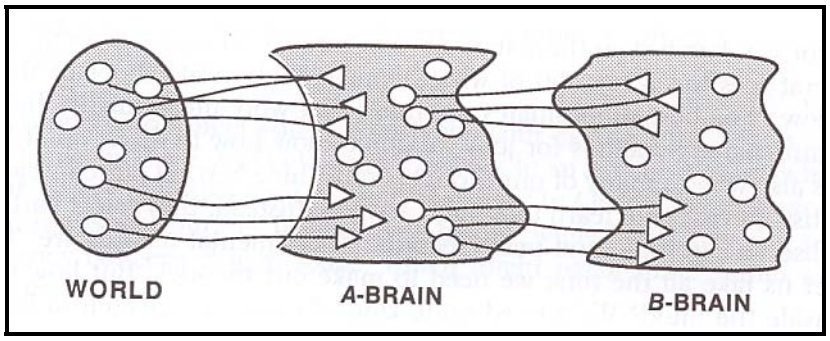

Figure 1: Minsky's A- and B-Brain (Minsky, 1988)

The A-Brain and B-Brain engineer emergence *automatically*, continuously adapting their K-Lines in order to bridge the $\mathcal{L}_1$ - $\mathcal{L}_2$ understanding gap.

2.3.4.2 Holland's Genetic Structures

Holland (1975) uses an *adaptive plan* as an internal observation mechanism. He defined a formal framework for a complex adaptive system as a set of objects $(\mathcal{a}, \Omega, I, \tau)$, where

$\mathcal{a} = \{A_1, A_2, ...\}$ *is the set of attainable structures, the domain of action of the adaptive plan,*

$\Omega = \{\omega_1, \omega_2, ...\}$ *is the set of operators for modifying the structures with* $\omega \in \Omega$ *being a function* $\omega : \mathcal{a} \rightarrow \mathcal{P}$, *where* $\mathcal{P}$ *is some set of probability distributions over* $\mathcal{a}$,

I *is the set of possible inputs to the system from the environment, and*

$\tau : I \times \mathcal{a} \rightarrow \Omega$ *is the adaptive plan which, on the basis of the input and structure at time t, determines what the operator is to be applied at time t.*

In Holland's formal framework, the *adaptive plan* (τ) is the internal observation mechanism of the complex adaptive system and $\mathcal{a}$ is the set of hyperstructures. Holland describes how a complex adaptive system can genetically adapt using a genetic adaptive plan (τ). Such a plan develops from the hyperstructures, which are populations of chromosomes ($\mathcal{a}$), interacting with the environment. The genetic operators (Ω) include for example mutation, crossover, inversion, and so forth.

In this framework, the engineering of emergence is automatic. The adaptive plan automatically applies operators that modify the population of chromosomes (the hyperstructures) in order to adapt to environmental inputs.

2.4 Conclusion

A complex adaptive system consists of many system components or agents acting together by interacting with each other and mutually affecting each other. Emergence refers to the collective behaviour of these system components.

Hyperstructures are higher-order structures that emerge from the lower-level dynamics generating collective behaviour (Baas & Emmeche, 1997). These structures are "emergent explanations" and are instrumental in the engineering of emergence in complex adaptive systems. Hyperstructures represent regularities in the input stream of a complex adaptive

system, and in turn constitute the internal model of such a system. Internal models are mechanisms common to all complex adaptive systems (Holland, 1995).

The engineering of emergence in a complex adaptive system involves a continuous process employing an (external or internal) observation mechanism to identify regularities in the information about the system's environment and about its own interaction with that environment (the input stream) and updating the hyperstructures in the internal model whenever new regularities are identified in the input stream. Emergence refers to the unexpected deviation of the regularities in the input stream from what is expected from the hyperstructures in the internal model.

If an external observation mechanism is employed to observe emergence relative to an internal model, emergent engineering is a manual process, involving extensive simulation, and testing in order to make sure that the system meets its requirements. This process involves manual modification of the hyperstructures, either by specification and subsequent compilation or by manually modifying the wiring structures in the implementation itself. We call this form of emergent engineering manual emergent engineering. This is a cumbersome process and far from ideal. In most of these systems, the external observation mechanism is the human software engineer, observing the behaviour of his or her "creation", operating in its environment. The engineer identifies regularities in the input stream of the system and adapts the hyperstructures manually to include these regularities.

A complex adaptive system with an internal observation mechanism has hyperstructures that are adaptive and able to evolve. The internal observation mechanism identifies regularities in the input stream and automatically modifies the hyperstructures in the internal model to include these regularities. The automatic engineering of emergence eliminates the need for human intervention. All complex adaptive systems in nature have internal observation mechanisms and they maintain their own internal models using automatic emergent engineering.

Chapter 3

Bayesian Networks as Hyperstructures

To every probability there is an opposite possibility. This possibility is compos'd of parts, that are entirely of the same nature with those of the probability; and consequently have the same influence on the mind and understanding. The belief, which attends the probability, is a compounded effect, and is form'd by the concurrence of several effects, which proceed from each part of the probability. Since therefore each part of the probability contributes to the production of the belief, each part of the possibility must have the same influence on the opposite side; the nature of these parts being entirely the same

- David Hume (1888)

3.1 Introduction

All complex adaptive systems maintain internal models, consisting of hyperstructures, to represent regularities in their input streams. An input stream consists of information about entities in the environment of the system and the interaction of the system with these entities. Regularities in the input stream are probabilistic in nature. Gell-Mann describes this probabilistic nature as follows:

Any entity in the world around us, such as an individual human being, owes its existence not only to the simple fundamental law of physics and the boundary condition on the early universe but also to the outcomes of an inconceivably long sequence of probabilistic events, each of which could have turned out differently.

(Gell-Mann, 1995)

Beliefs are formed as a "distillation of sensory experiences" during a process in which the actual experiences are learnt in terms of averages, weights or qualitative relationships that are used to determine future actions (Pearl, 1988). Bayesian networks (also called Bayesian

belief networks or causal networks) provide the ideal formalism to express these probabilistic regularities. These networks encode beliefs and causal relationships between beliefs and provide a formalism for reasoning about partial beliefs under conditions of uncertainty (Pearl, 1988). A complex adaptive system can use a Bayesian network as a probabilistic model of what the emergent effects are of certain interactions and behaviours in response to certain environmental states (the causes). Such a causal model can then be queried by an arbitration process to decide which action(s) are most relevant given a certain state of the environment. Bayesian networks are therefore ideally suited to be used as hyperstructures in the internal models of complex adaptive systems.

This chapter gives a background on Bayesian networks and describes learning and inference in these networks. The suitability of Bayesian networks as hyperstructures is then described in terms of Holland's properties and mechanisms of complex adaptive systems.

3.2 Basic Concepts

In this section, the basics concepts and definitions needed to understand Bayesian networks are given.

3.2.1 Propositions

A *proposition* is a statement or assertion about a state of the world.

3.2.2 Variables

A variable X is a set of propositions x_i such that these propositions are mutually exclusive (for all i, j, with $i \neq j$ $(x_i \wedge x_j)$ is false)[1] and exhaustive (at least one of the propositions x_i is true). Variables are denoted by upper-case letters, for example (X, Y, Z), and the propositions of variables in lowercase, for example (X=x, Y=y, Z=z), also shortened as (x, y, z). Sets of variables are represented by boldface uppercase letters, for example ($\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$).

A variable can be *observable* or *latent*. A latent or hidden variable is a variable of which the states are inferred but never observed directly.

[1] $\wedge$ means "and"

3.2.3 Probabilities

In order to deal with uncertainty, probabilities are attached to propositions. For example, *P(X=x)* represents the probability that *X*=*x*, also shortened as *P*(*x*). Each probability reflects a *degree of belief* rather than a frequency of occurrence.

Eqs. (1) to (11) below describes the fundamental principles of Bayesian methods, as formulated in (Pearl, 1988).

Beliefs must obey the three basic axioms of probability theory, namely

$$0 \le P(A) \le 1, \tag{1}$$

$$P(\textit{Sure Proposition}) = 1, \tag{2}$$

$$P(A \text{ or } B) = P(A) + P(B) \text{ if } A \text{ and } B \text{ are mutually exclusive.} \tag{3}$$

The third axiom states that the belief assigned to any set of events is the sum of the beliefs assigned to its nonintersecting components. Therefore, since any event A can be written as the union of the joint events (A and B) and (A and $\neg B$), their associated probabilities are given by

$$P(A) = P(A,B) + P(A,\neg B), \tag{4}$$

where $P(A,B)$ is short for $P(A \text{ and } B)$. More generally, if $B_i, i = 1,2...n$ is a set of exhaustive and mutually exclusive propositions, then $P(A)$ can be computed from $P(A,B_i), i = 1,2...n$ using the sum

$$P(A) = \sum_i P(A,B_i). \tag{5}$$

P(A) above is called the *marginal probability* of A (Nilsson, 1998) and the $B_i, i = 1,2...n$ are said to be *marginalized out* or *summed out* (Russell & Norvig, 2003). A direct consequence of Eqs. (2) and (4) is that a proposition and its negation must be assigned a total belief of unity,

$$P(A) + P(\neg A) = 1. \tag{6}$$

The basic expressions in the Bayesian formalism are statements about *conditional probabilities*, for example $P(A \mid B)$ represents the belief in A under the assumption that B is known with absolute certainty. In $P(A \mid B)$, A is said to be *conditioned on B*.

Conditional probabilities can be defined in terms of *joint events*,

$$P(A \mid B) = \frac{P(A,B)}{P(B)}, \tag{7}$$

where $P(A,B)$ is the belief in the joint event of A and B (also called a joint probability), and $P(B)$ is the marginal probability of B. The belief in joint events can be calculated using the *product rule*

$$P(A,B) = P(A \mid B)P(B) \tag{8}$$

From Eqs. (5) and (8), it follows that the probability of any event A can be computed by conditioning it on any set of exhaustive and mutually exclusive events $B_i, i = 1,2...n$:

$$P(A) = \sum_i P(A \mid B_i)P(B_i) \tag{9}$$

This equation states that the belief in any event A is a weighted sum over the beliefs in all the distinct ways that A might be realized.

The product rule (Eq. (8)) can be generalized by the so-called *chain-rule* formula. It states that for a set of n events $E_1, E_2...E_n$, the probability of the joint event ($E_1, E_2...E_n$) can be written as the product of n conditional probabilities:

$$P(E_1, E_2...E_n) = P(E_n \mid E_{n-1},..., E_2, E_1)...P(E_2 \mid E_1)P(E_1) \tag{10}$$

The *inversion formula*

$$P(H \mid e) = \frac{P(e \mid H)P(H)}{P(e)} \tag{11}$$

states that the belief in a *hypothesis* H in the presence of *evidence* e can be computed by multiplying the previous belief $P(H)$ by the likelihood $P(e \mid H)$ that e will materialize if H is true. $P(H \mid e)$ is sometimes called the *posterior* probability and $P(H)$ is called the *prior* probability.

Eq. (11) is called *Bayes' Rule* and forms the basis of Bayesian techniques.

3.2.4 Conditional Independence

A variable, V, is *conditionally independent* of a set of variables, $\boldsymbol{V_i}$, given a set $\boldsymbol{V_j}$, if $P(V \mid \boldsymbol{V_i}, \boldsymbol{V_j}) = P(V \mid \boldsymbol{V_j})$, represented by the notation $I(V, \boldsymbol{V_i} \mid \boldsymbol{V_j})$ (Nilsson, 1998).

Eqs. (12) to (16) describe conditional independence, as formulated in (Nilsson, 1998).

If a variable, V_i, is conditionally independent of another variable, V_j, given a set $\boldsymbol{V}$, the notation $I(V_i, V_j \mid \boldsymbol{V})$ is used, meaning

$$P(V_i \mid V_j, \boldsymbol{V}) = P(V_i \mid \boldsymbol{V}). \tag{12}$$

From the product rule (Eq. (8)),

$$P(V_i \mid V_j, \boldsymbol{V}) P(V_j \mid \boldsymbol{V}) = P(V_i, V_j \mid \boldsymbol{V}). \tag{13}$$

Combining Eqs. (12) and (13) yields

$$P(V_i, V_j \mid \boldsymbol{V}) = P(V_i \mid \boldsymbol{V}) P(V_j \mid \boldsymbol{V})$$

If $\boldsymbol{V}$ is empty, then V_i and V_j are said to be independent.

The form of independence described above, is called *pairwise independence*. As a generalization of this form of independence, the variables $V_1,\ldots,V_k$ are *mutually conditionally independent,* given a set $\boldsymbol{V}$ if each of the variables is conditionally independent of all of the others, given $\boldsymbol{V}$. From Eq. (10)

$$P(V_1, V_2, \ldots, V_k \mid \boldsymbol{V}) = \prod_{i=1}^{k} P(V_i \mid V_{i-1}, \ldots, V_1, \boldsymbol{V}) \tag{14}$$

and, since each V_i is conditionally independent of the others given V,

$$P(V_1, V_2, \ldots, V_k \mid \boldsymbol{V}) = \prod_{i=1}^{k} P(V_i \mid \boldsymbol{V}). \tag{15}$$

When $\boldsymbol{V}$ is empty,

$$P(V_1, V_2, \ldots, V_k) = P(V_1) P(V_2) \ldots P(V_k), \tag{16}$$

and the variables $(V_1, V_2, \ldots, V_k)$ are said to be *unconditionally independent.*

3.3 What is a Bayesian Network?

A *Bayesian network* is a directed acyclic graph (DAG) that consists of a set of nodes that are linked together by directional links. Each node represents a random variable or uncertain quantity. Each variable has a finite set of mutually exclusive propositions, called *states*. The links represent informational or causal dependencies among the variables, where a parent node is the *cause* and a child node the *effect*. The dependencies are given in terms of conditional probabilities of states that a node can have given the values of the parent nodes

(Pearl, 1988) (Dechter, 1996) (Pearl & Russell, 2000). A Bayesian network can either be singly-connected (without loops) or multiply-connected (with loops).

To each node X there is attached a conditional probability matrix $P = \{P(X \mid \boldsymbol{pa(X)})\}$, where $\boldsymbol{pa(X)}$ represents the value combinations of the parents of X. For example for variable X with set of states $\{x_1, x_2, ..., x_m\}$ and Y with set states $\{y_1, y_2, ..., y_n\}$, the conditional probability matrix $P(\boldsymbol{y} \mid \boldsymbol{x})$ represents the conditional probability of Y given X as follows:

$$P(\boldsymbol{y} \mid \boldsymbol{x}) = \begin{bmatrix} P(y_1 \mid x_1) & P(y_2 \mid x_1) & \ldots & P(y_n \mid x_1) \\ P(y_1 \mid x_2) & P(y_2 \mid x_2) & \ldots & P(y_n \mid x_2) \\ \vdots & \vdots & \vdots & \vdots \\ P(y_1 \mid x_m) & P(y_2 \mid x_m) & \ldots & P(y_n \mid x_m) \end{bmatrix}$$

For a set of variables $\boldsymbol{Z} = (Z_1, Z_2 \ldots . Z_n)$ represented by a Bayesian network, the network represents a global joint probability distribution over $\boldsymbol{Z}$ having the product form

$$P(z_1, ... z_n) = \prod_{i=1}^{n} P(z_i \mid \boldsymbol{pa(z_i)}) \qquad \text{(Dechter, 1996) (Pearl, 1988)} \qquad (17)$$

Figure 2 illustrates a Bayesian network, which we adapted from the user-words aspect model proposed by Popescul, Ungar, Pennock & Lawrence (2001), which models the three-way co-occurrence between users, words and documents in a document recommender system. Our network models the relationship between users (U), the contents of browsed web pages characterized in terms of concepts (C), and products bought from these pages (P). Our simple model includes the three-way co-occurrence data among two users, two products and two concepts.

The users in Figure 2 are represented by $u \in U$ = *{mathematician (m), rugby player(r)}*. The products are represented by $p \in P$ = *{book authored by Professor Michael Jordan on neural networks (nn), book authored by Michael Jordan, the well-known basketball player, on basketball (bb)}*. The concepts inferred from the web pages the users viewed are represented by $c \in C$ = *{statistics(st), sport(sp)}*. The users (U), products (P) and concepts (C) form observations (u, c, p), which are associated with a latent variable class (Z). There are two latent classes, namely $z \in Z$ = {*class1*(*c1*), *class2*(*c2*)}. In Figure 2, the conditional probability matrices are shown next to the nodes, and marginal probabilities are indicated on the nodes.

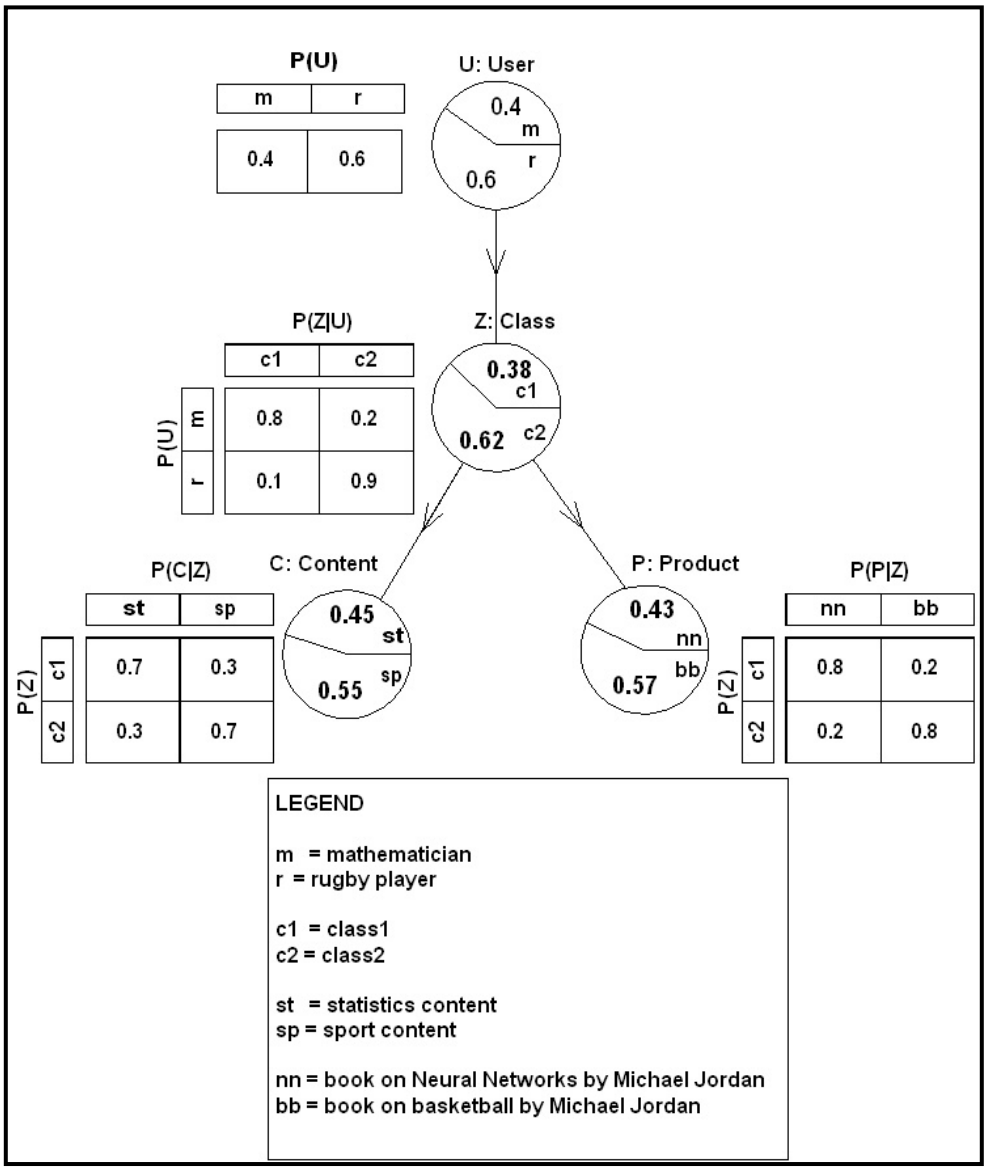

Figure 2: A Bayesian Network

The example Bayesian network above represents the joint distribution:

$$P(u,z,c,p) = P(u)P(z \mid u)P(c \mid z)P(p \mid z) \tag{18}$$

Using Bayes' rule (Eq. (11)), an equivalent joint distribution for Eq. (18) is given by

$$P(u,z,c,p) = P(z)P(u \mid z)P(c \mid z)P(p \mid z) \tag{19}$$

Using (Eq. (5)), the joint distribution over users, contents and products is given by

$$P(u,c,p) = \sum_z P(z)P(u \mid z)P(c \mid z)P(p \mid z) \tag{20}$$

by marginalizing z out.

In Eq. (20) above, we have a single cause (Z) influencing multiple effects (U, P and C), as illustrated in Figure 3. This probability distribution is called a naïve Bayes model or sometimes called a Bayesian classifier (Russell & Norvig, 2003).

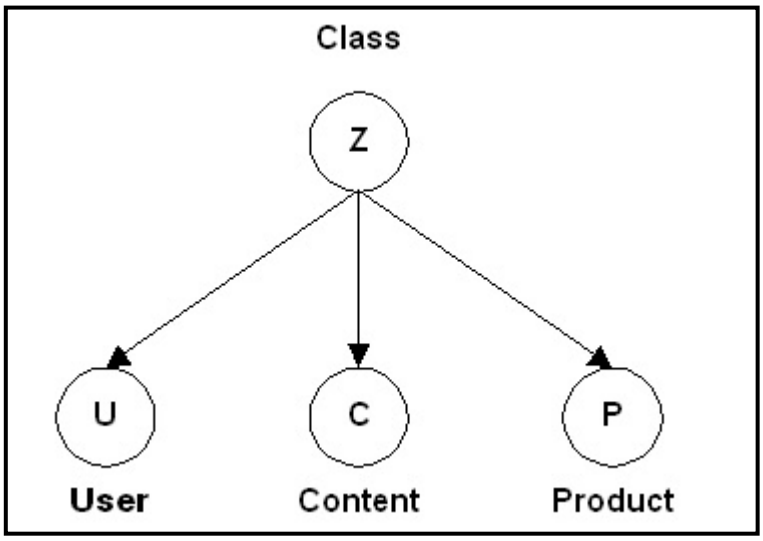

Figure 3: Naïve Bayes Model

3.4 Dynamic Bayesian Networks

In a changing environment, some variables can have values that change over time. In dynamic Bayesian networks, multiple copies of the variables are represented, one for each time step (Pearl & Russell, 2000). Figure 4 illustrates a dynamic Bayesian network that models the three-way co-occurrence data among users, products and concepts at different time-steps. Each time-step corresponds to a different web page being browsed. This model merges a Hidden Markov Model[2] and a naïve Bayes model.

[2] A Hidden Markov Model is a temporal probabilistic model, where each time step has one observed variable depending on one hidden variable. The hidden variables are distributed according to a Markov process.

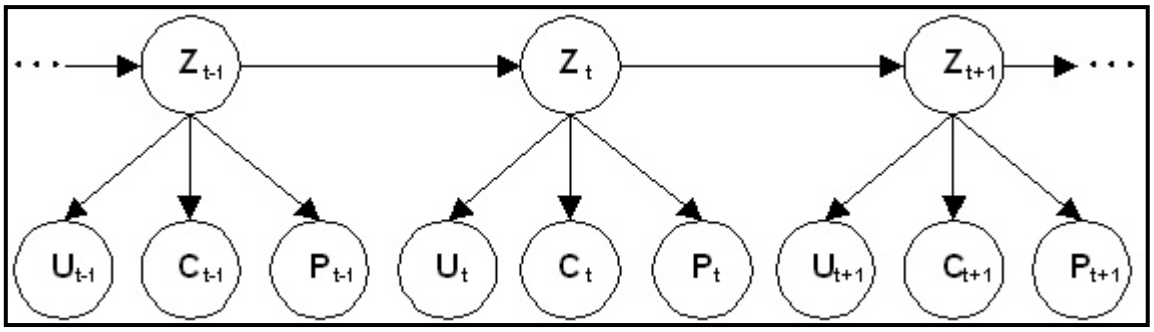

Figure 4: A Dynamic Bayesian Network

3.5 Conditional Independence in Bayesian Networks

A Bayesian network represents conditional independencies between variables (Nilsson, 1998). Using the conditional independencies encoded in the Bayesian network graph structure together with the conditional probability matrices, the full joint probabilities of all the variables represented in the network can be reconstructed. In the next sections, we will discuss two forms of independencies in the Bayesian network representation. The first form involves the parents of nodes, and the second form is called d-separation.

3.5.1 Conditional Independence involving the Parents of a Node

In a Bayesian network, each node is conditionally independent of all its non-descendent nodes, given its parent nodes. Pearl & Russell (2000) refer to this as the "local semantics" of a Bayesian network. Pearl & Russell further refer to equation (17) as the "global semantics" of a Bayesian network. Eqs. (17) and (18) are global distributions described in terms of local distributions. For example, using the local semantics of the Bayesian network in Figure 2,

$$P(c \mid u, z) = P(c \mid z) \tag{21}$$

The local semantics of Bayesian networks make this technology ideal for distributed implementation.

3.5.2 D-Separation

D-separation is another form of conditional independence encoded in a Bayesian network. Nilsson (1998) describes and illustrates d-separation, as follows:

Two nodes X and Y are conditionally independent given a set of evidence nodes $\boldsymbol{\varepsilon}$ (that is, $I(X,Y \mid \boldsymbol{\varepsilon})$) if, for every *undirected* path in the Bayesian network between X and Y, there is some node, B, on the path having one of the following three properties (see Figure 5):

1. B is in $\boldsymbol{\varepsilon}$, and both arcs on the path lead out of B (for example B_1 in Figure 5).
2. B is in $\boldsymbol{\varepsilon}$, and one arc on the path leads in to B and one arc leads out (for example B_2 in Figure 5).
3. Neither B nor any descendant of B is in $\boldsymbol{\varepsilon}$, and both arcs on the path lead in to B (for example B_3 in Figure 5).

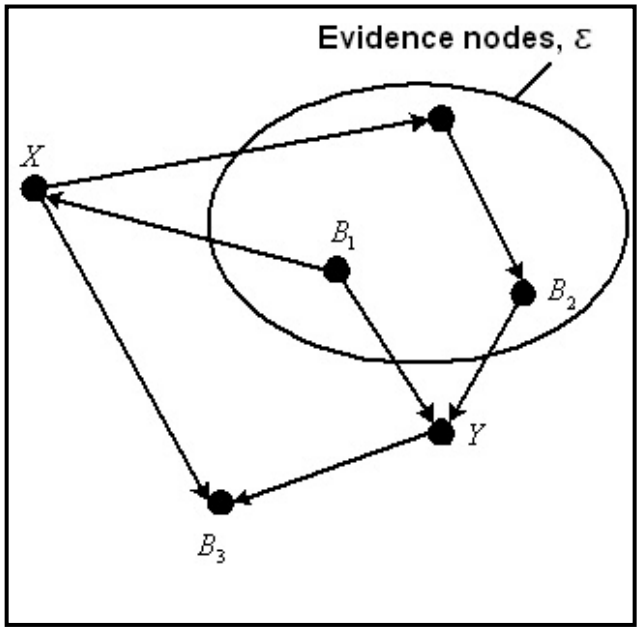

Figure 5: Conditional Independence via Blocking Nodes (Nilsson, 1998)

When any one of the above conditions holds for a path, it is said that node B *blocks* the path between X and Y, given $\boldsymbol{\varepsilon}$. Note that the paths referred to are undirected paths, that is, paths that ignore arc directions. If all paths between X and Y are blocked, then it is said that $\boldsymbol{\varepsilon}$ *d-separates* X and Y (direction-dependent separation) and the conclusion can be made that X and Y are conditionally independent given $\boldsymbol{\varepsilon}$.

In Figure 5, X is independent of Y given the evidence nodes because all three paths between them are blocked. The nodes B_1, B_2 and B_3 are blocking nodes.

3.6 Bayesian Learning

Bayesian learning can be described as finding a network and calculating the conditional probability matrices for a set of training data consisting of a set of instances for the variables. Table 1 is an example training set for the Bayesian network in Figure 2.

***U*:User**	***C*:Content**	***P*:Product**	**Number of Instances**
m	*st*	*nn*	$count_1$
m	*st*	*bb*	$count_2$
m	*sp*	*nn*	$count_3$
m	*sp*	*bb*	$count_4$
r	*st*	*nn*	$count_5$
r	*st*	*bb*	$count_6$
r	*sp*	*nn*	$count_7$
r	*sp*	*bb*	$count_8$
Total			$\sum_{i=1}^{8} count_i$

Table 1: A Training Set

Note that there are no entries for *Z*, as *Z* is a latent variable, and it must be treated as "missing data". For a complex adaptive system, the training data is the input stream over a given time period. A training set is characterized by the number of parameters it has as well as the number of samples in the training set. For each combination of different values for variables, there will be one entry in the training set table, called a parameter. For each parameter, there is a count of the number of instances in the training data that had these values for its variables. The total number of samples is the sum of all the instances in the training set.

There are different conditions that can influence Bayesian learning. The structure of the Bayesian network can be known or unknown and the variables can be observable or hidden.

3.6.1 Known Structure

If the structure of the Bayesian network is known, then only the conditional probability matrices need to be calculated. If there is missing data the calculations differ from the case when there is no missing data.

3.6.1.1 No Missing Data

For a known network structure and no hidden variables, the values of the conditional probability matrices are easy to calculate from the sample statistics for each node and its parents (Nilsson, 1998).

3.6.1.2 Missing Data

If the network structure is known, but there are hidden variables or missing data, the Expected Maximization (EM) algorithm is commonly used to calculate the conditional probability matrices. This algorithm will not be described here, as a detailed study of Bayesian learning falls outside the scope of this research. Russell & Norvig (2003) describe this algorithm and its use in detail.

Nilsson (1998) describes how the EM algorithm can be used to calculate the conditional probability matrices if there are missing data or hidden variables. The process starts by assigning random values to all the cells of the conditional probability matrices. Conditional probabilities (weights) are then calculated for the missing data, given the values of the observed data. These weights are then used to estimate new conditional probability matrices, until the matrices converge, which according to Nilsson, are guaranteed and mostly rapid.

3.6.2 Unknown Structure

If the structure is unknown and all the variables are observable, then the learning problem involves a search through the possible structures to find the structure that represents the data best, followed by the updating of the conditional probability matrices.

Nilsson (1998) describes a scoring metric that can be used to compare different structures in order to choose the structure that represents the data best (Eqs. (22) and (23)), which we summarize below:

The scoring metric measures the *minimum description length* of the joint probability of a set of variables in a training set.

Suppose the training data, Ξ, consists of m samples: $\boldsymbol{v}_1,...,\boldsymbol{v}_m$ where each $\boldsymbol{v}_i$ is an n-dimensional vector of values of the n variables. $P(\Xi)$ is then the joint probability $P(\boldsymbol{v}_1,...,\boldsymbol{v}_m)$. Assuming that each datum is provided independently according to the probability distribution specified by the Bayesian network $\mathcal{B}$, then

$$P(\Xi) = \prod_{i=1}^{m} P(\boldsymbol{v}_i) \tag{22}$$

The scoring metric is

$$\mathrm{L}(\Xi, \mathcal{B}) = -\sum_{i=1}^{m} \log P(\mathbf{v}_i) + \frac{|\mathcal{B}| \log m}{2}, \tag{23}$$

where $|\mathcal{B}|$ is the number of parameters in network $\mathcal{B}$, and m is the number of samples in the training set Ξ. As an example, using Eqs. (5) and (18), the probability of the first entry in Table 1 is:

$$\begin{aligned} &P(First\ Entry) \\ &\quad = P(U = m, C = st, P = nn) \\ &\quad = \sum_{z} P(m)P(z \mid m)P(st \mid z)P(nn \mid z) \end{aligned} \tag{24}$$

As Z is a hidden variable, the EM algorithm must first be used to fill the conditional probability matrix of node Z, before Eq. (24) can be calculated.

The Bayesian network that has the minimum value for $L(\Xi, \mathcal{B})$ will have the minimum description length. Nilsson describes how to find such a network, using a gradient descent method, which we summarize below:

This method starts with the assumption that there are no conditional relationships between variables (no arcs in the Bayesian network). Arcs are then added or deleted, one by one, or their directions are deleted, and the networks that decrease $L(\Xi, \mathcal{B})$ is chosen. This computation is *decomposable* into computations over each conditional probability matrix in the network. Each total scoring metric is the sum of the local scoring metrics. Every time an arc is added, deleted or reversed, only the local changes need to be taken into account (Nilsson, 1998).

Applied to our three-way aspect model illustrated in Figure 2, these calculations will be as follows:

A local minimum must be found for L, which is:

$$L(\Xi,\mathcal{B}) = -\sum_{u,c,p} \log P(u,c,p) + \frac{|\mathcal{B}| \log m}{2} \tag{25}$$

where $P(u,c,p)$ can be calculated as in Eq. (20), and the number of samples $m = \sum_{i=1}^{8} \text{count}_i$ and the number of parameters $|\mathcal{B}| = 8$ (See Table 1).

Nilsson (1998) describes how the use of hidden variables can reduce the value of L significantly. Hidden variables cannot be observed, and the gradient descent search process must therefore invent them. Hidden nodes that can possibly simplify the causal relationships between variables must be added during the search, and the values of these variables are then "missing values" that must be estimated using the EM algorithm. If the addition of a hidden node decreases the minimum description length of the network, it can be added as a node.

3.7 Bayesian Inference

Bayesian inference is the process of calculating the posterior probability of a hypothesis H (involving a set of query variables) given some observed event (assignments of values to a set of evidence variables e), see Eq. (11) (repeated below).

$$P(H \mid e) = \frac{P(e \mid H)P(H)}{P(e)}$$

Bayesian inference takes the conditional independencies represented by the Bayesian network into account.

Figure 6 illustrates the results of Bayesian inference in the network in Figure 2, in the presence of evidence $e = \{C = st\}$. Node C, the evidence node, is circled. The new posterior probabilities updated during inference are indicated on nodes P, Z and U.

In the presence of the evidence, namely that a user is interested in statistical concepts, an example posterior probability is:

$$P(H \mid e) = P(U = m \mid C = st) = 0.55$$

The posterior probability of the hypothesis that he/she is a mathematician rises from 0.4 in Figure 2 to 0.55. Another example:

$$P(H \mid e) = P(P = nn \mid C = st) = 0.55$$

The posterior probability of the hypothesis that he/she will be interested in a book on neural networks authored by Professor Michael Jordan rises from 0.43 in Figure 2 to 0.55.

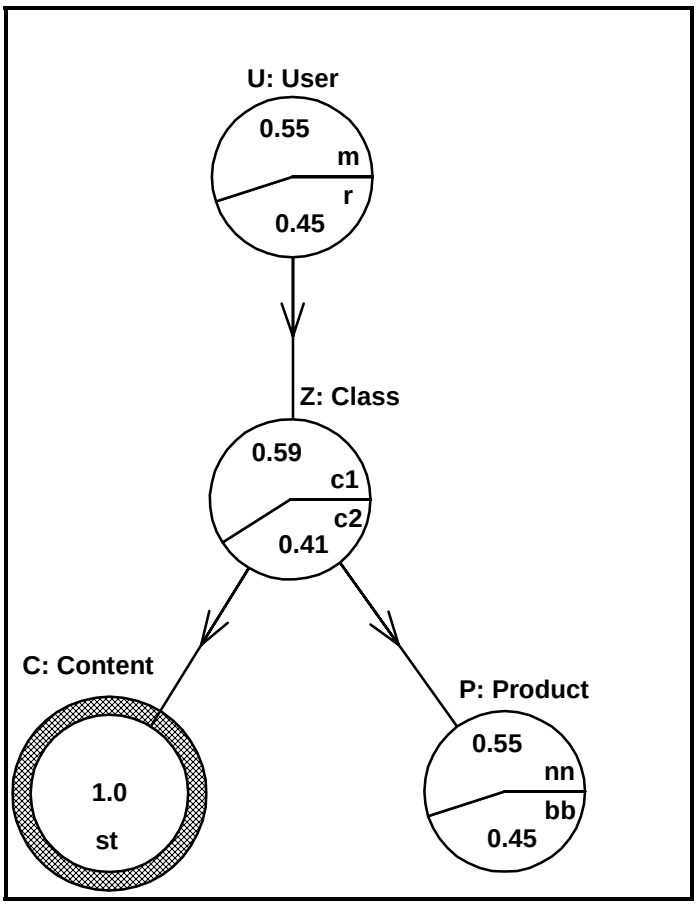

Figure 6: Bayesian Inference Example

Bayesian inference can be either *exact* or *approximate*. In exact inference, the posterior probability of a hypothesis is calculated, but in approximate inference, the posterior probability of a hypothesis is approximated using randomised sampling algorithms.

Bayesian inference can be executed in a *top-down* or *bottom-up* fashion, or using a combination of both. Top-down inference in a Bayesian network is called causal inference, and bottom-up inference is called diagnostic inference.

- *Causal or top-down inference* is the process of determining the probabilities of effects from given causes, that is, when the evidence e is above a given node in the Bayesian network. For example, calculating $P(C = sp \mid U = m)$ in Figure 2 is an example of causal inference involving the calculation of the probability of the effect $C = sp$ from the causal evidence $U = m$ above node *C*.

- *Diagnostic or bottom-up inference* is the process of determining the probabilities of causes from given effects, that is, when the evidence e is below a given node in the Bayesian network. For example, calculating $P(U = m \mid C = st)$ in Figure 2 is an example of diagnostic inference involving the calculation of the probability of the cause $U = m$ from the diagnostic evidence $C = st$ below node *U*.

- *A combination of causal and diagnostic inference* is the process of determining the probabilities of values for a variable from given causes and effects, that is when the evidence e contains values for nodes above and below a given node in the Bayesian network. For example, calculating $P(Z = c_1 \mid U = m, P = nn)$ in Figure 2 is an example of a combination of causal and diagnostic inference involving the calculation of the probability that $P(Z = c_1)$ from the combined causal evidence $U = m$ above node *Z* and the diagnostic evidence $P = nn$ below node *Z*.

Bayesian inference is NP-hard (Pearl, 1988) (Dechter, 1996). In order to simplify inference, Bayesian networks are simplified to trees or singly-connected polytrees. A tree is a DAG in which each node has only one parent (Pearl, 1988). A singly-connected polytree is a DAG in which the nodes can have multiple parents, but with the restriction that there is only one path, along arcs in either direction, between any two nodes in the DAG (Nilsson, 1998) (Pearl, 1988).

Some implementations of exact Bayesian inference use mechanical methods to calculate probabilistic queries about variables given evidence from the environment. Nilsson (1998) describes recursive algorithms that use conditional independencies and Bayes' rule to rewrite the probabilities that must be calculated in terms of the conditional probabilities specified by the Bayesian network. In the variable elimination approach, summations are performed over

variables in order to eliminate them from the network (Pearl, 1988). The elimination approach is based on non-serial dynamic programming algorithms, which suffer from exponential space and exponential time difficulties. Dechter (1996) combined elimination and conditioning in order to address the problems associated with dynamic programming.

Judea Pearl developed a distributed message-passing algorithm for trees (Pearl, 1982) and a general algorithm for singly-connected polytrees called belief propagation (Kim & Pearl, 1983). This algorithm was extended to general multiply-connected networks by different researchers. Two main approaches exist, namely Judea Pearl's cycle-cutset approach and the tree-clustering approach. The cycle-cutset and tree-clustering approaches work well only for sparse networks with small cycle-cutsets or clusters. The belief propagation algorithms for general multiply-connected networks generally have two phases of execution. In the first phase, a secondary tree is constructed. This can for example be a "good" cycle-cutset used during conditioning (Becker, Bar-Yehuda & Geiger, 2000) or an optimal junction tree used by tree-clustering algorithms (Jensen, Jensen & Dittmer, 1994). In the second phase, the secondary tree structure is used for inference. Becker et al. argue that finding a "good" cycle-cutset is NP-complete and finding an optimal junction tree is also NP-complete (Becker & Geiger, 1996). A number of approximation algorithms were developed to find the secondary trees, as in (Becker et al.) (Becker & Geiger).

It is possible to use the original Bayesian network during belief propagation. Diez (1996) describes a conditioning algorithm that uses the original Bayesian network during belief propagation and detects loops using the DFS (Depth-First Search) algorithm.

Approximate Bayesian inference methods are based on randomised sampling algorithms. In approximate inference, approximate answers are given to queries, and the accuracy of these answers depends on the number of samples generated. Russell & Norvig (2003) describe sampling applied to the computation of posterior probabilities, as well as variational approximation methods. Pearl's belief propagation algorithm is an approximation method for general (multiply-connected) networks. The results will not necessarily be correct, and messages might circulate indefinitely around loops, but the values obtained are usually very close to the true values (Pearl, 1988) (Russell & Norvig, 2003).

A detailed study of Bayesian inference and learning falls outside the scope of this research. The prototype implementation uses belief propagation and learning in singly-connected Bayesian networks with known structure. An overview of belief propagation will be given in the next section, and the belief propagation algorithm itself will be described in more detail in Chapter 6.

3.7.1 Belief Propagation

Belief propagation is the process of probabilistic inference in Bayesian networks using message passing. Pearl (1988) describes two forms of belief propagation, namely *belief updating* and *belief revision*.

(Pearl, 1988) describes *belief updating* as the process of

> *propagating the impact of new evidence and beliefs through Bayesian networks so that each proposition eventually will be assigned a certainty measure consistent with the axioms of probability theory,*

and the aim of *belief revision* as

> *not to associate a measure of belief with each individual proposition, but rather to identify a composite set of propositions – one from each variable – that best explains the evidence at hand.*

The composite set of propositions identified during belief revision is called the most probable explanation (MPE) for the variables in the presence of evidence (e).

Dechter (1996) defines the MPE as the maximum assignment x in

$$\max_x P(x) = \max_x \prod_{i=1}^{n} P(x_i | \boldsymbol{pa}(\boldsymbol{x}_i), e) \tag{26}$$

(See Eq. (17) for the meaning of $\boldsymbol{pa}(\boldsymbol{x}_i)$ above).

Belief revision is very complex, and according to Pearl (1988)

> *We do not ordinarily reason that way, though; in trying to explain the cause of a car accident, we do not raise the possibility of lung cancer merely because accidents and lung cancer can both lead to the same eventual consequence – death. Computationally, it appears that in large systems, the task of finding the most satisfactory explanation would require insurmountable effort.*

For this reason, we only consider belief updating in the rest of this thesis. When we refer to belief propagation, we assume belief propagation using belief updating.

Judea Pearl's belief propagation algorithm is a general algorithm for exact inference in singly-connected networks. The goal of this algorithm is to update the marginal probabilities of each node in the network, taking into account the new evidence. Evidence is propagated through the network using π-messages propagated between each parent and its children, and λ-messages propagated between the children of each parent.

In a singly-connected Bayesian network, an arbitrary link XY divides the evidence into the evidence above the link $e^{+}{}_{XY}$, and the evidence below the link, $e^{-}{}_{XY}$, illustrated in Figure 7.

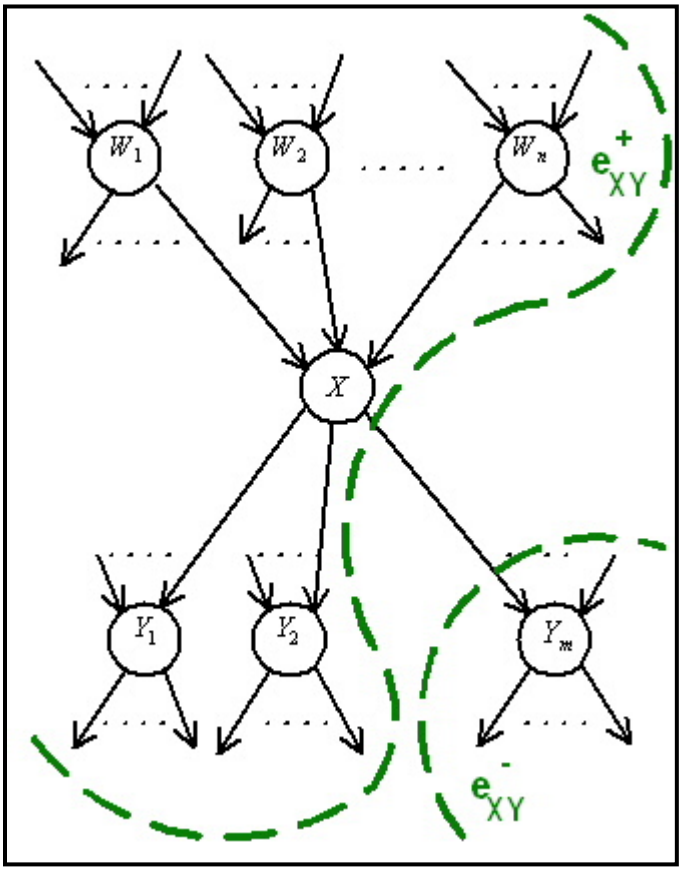

Figure 7: Separation of Evidence by a Link

The π - messages propagated from node X to each of its children Y_i is the probability of each setting of X and the evidence in $e^{+}{}_{XY_i}$ ($P(x, e^{+}{}_{XY_j})$). The λ - messages propagated

from each child node Y_i to its parent X is the probability of the evidence in $e^-_{XY_j}$ given each setting of X ($P(e^-_{XY_j} \mid x)$).

In a singly-connected Bayesian network, an arbitrary node X divides the evidence into that connected to its causes e^+_X (prior evidence) and that connected to its effects e^-_X (observed evidence), illustrated in Figure 8.

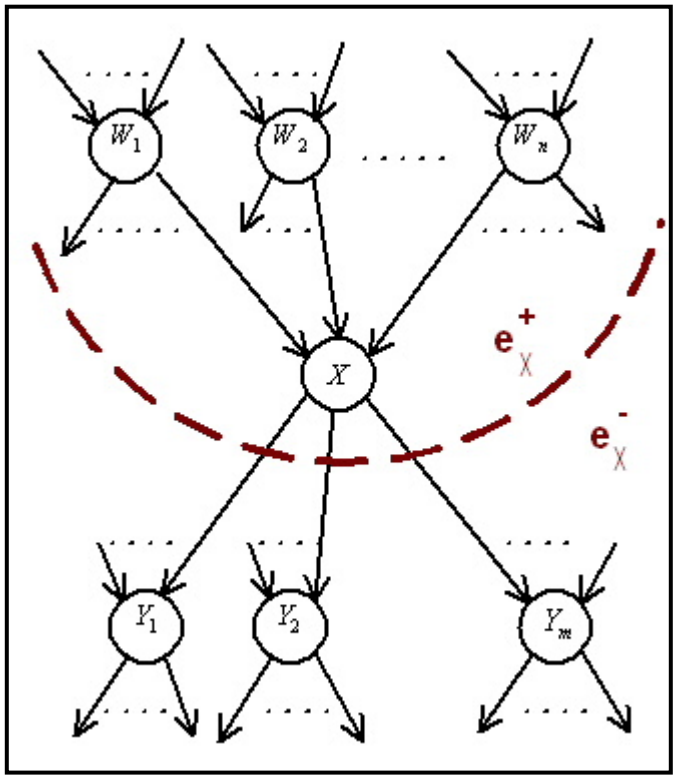

Figure 8: Separation of Evidence by a Node

The prior probabilities vector ($P(x \mid \boldsymbol{e}^+_X)$) of node X is calculated as the product of all the π-messages obtained from the parents of node X, weighted by the conditional probability of node X given its parents. The likelihood vector ($P(e^-_X \mid x)$) of node X is calculated as the product of all the λ-messages obtained from the children of node X. The marginal probability of node X is the normalized product of the prior probabilities vector and the likelihood vector. This probability is referred to as the *belief* of node X.

3.8 Bayesian Networks as Hyperstructures

Bayesian networks are ideally suited to be used as adaptive hyperstructures in internal models. Bayesian learning can be used to adapt to environmental changes, and belief propagation to reason about what action to take next given environmental conditions. A system that uses Bayesian networks in this manner will satisfy Holland's flows and diversity properties and tagging and internal model mechanisms as follows:

1. Flows (Property) – If a Bayesian network is implemented using a set of agents, where each agent implements a network node, and where the agents communicate with each other according to the network links, using messages in correspondence to the belief propagation algorithm, the flows property will be satisfied. The flows are in terms of λ - messages exchanged between children and their parents and π - messages exchanged between parents and their children during belief propagation. The flows will vary over time, depending on evidence received from the environment.
2. Tagging (Mechanism) – Tags facilitate selective interaction. In order to implement the belief propagation algorithm, this mechanism must be implemented. Tagging will enable parents to identify their children in order to send them π-messages and children will be able to select their parents in order to send them λ-messages.
3. Diversity (Property) - Bayesian learning will enable a system to cope with diversity by incrementally discovering regularities in the input stream. Each new adaptation in the Bayesian network can be used to activate different or new actions or interactions.
4. Internal Models (Mechanism) – A system that uses Bayesian networks as hyperstructures in its internal model, will be able to anticipate and predict. The ability of Bayesian learning to discover structure from data, will enable the "mining" of regularities from the input stream and the integration thereof into the Bayesian network topology. The modified internal model will then enable the complex adaptive system to anticipate the consequences that follow when a similar pattern is encountered.

A system that uses Bayesian networks as hyperstructures can only function as a complex adaptive system if it satisfies Holland's other properties and mechanisms as well. In order to achieve this, the agents that implement the Bayesian networks must be organized into adaptive aggregates (aggregation property), having collective behaviour that is more

complicated than would be predicted from summing or averaging their individual behaviours (non-linearity property) and these agents must be implemented using re-usable components (building blocks mechanism).

3.9 Conclusions

Bayesian networks can be used as hyperstructures in the internal model of a complex adaptive system enabling such a system to understand and explain the causal relationships between its own emergent behaviour and the local interactions between agents in the system generating the collective behaviour.

Bayesian learning will enable the system to incrementally "mine" regularities from the input stream, and to encode these regularities into the Bayesian network graph structure constituting the hypergraphs in the internal model. Bayesian learning algorithms are generally decomposable and can be implemented by multiple agents collectively learning from experience.

Belief propagation is a general algorithm for exact inference in singly-connected Bayesian networks. It uses message passing between nodes in a network in order to update the marginal probabilities in the network taking into account new evidence. The belief propagation algorithm exploits the conditional independencies encoded in the Bayesian network graph structure. It uses localized message passing between parents and children and between children and parents in order to propagate beliefs through the network. This algorithm can also be applied to general (multiply-connected) Bayesian networks as an approximation method. The results will not necessarily be correct, and messages might circulate indefinitely around loops, but the marginal probabilities will be very close to the true values.

A system that implements the Bayesian networks in its internal model using agents that can collectively learn using Bayesian learning and that interact with each other in accordance to the belief propagation algorithm, satisfies Holland's flows and diversity properties as well as the tagging and internal model mechanisms. In order to function as a complex adaptive system, these agents must be implemented in such a way that they satisfy Holland's other properties and mechanisms, namely the aggregation and non-linearity properties, as well as the building blocks mechanism.

Chapter 4

Agent Architectures

And you, like me
like any river or creature
like any season or drum
will move any and every day
to a particular rhythm
without even thought to it

We live under the sun,
if we do, and die here
when we do, where all is
collected, collective, and old
as childbirth or death

- Keorapetse Kgositsile

4.1 Overview

A complex adaptive system in nature "moves to" the "rhythms" and regularities in its environment by using agents, organised into adaptive aggregates (agencies), interacting with each other and collectively acting in response to environmental changes. We refer to such a collection of agents and adaptive aggregates as an adaptive agent architecture. There are two other types of (non-adaptive) agent architectures, namely deliberative and reactive agent architectures. An adaptive agent architecture can learn from its environment, whereas deliberative and reactive agent architectures cannot.

An agent architecture consists of one or more agents. These agents are organised into one or more agencies according to the functionality that they collectively achieve. These agencies, in turn, are organised into an organisational structure, usually a hierarchy, depending on their inter-relationships. Even though the concept of an agent is widely used in the research community, there is a lot of confusion on what the term "agent" means. This chapter will define the underlying concepts of agents and agencies, hierarchies and heterarchies and will attempt to create some order out of the terminology chaos. It will then proceed to define what an agent architecture is, and will describe the differences between

deliberative, reactive and adaptive agent architectures. Two representative deliberative and reactive agent architectures are discussed and analysed with respect to Holland's properties and mechanisms.

Adaptive agent architectures are reactive agent architectures that can learn from experience. This chapter describes how learning can be incorporated into a reactive agent architecture in order for it to become adaptive.

4.2 Agents: what Confusion!

Agents are commonly viewed as the next-generation model for engineering complex, distributed systems. There is, however, no consensus in the research community on what an agent is. Some researchers refer to single beings as agents (complex agents), while other researchers refer to independent components within a single being or system as agents (simple agents).

Minsky (1988) first established the concept of simple unintelligent agents combined into intelligent agencies. He describes the mind as a "society" of tiny components that are themselves mindless. He refers to each of these components as agents. His simple agents combine into (sub)societies, called agencies. The agencies are intelligent through the interaction amongst the (unintelligent) agents. According to Minsky (1988), an agent is:

> *Any part or process of the mind that by itself is simple enough to understand - even though the interactions among groups of such agents may produce phenomena that are much harder to understand*

Most researchers use the term "agent" to refer to complex agents rather than to simple agents. Jennings et al. (1998) define an *autonomous agent* as:

> *a computer system,* situated *in some environment, that is capable of* flexible autonomous *action in order to meet its design objectives.*

Maes (1994) defines an agent as:

> *a system that tries to fulfil a set of goals in a complex, dynamic environment. An agent is situated in the environment: it can*

> *sense the environment through sensors and act upon the environment using its actuators.*

According to Maes, such an agent is autonomous if:

> *it operates completely autonomously, i.e. if it decides itself how to relate its sensor data to motor commands in such a way that the goals are attended to successfully*

and adaptive if:

> *it is able to improve over time, i.e. if the agent becomes better at achieving its goals with experience ... i.e. being able to change and improve behaviour over time.*

4.3 Agencies – Order out of Chaos

Simple and complex agents share one common concept, namely the concept of *agency*. Working with the concept of agencies rather than with the concept of agents, can help to create order out of the terminology chaos. In this section, the concepts of agents and agencies are defined, and a description is given of how complex and simple agents share the same concept of agency.

The Oxford Dictionary defines an agent as something *that acts or produces an effect* and an agency as *the function of such an agent*.

A software agent can be defined as:

a software entity that can interact with its environment (adapted from Object Management Group, 2000).

Agents are grouped into agencies, where the definition of an *agency* is:

any *collection of simple agents* considered in terms of what it can accomplish as a unit, without regard to what each of its constituent agents does by itself (adapted from Minsky, 1988).

An agent that can accomplish all its goals independently from other agents forms a single agency consisting of this particular agent as its sole member, having the same functionality as the agent.

Agents that collectively accomplish goals are grouped into agencies according to the functionality that they collectively achieve.

An agency can have one or more of the following characteristics. The following definitions were adapted for agencies from the characteristics of complex agents given by the Object Management Group (2000).

- ***Autonomy*:** the capability to act without direct external intervention. The agency has some degree of control over its internal state and actions based on its own experiences;
- ***Interactivity*:** the ability to communicate with the environment and other agencies;
- ***Adaptivity*:** the capability to respond to other agencies and/or the environment to some degree. Advanced forms of adaptivity permit an agency to modify its behaviour based on its experience;
- ***Sociability*:** the ability to interact in a friendly and pleasant manner;
- ***Mobility*:** the ability to transport itself from one environment to another;
- ***Proxy*:** the ability to act on behalf of someone or something;
- ***Proactivity*:** the capability of goal-oriented, purposeful behaviour. The agency does not simply react to the environment;
- ***Intelligence*:** intelligence of an agency emerges from interactions amongst agents in the agency, and from interactions between the agents and the environment (Brooks, 1991);
- ***Rationality*:** the ability to choose an action based on internal goals and the knowledge that a particular action will bring the agency closer to its goals;
- ***Unpredictability*:** the ability to act in ways that are not fully predictable, even if the initial conditions are known;
- ***Temporally continuous*:** the ability to run continuous;
- ***Character*:** believable personality and emotional state;
- ***Transparency and accountability*:** must be transparent when required, yet provide a log on demand;

- ***Coordinative*:** ability to perform some activity in a shared environment with other agencies;
- ***Collaboration*:** Ability to cooperate with other agencies to achieve a common purpose;
- ***Competitiveness*:** Ability to coordinate with other agencies except that the success of one agency implies the failure of others (opposite of collaboration);
- ***Ruggedness*:** Ability to deal with errors and incomplete data robustly;
- ***Trustworthiness*:** Is truthful.

An agent can belong to more than one agency, if it contributes to the characteristics of different agencies. An agent that is the *sole contributor* to the characteristics of an agency will be the *only member* of that agency.

4.4 Hierarchies and Heterarchies

Agents can be organized into *hierarchies* or *heterarchies*. According to Minsky (1988), a hierarchical organization is:

> *like a tree in which the agent at each branch is exclusively responsible for the agents on the twigs that branch from it*

Heylighen, Joslyn & Turchin (2001) define a heterarchy as:

> *A form of organization resembling a network or fishnet. Authority is determined by knowledge and function.*

Heterarchies are more powerful than hierarchies (Minsky, 1988). We view a hierarchy of agents as a simplified heterarchy of agents. Agencies in turn, can be organized into hierarchies or heterarchies. As with agents, we view a hierarchy of agencies as a simplified heterarchy of agencies.

Potgieter & Bishop (2001) define the relationship between agents, agencies and heterarchies as follows:

> *An **agency** consists of a society of **agents** that inhabit some complex dynamic environment, where the agents collectively sense and act in this environment so that the agency accomplishes what its composite agents set out to accomplish*

> *by interacting with each other. If agents in a society belong to more than one agency, the set of "overlapping" agencies forms a **heterarchy**.*

4.5 Multi-agent Systems

According to Jennings (2001), a collection of interacting autonomous agents forms a multi-agent system. This definition applies to complex agents only.

The Object Management Group (2000) has a more general definition that applies to both simple and complex agents. According to them, multi-agent systems are:

> *systems composed of agents, coordinated through their relationships with one another.*

We prefer to define multi-agent systems in terms of agencies, as follows:

A multi-agent system is a collection of interacting agencies.

4.6 Agent Architectures

4.6.1 What is an Agent Architecture?

Agent architectures are software engineering models of agents (Wooldridge & Jennings, 1995). Agents in an agent architecture are organised into agencies, which in turn are organised into hierarchies or heterarchies.

Agent architectures provide a new and natural way to analyse, design and implement complex software systems (Jennings et al., 1998).

Brooks (1991) identifies four desirable properties of agent architectures, namely *situatedness*, *embodiment*, *intelligence* and *emergence*. We adapted Brooks' definitions of these characteristics in terms of agencies as follows:

- **Situatedness:** The agencies are situated in their environment (the world) – they do not use abstract descriptions of their environment. The state of the environment directly influences the behaviour of the agencies;
- **Embodiment:** The agencies experience their environment directly. Their actions are part of dynamic interaction with the world, and this changes their experience of the world;

- **Intelligence:** The agencies are observed to be intelligent. Their intelligence comes from the state of the environment and how the agencies interact amongst themselves and with the environment;
- **Emergence:** New global behaviours of the system emerge from the interactions between agencies and their environment and from the interactions amongst themselves.

There are three types of agent architectures, namely *deliberative*, *reactive* and *adaptive* agent architectures.

A *deliberative agent architecture* maintains a symbolic internal world model that it manipulates. This world model is a representation of the states of the environment, which is then used by a generalized control strategy during reasoning to determine what actions to take. The popular and widely used Belief-Desire-Intention (BDI) architecture (Rao & Georgeff, 1995) is an example of a deliberative agent architecture. These architectures exhibit varying degrees of situatedness and embodiment depending on the complexity of the symbolic world models they maintain, but these architectures lack emergence.

A *reactive agent architecture* is an agent architecture that reacts and responds directly to the states of the world. These architectures are characterized by situatedness, embodiment, intelligence and emergence (Brooks, 1991). Examples of reactive agent architectures include the Subsumption Architecture (Brooks, 1985) and agents that use behaviour networks (Maes, 1989).

An *adaptive agent architecture* is an agent architecture that can learn from experience in addition to the characteristics identified by Brooks (situatedness, embodiment, intelligence and emergence) – therefore able to function as a complex adaptive system.

In order for an agency to achieve its goals, it must perform an ongoing process of action selection - that is the process of deciding what to do next. Deliberative agent architectures use *deliberative planning* in which an agency constructs a sequence of steps that will guarantee to move an agent from its present state to its goal state. *Reactive planning* in reactive agent architectures chooses only the next action, given the current state of the environment. *Adaptive planning* in adaptive agent architectures chooses only the next action, given the current state of the environment and experience.

The difference between deliberative, reactive and adaptive agent architectures can be described as *knowledge vs. control*. In a deliberative agent architecture, the states of the

environment *are separate from* the control. In a reactive or adaptive agent architecture, the states of the environment *are part of* the control.

4.6.2 Deliberative Agent Architectures

4.6.2.1 Overview

A deliberative agent architecture is an agent architecture that maintains an explicit symbolic model of the world, and it uses logical or pseudo-logical reasoning based on pattern matching and symbolic manipulation for decision-making (Wooldridge & Jennings, 1995). These architectures are by far the most popular and widely used agent architectures.

Wooldridge & Jennings describe the following problems associated with these architectures, namely:

- How to translate the world into and accurate symbolic description in time to be useful (this will determine the degree of *situatedness*); and
- How to symbolically represent information about complex real-world processes and how to reason about this information in time for the results to be useful (this will determine the degree of *embodiment*).

These problems, together with the lack of emergence, make deliberative agent architectures unsuitable for the implementation of complex adaptive systems, although they may be well suited to other complex systems. In the next section, the popular *Belief-Desire-Intention* and *Belief-Desire-Joint-Intention* agent architectures are described. Even though these architectures fall outside the scope of this thesis, they are described as they use well-principled software engineering methodologies. The methodology proposed in this thesis to engineer emergence, extends software engineering methodologies currently applied to deliberative agent architectures as a point of departure.

4.6.2.2 The Belief-Desire-Intention (BDI) Architecture

The Belief-Desire-Intention (BDI) Architecture is a deliberative agent architecture for single complex agents, developed by Rao & Georgeff (1995). This architecture implements a single agency consisting of a single complex agent determining the functionality of the agency. A BDI agent has a mental state consisting of three components, namely beliefs, desires and intentions, representing the information, motivational and deliberative states of the agent. The beliefs represent information about the agent's environment, the desires represent the

different "options" available to the agent and the intentions represent chosen states that the agent has committed resources to (Jennings et al., 1998).

Each agent maintains a behavioural model containing all possible states of the environment, and all possible actions that can be taken by the agent. This model is referred to as the "possible worlds" model. We summarize Rao and Georgeff's description of the possible worlds model below:

The states of the environment and possible actions that an agent can take are modelled by a decision tree that consists of choice nodes, chance nodes and terminal nodes. The choice nodes represent the options available to the system and the chance nodes represent the states of the environment. A probability function maps chance nodes to real-valued probabilities and a pay-off function maps terminal nodes to real numbers. A deliberation function chooses one or more best sequences of actions to perform at a given node. The objectives of the agent are identified by particular paths through the tree structure, each labelled by the objective it realizes and if necessary, a benefit or payoff obtained by traversing this path. This decision tree and its deliberation functions are then transformed into beliefs, desires and intentions as separate relations over the set of possible worlds.

The process of creating the possible world decision trees starts with the full decision tree. The process starts at the root node, and each arc is traversed. For each unique state labelled on an arc emanating from a chance node, a new decision tree is created, identical to the original tree, except that the chance node is removed and the incident arc to the chance node is now connected to the successor of the chance node. This process is carried out repeatedly until there are no more chance nodes left. This process yields a set of decision trees, where each decision tree corresponds to a different possible world with a different probability of occurrence. The payoff function is finally assigned to the appropriate paths.

The resulting possible worlds-model contains two types of information, namely the probabilities across the worlds and the payoff functions assigned to paths. This information is used to define accessibility relations. The probabilities are represented in the belief-accessibility relation and the payoffs in the desire-accessibility relation. Given a decision tree and the above transformations, an agent can use the deliberation function to decide on the best course of action. A third accessibility function can now be defined over the possible worlds, namely the intention-accessibility relation corresponding to the intentions of the agent. For each desire-accessible world, there exists a corresponding intention-accessible world that contains only the best course(s) of action determined by the deliberation function.

The possible world model consists of a set of possible worlds, where each possible world is a tree structure. An index within a possible world is called a situation. With each situation is associated a set of belief-accessible worlds, desire accessible worlds, and intention-accessible worlds. These worlds correspond to what the agent believes to be possible, desires to bring about, and intends to bring about, respectively.

The abstract BDI architecture uses three data structures representing the agent's beliefs, desires and intentions, together with an input queue of events, including internal and external events.

Rao & Georgeff 's main interpreter loop is given below:

BDI-Interpreter
initialize-state();
repeat
  options := option-generator(event-queue);
  selected-options := deliberate(options);
  update-intentions(selected-options);
  execute();
  get-new-external-events();
  drop-successful-attitudes();
  drop-impossible-attitudes();
end repeat;

We summarize Rao & Georgeff's description of the operation of their abstract BDI architecture below:

At the beginning of every cycle, the option generator reads the event-queue and returns a list of options. Next, the deliberator selects a subset of these options that can be adopted and add these to the intentions structure. If there is an intention to perform an atomic action, the agent executes it. Any events that have occurred during the interpreter cycle are then added to the queue. Internal events are added as they occur. Next, the intention and desire structures are modified by dropping all successful desires and satisfied intentions, as well as impossible desires and unrealisable intentions.

Many BDI architectures have been implemented using Rao & Georgeff's abstract BDI architecture as basis. Examples include the Procedural Reasoning System (PRS) and dMARS(distributed MultiAgent Reasoning System) (Rao & Georgeff, 1995). In PRS, an

agent's beliefs are similar to PROLOG facts, its desires are the tasks allocated to it, and its intentions represent the desires that it has committed to achieving (Wooldridge, 1997).

The BDI architecture cannot function as a complex adaptive system, as it does not satisfy all Holland's properties and mechanisms. The following properties and mechanisms are not satisfied by this architecture:

- the aggregation property, tagging mechanism and flows property are not satisfied as this agent architecture has a single agent in a single agency;
- the non-linearity property is not satisfied as the possible worlds model is a static knowledge structure, which cannot exhibit emergent properties;
- the diversity property is not satisfied, as the possible worlds model is static and it cannot be updated as events occur in the environment.

The BDI architecture satisfies the following two of Holland's mechanisms:

- the internal model mechanism – the possible worlds model forms the internal model mechanism of the BDI architecture;
- the building blocks mechanism – the belief, desires, intentions and actions in this architecture can be implemented as re-usable components.

4.6.2.3 The Belief-Desire-Joint-Intention Architecture

One of the agent architectures proposed to overcome the limitations of single complex agents having individual intentions, was the Belief-Desire-Joint-Intention architecture described by Jennings (1993).

Jennings defines a joint intention as

> *A joint commitment to perform collective action while in a certain shared mental state.*

What this means, is that an agency, consisting of more than one complex agent, has a mental state of beliefs, desires and joint intentions, and each individual agent in the agency has individual intentions.

In Figure 9, we illustrate the major processes in the Belief-Desire-Joint-Intention architecture using an object process diagram (OPD)[3], which we constructed from the description in (Jennings, 1993). In Figure 9, the objects are indicated by rectangles and processes by ellipses. We summarize Jennings' description of the processes in the Belief-Desire-Joint-Intention architecture below:

There are two sources of events. Local events can be due to local problem solving or to changes in the environment through the deliberative actions of the agents. Community events occur elsewhere in the society. They can be detected directly through the receipt of a message, or indirectly through perception or deduction. The monitor-event process monitors these events. This process modifies the beliefs of an agent and decides when to raise a new objective. This new objective is used by the means-end analysis reasoning process to decide if this local objective should be met, and how best to realize it. For example, the agent might already have other active intentions that might satisfy the new objective.

The means-end analysis process uses its library of recipes, which are sequences known by an agent for fulfilling particular objects, together with its current intentions. It then reasons if the objective must be pursued locally or in a collaborative fashion. If collaboration is required, a new social act must be established. The agents that must collaborate can be identified by one controlling agent, or by negotiation amongst agents in the community.

The individual actions are then identified using the intentions, joint intentions and capabilities of the potential collaborators. Joint intentions cannot be executed directly, but they are used during problem solving to bind the individual actions of the agents together. At all times, coherency checking is done between individual intentions and joint intentions.

The Belief-Desire-Joint-Intention architecture satisfies the following of Holland's mechanisms and properties:

- the aggregation property – joint intentions group agents into agencies. The agents in these agencies must collaborate to achieve these joint intentions;
- the tagging mechanism and flows property - agents have joint intentions, and they collaborate, communicate and negotiate with each other, using messaging and selective interaction;
- the internal model mechanism – implemented by the possible worlds model;

[3] Object process diagrams are design diagrams proposed for UML 2.0 (Sight Code Inc., 2001)

- the building blocks mechanism – the belief, desires, intentions, joint intentions and actions in this architecture can be implemented as re-usable components.

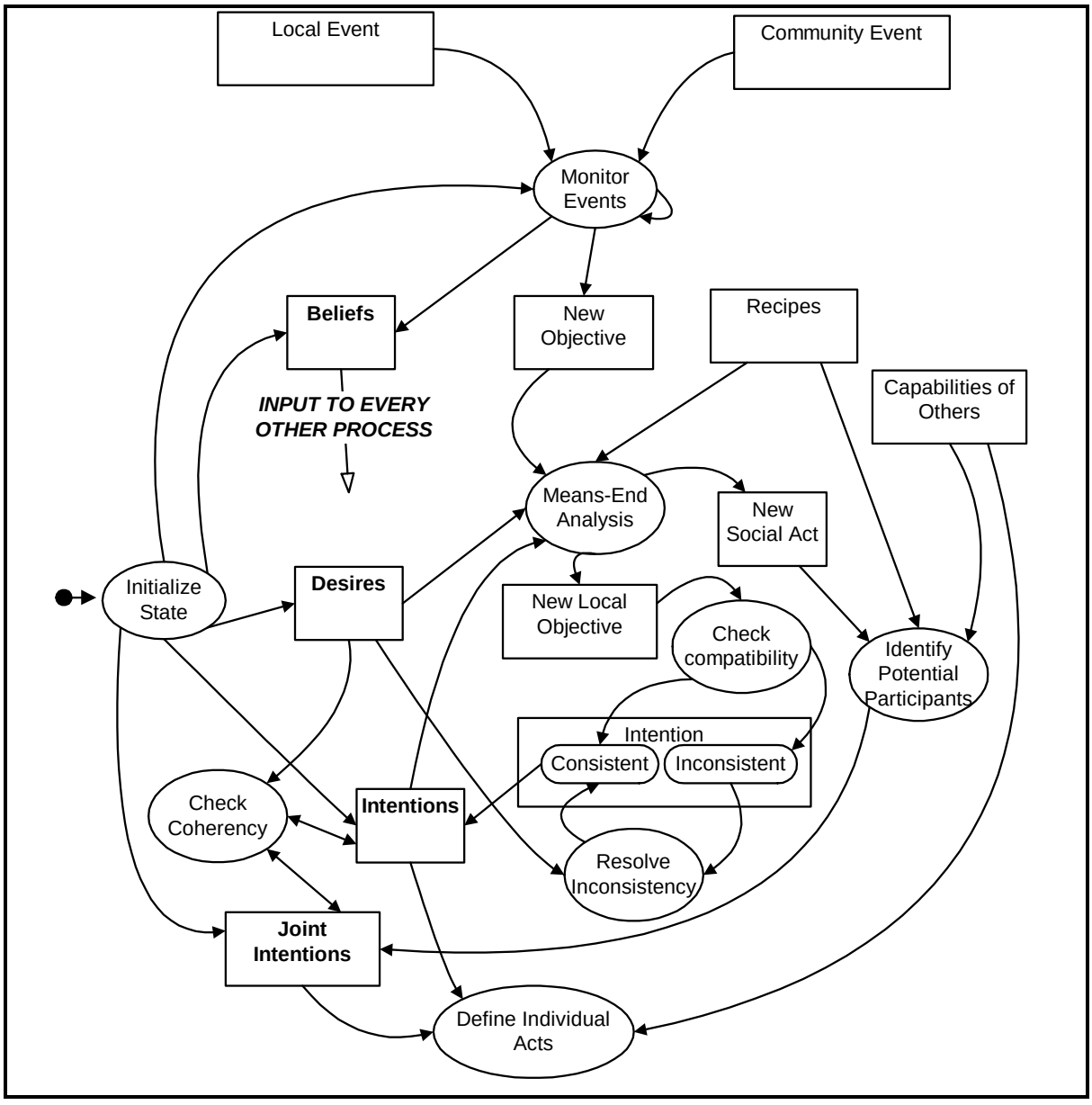

Figure 9: The Belief-Desire-Joint-Intention Agent Architecture

The Belief-Desire-Joint-Intention architecture cannot function as a complex adaptive system. This architecture does not satisfy Holland's non-linearity and diversity properties, as

the possible worlds model is a static knowledge structure that cannot exhibit emergent properties and it cannot be updated as events occur in the environment.

4.6.3 Reactive Agent Architectures

4.6.3.1 Overview

Reactive agent architectures are characterized by *situatedness*, *embodiment*, *intelligence* and *emergence* (Brooks, 1991). The next few sections give a brief overview of two representative reactive agent architectures.

4.6.3.2 The Subsumption Architecture

The Subsumption Architecture (Brooks, 1985) is a reactive agent architecture for single robots. Each robot (complex agent) consists of a set of task accomplishing behaviours as simple agents, organized into multiple nested agencies. Each simple agent is a competence module that is a finite state machine mapping perceptual input to action output (Jennings et al., 1998).

Competence modules are organized into a hierarchy in which the level of abstraction and authority increases from the bottom level upwards (Jennings et al., 1998). Each competence module achieves a competence level that is an informal specification of the desired behaviour of the robot in its environment (Brooks, 1988).

In Figure 10, we represent the competence levels in terms of a state diagram, which we constructed from the description in (Brooks, 1985).

In our state diagram, each competence level is “nested” within in a higher competence level. Each level of competence defines a class of valid behaviours, and each higher level of competence adds additional constraints on the valid behaviours. A higher-level competence module can inhibit the input of a lower level competence module and send its own output as input to the lower level competence module. A higher-level competence module can also suppress the output of the lower level competence module, therefore preventing it from executing - “subsuming” its behaviour (Brooks, 1985).

The complexity of a robot increases with the number of competence levels it has. Developing a robot that exhibits coherent behaviour is a process of carefully developing and experimenting with new competence levels, usually by placing the robot in its environment and observing the results (Jennings et al., 1998). Overall behaviour emerges from the

behaviours of the individual competence modules when the robot is placed in its environment.

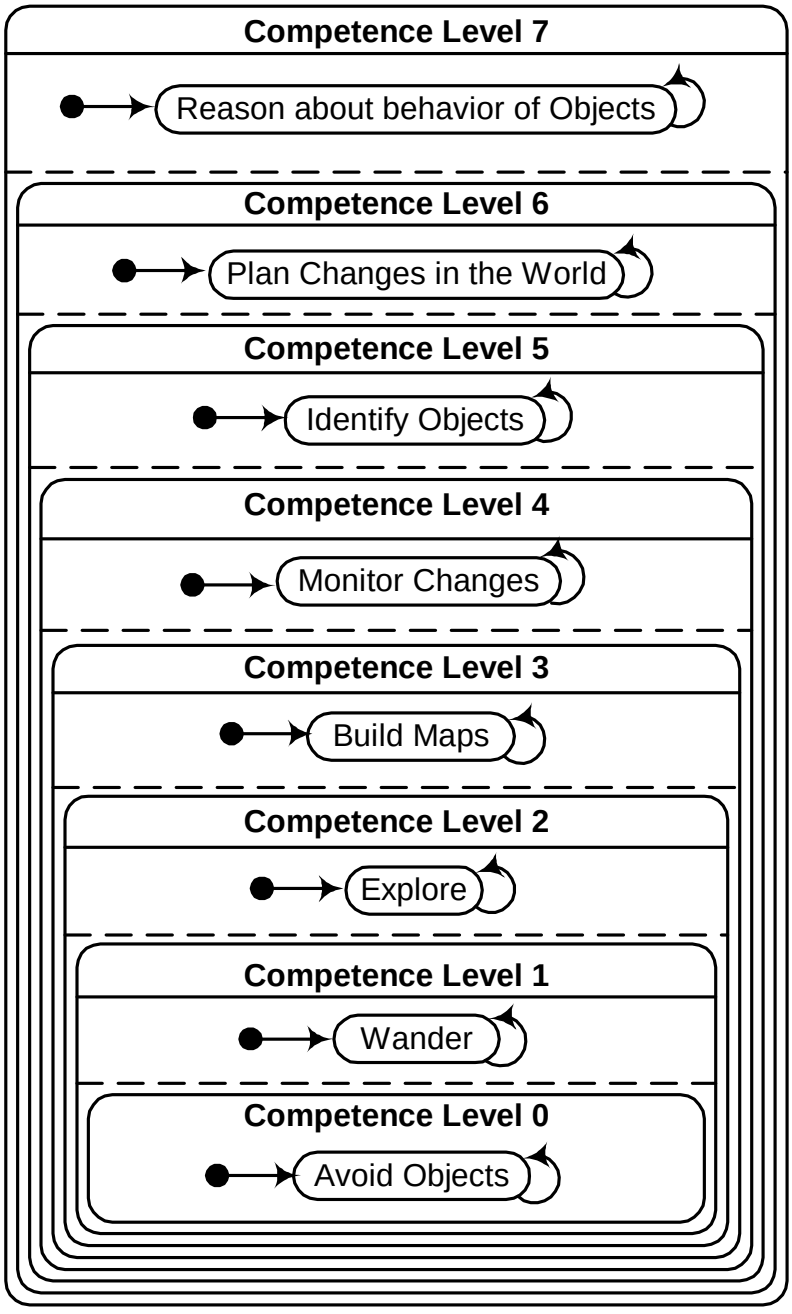

Figure 10: The Subsumption Architecture

Brooks (1985) describes the different levels of competence in the Subsumption Architecture as follows:

- **Competence Level 0:** Avoid contact with moving or stationary objects;
- **Competence Level 1:** Wander aimlessly about without hitting things;
- **Competence Level 2:** "Explore" the world by seeing places in the distance which look reachable, and head for them;
- **Competence Level 3:** Build a map of the environment and plan routes from one place to another;
- **Competence Level 4:** Notice changes in the static environment;
- **Competence Level 5:** Reason about the world in terms of identifiable objects and perform tasks related to certain objects;
- **Competence Level 6:** Formulate and execute plans which involve changing the state of the world in some desirable way;
- **Competence Level 7:** Reason about the behaviour of objects in the world and modify plans accordingly.

The engineering methodology followed in this architecture is a manual process in which the software engineer must analyse the environment as well as the tasks that the competence modules must accomplish. The behaviour of each competence level must be designed as well as the interactions between the competence modules. Goals are implicit, known only to the designer (Maes, 1994). The competence levels are designed and built layer by layer. The process starts with building a complete robot achieving only competence level 0. It is debugged thoroughly, and the next competence level is designed, built and debugged. This process is repeated for the next competence level and so forth (Brooks, 1985).

In Figure 11 we illustrate the interactions between competence levels 0 and 1 in terms of a state diagram, which we constructed from the LISP-like specification of the avoid module defined in (Brooks, 1985).

The Subsumption Architecture uses an internal model consisting of static hyperstructures. The hand-wired arbitration network is a hyperstructure, consisting of a set of suppression and inhibition wires as well as simple arbitration circuitry. These wires form a network connecting competence modules with each other. These hyperstructures have to be handcrafted until the system exhibits the desired behaviour when operating in its

environment. The engineering of emergent behaviours in the Subsumption Architecture consists of a laborious manual process of experimentation, trial and error (Jennings et al., 1998). In a robot with many layers, the emergent behaviour becomes too complex to be understood by the human software engineer (Jennings et al.).

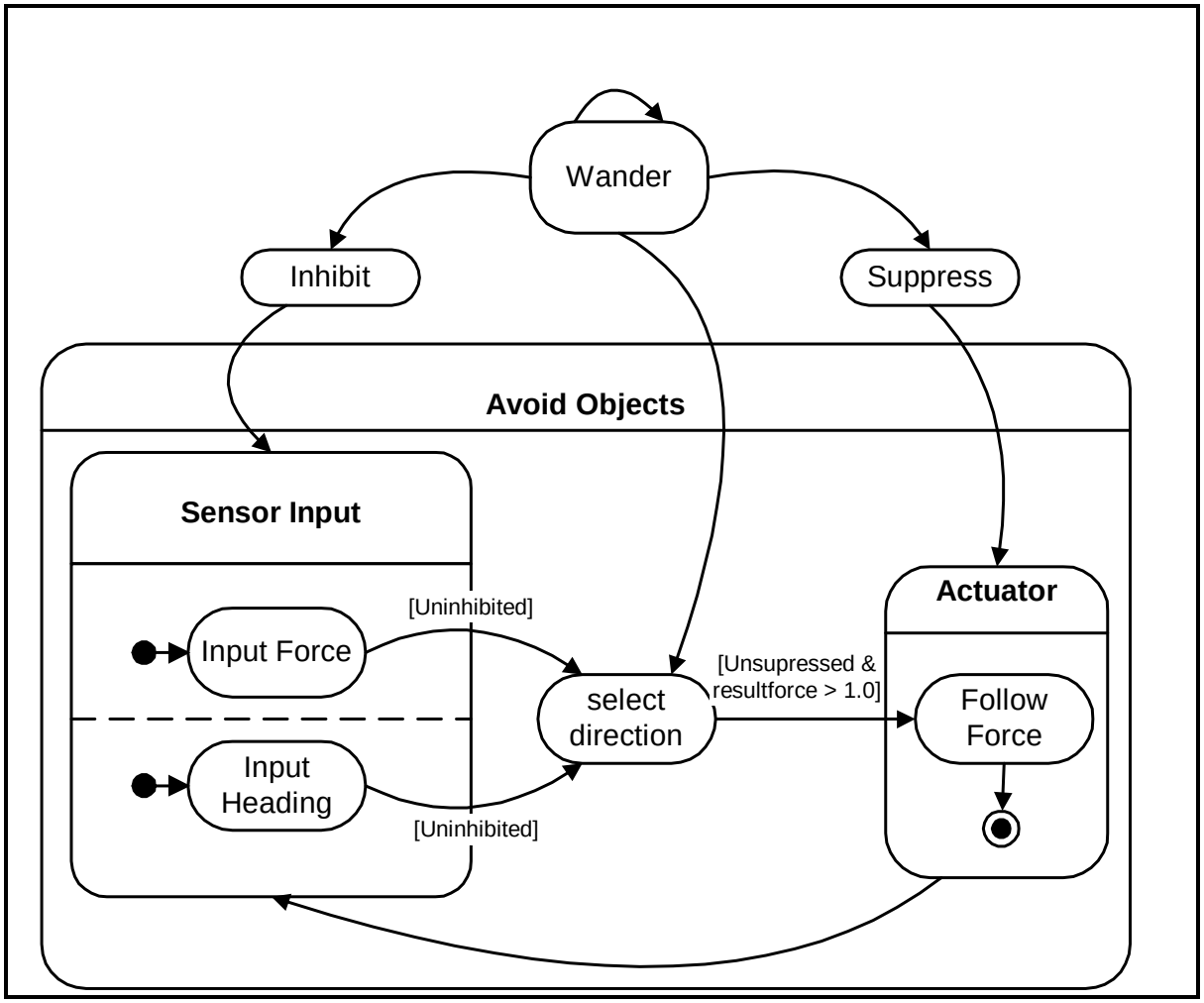

Figure 11: Communication between Competence Levels 0 and 1

The Subsumption Architecture satisfies the following of Holland's properties and mechanisms:

- the aggregation property - the competence modules are aggregated into a hierarchical structure of competence levels;

- the internal model mechanism is implemented by a set of suppression and inhibition wires as well as simple arbitration circuitry. These wires implement the hyperstructures as a network connecting competence modules with each other;
- the building blocks mechanism is implemented as each competence level re-uses the competence modules in the lower competence levels;
- the non-linearity property is satisfied, as emergence is one of the main characteristics of this architecture. The overall behaviour of the robot is emergent and cannot be predicted from the individual behaviours of the competence modules;
- the tagging mechanism and flows property are satisfied by competence modules interconnected through a network of wires, selectively suppressing and inhibiting each other (tagging). The competence modules are nodes interacting with each other through the hand-wired circuitry. The flows in the network vary over time, through constant interaction with the environment.

This architecture does not satisfy Holland's diversity property. It cannot progressively adapt to environmental changes, as it does not support learning. If learning is integrated into this architecture as discussed by Maes & Brooks (1990), this property will be satisfied, enabling this architecture to function as a complex adaptive system.

4.6.3.3 Behaviour Networks

In the reactive agent architecture of Maes (1989) a behaviour network is used to control action selection in response to environmental states. These networks model the relationships between "competence modules". The competence modules react to states of the environment, as well as to the spreading of activation energy along links in the behaviour network.

Figure 12 is an object-process diagram (OPD) for a simple toy robot, which we constructed from the example given in (Maes, 1989). This toy robot has two hands, which it must use to spray-paint itself and sand a board. This OPD represents the competence modules as processes and the states of the environments that will influence these modules are linked by arrows from the states to the competence modules. Arrows from the processes to the states that will be changed indicate the effect of the competence modules on the environmental states. The objects are indicated by rectangles, the states of objects are indicated by rounded rectangles within objects and processes are indicated by ellipses. The states of objects represent propositions about the states of the environment.

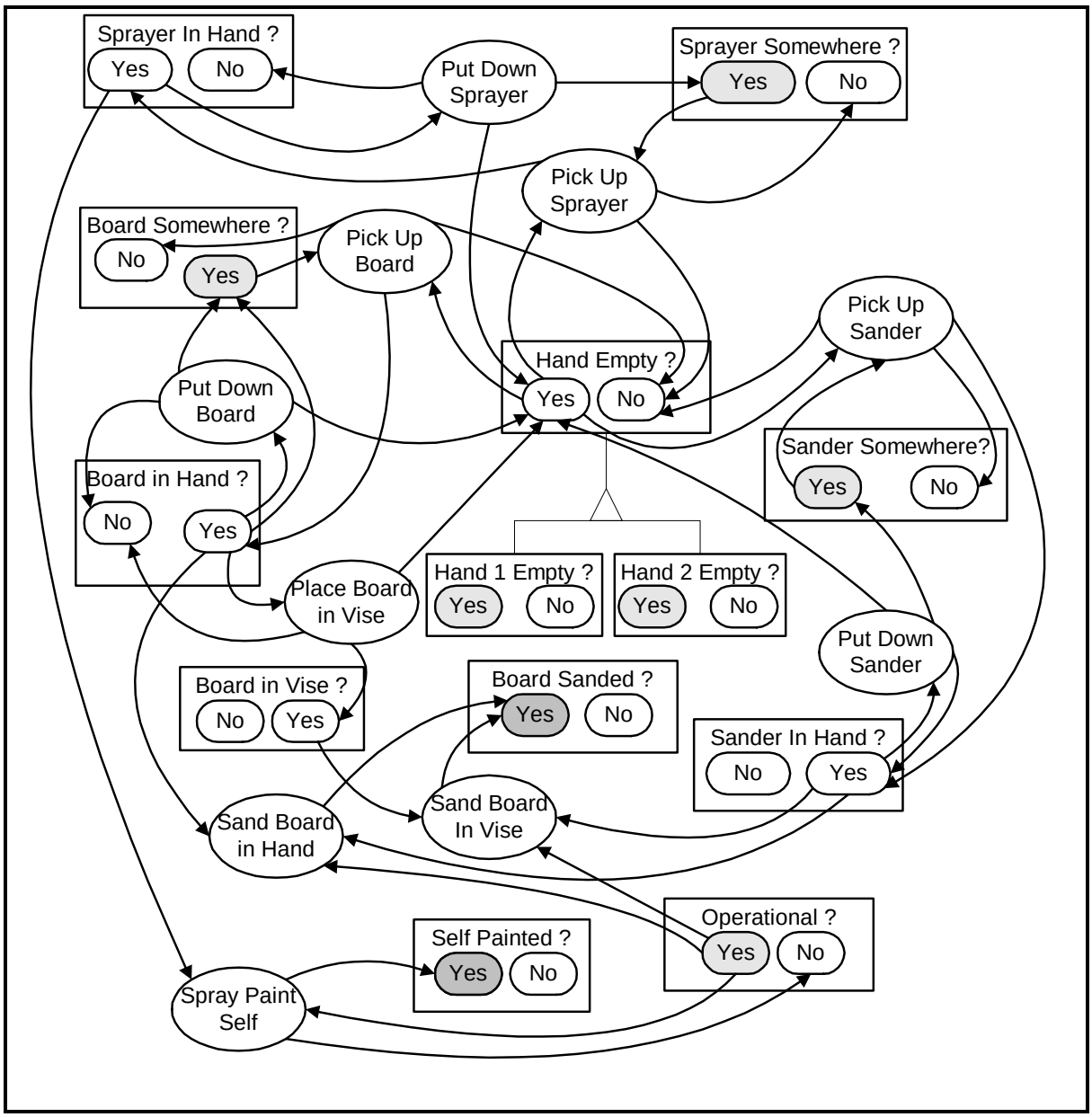

Figure 12: Object Process Diagram for a Simple Toy Robot

The initial state of the environment is represented by the lightly shaded states, namely hand 1 is empty, hand 2 is empty, the sander is somewhere, the board is somewhere and the robot is operational. The goals of the environment are indicated by the dark shaded states, namely that the toy robot must have painted itself and that the board must be sanded.

Each of the competence modules can be viewed as a simple agent, and they are grouped together into a single agency – the autonomous agent.

Maes (1989) describes each competence module i is a tuple $(c_i, a_i, d_i, \alpha_i)$, as follows:

- c_i is a list of preconditions that must be fulfilled before the competence module can become active and execute the behaviour;
- a_i is an add list of predicates which are expected to become true by execution of the behaviour. Some of these predicates can be global goals that an autonomous agent must achieve;
- d_i is a delete list of predicates, which are expected to become false by the execution of the behaviour;
- α_i is the level of activation of the competence module.

A competence module is executable at time t when all its preconditions are observed to be true at time t. The competence modules react to states of the environment, and change the states after their execution.

The software engineer designs an autonomous agent in terms of a behaviour network (also called a spreading activation network), consisting of a number of nodes, representing competence modules, linked together by causal links. In Figure 13 we illustrate the behaviour network for the simple toy robot problem, which we constructed from the toy robot example and the definitions of the successor, predecessor and conflicter links described in (Maes, 1989). We summarize these descriptions below:

Behaviour nodes are linked together by three types of links: successor links, predecessor links and conflicter links. These links represent causal relations among the competence modules. The links between competence modules can be described as follows:

- a successor link from a competence module *x* to every competence module *y* of which a precondition might come true after the execution of module *x*. (We indicated these links by the directional links in Figure 13 – the arrow points to the successor node);
- a predecessor link from a competence module to every competence module that can make a precondition true (For each successor link there is a predecessor link in the

opposite direction as the successor link – we did not indicate these links in Figure 13);

- a conflicter link from a competence module to every competence module that would make a precondition false. (We indicated these links by red dashed lines that end in circles in Figure 13).

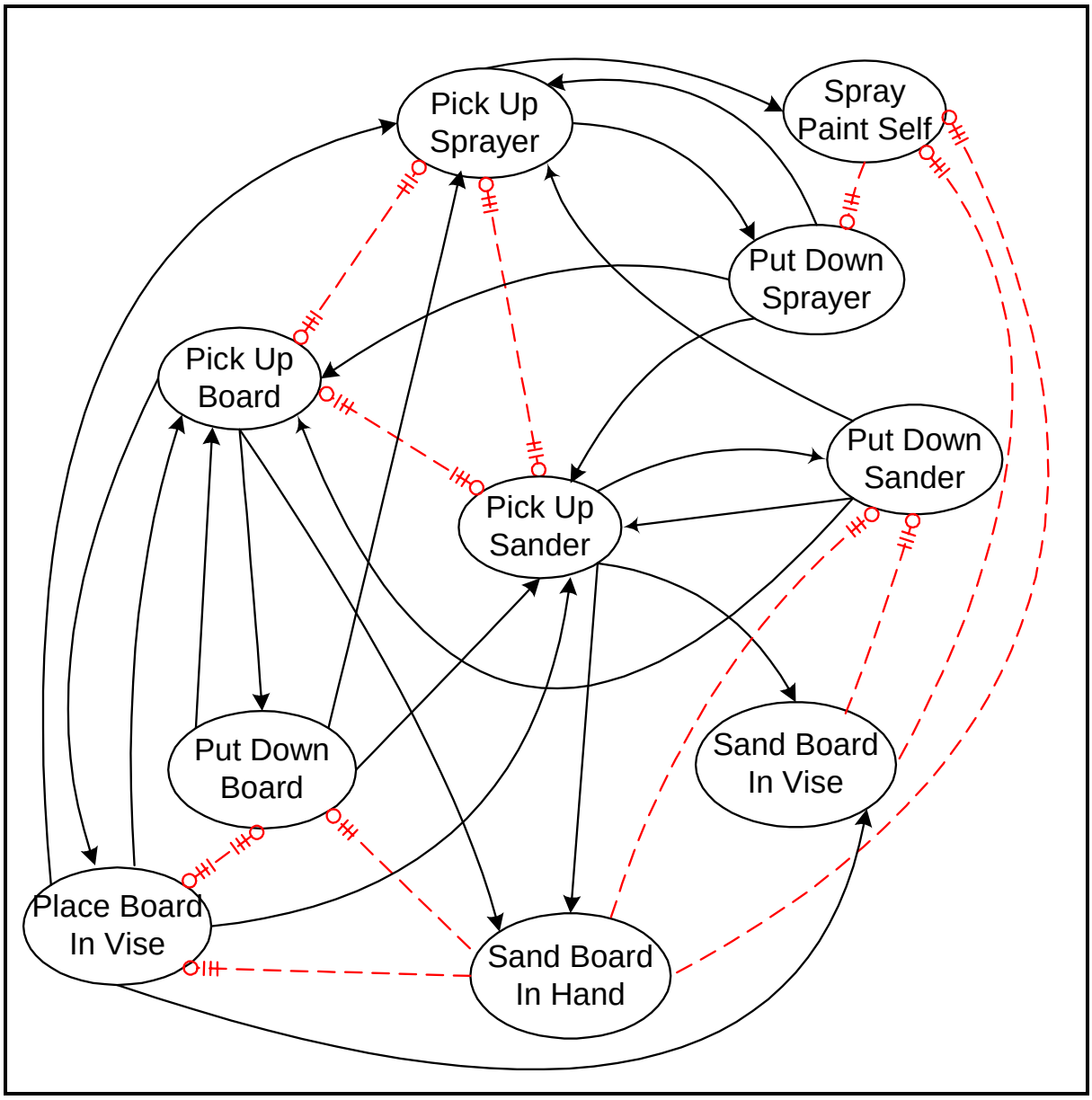

Figure 13: A Behaviour Network

A compiler analyses the behaviour network and generates a circuit that will implement the desired goal-seeking behaviour (Maes, 1994). Once implemented, the competence modules will activate and inhibit each other along the links specified in the behaviour network, so that after some time the activation energy accumulates in the competence modules that represent the "best" actions to take given the current state of the environment and current global goals of the autonomous agent (Maes, 1989) (Maes, 1990).

Once the activation level of a competence module surpasses a certain threshold, and if the module is executable, it becomes active and executes real actions (Maes, 1989). The pattern of spreading of activation and the input of new activation energy into the network is determined by the current state of the environment and the current global goals of the autonomous agent. Maes (1989) describes the spreading of activation, which we summarize below:

- **Activation by the State:** There is an input of activation energy coming from the state of the environment towards modules that partially match the current state. A competence module matches the current state of the environment if at least one of its preconditions is observed to be true;
- **Activation by the Goals:** Each competence module that achieves one of the global goals (if it is a member of the add-list of the competence module) receives activation energy;
- **Inhibition by the Protected Goals:** Each competence module that would undo one of the global goals that has already been achieved (protected goals) gets inhibited – that is some of its activation energy is removed. A competence module will undo a goal if the goal is in its delete list;
- **Activation of Successors:** An executable competence module *x* spreads activation forward to successors *y* along the successor links for which a shared proposition $p \in a_x \cap c_y$ is not true, where a_x is the add-list of *x,* and c_y is the precondition-list of y. This is to make the successor modules more executable since more of their preconditions will become true after competence module *x* has executed;
- **Activation of Predecessors:** A competence module *x* that is not executable spreads activation backward to predecessors *y* for which a shared proposition $p \in c_x \cap a_y$ is not true, where c_x is the precondition-list of *x*, and a_y is the add-list

of *y*. This is to make the predecessors that "promise" to fulfil preconditions that are not yet true more executable after module *x* has executed;

- **Inhibition of Conflicters:** Every competence module *x* (executable or not) decreases (by a fraction of its own activation level) the activation level of the conflicters *y* for which the shared proposition is true. This is to prevent a module that will undo its true preconditions from becoming active.

The algorithm performs in a loop, and in every time step the following computation takes place over all the competence modules, summarized below from (Maes, 1989):

1. Compute the impact of the state of the environment, goals and protected goals;
2. Increase or decrease activation levels through successor, predecessor and conflicter links;
3. Use a decay function to ensure that the overall activation level remains constant.
4. Competence modules that fulfil the following conditions become active:
   - (i) It must be executable;
   - (ii) Its level of activation must surpass a certain threshold;
   - (iii) When two competence modules fulfil these conditions, one is chosen randomly;
   - (iv) If no module fulfils (i) and (ii), the threshold is lowered by 10 %.

The above four steps are repeated infinitely.

The behaviour networks implement the hyperstructures representing regularities in the input stream, identified by the software engineer. These networks control action selection in response to environmental states. These networks model the relationships between "competence modules" that react to states of the environment, as well as to the spreading of activation energy along links in the behaviour network. The networks are specified by the system designer (also the external observer), and then compiled into circuit diagrams, which are then implemented. Once implemented, the structure of the behaviour networks can only be changed by recompilation and re-implementation. For this reason, these hyperstructures are static. Testing if the autonomous agent meets its requirements consists of a process of (human) observation, extensive testing and simulation.

This architecture satisfies the following of Holland's properties and mechanisms:

- the internal model mechanism is implemented by the behaviour network;
- the competence modules can be implemented as re-usable components;
- the non-linearity property is satisfied, as this architecture exhibits emergent behaviour. The spreading of activation energy through the network, in response to environmental states, causes emergent properties. The global behaviour of the agent cannot be predicted from the individual behaviours of the competence modules;
- the tagging mechanism and flows property are satisfied by competence modules interconnected by the behaviour network links. The competence modules can selectively interact with each other through successor links, predecessor links and conflicter links. Activation energy flows through the network in response to environmental changes. The flows in the network vary over time, through constant interaction with the environment.

The following of Holland's properties and mechanisms are not satisfied by this architecture:

- the competence modules are not aggregated;
- the diversity property is not satisfied, as this architecture cannot progressively adapt to environmental changes, as it does not support learning.

If learning is integrated into this architecture as discussed by Maes & Brooks (1990), the last property above will be satisfied.

4.6.4 Adaptive Agent Architectures

4.6.4.1 Overview

An *adaptive agent architecture* is an agent architecture that can function as a complex adaptive system. These agent architectures have the same characteristics as reactive agent architectures, namely situatedness, embodiment, intelligence and emergence with the added characteristic that they can learn from experience.

Reactive agent architectures that cannot learn from experience are adaptive only in a restricted sense in that they are able to deal with unexpected situations, but they cannot learn from environmental feedback. They do not become better at achieving goals with experience

(Maes, 1994). They therefore do not have Holland's diversity property. The next section describes how learning can be implemented into a reactive agent architecture in order to convert it to an adaptive agent architecture.

4.6.4.2 Learning

Maes (1994) describes the following requirements for learning, summarized below:

- Learning should be *incremental* – every experience should contribute to the learning process;
- Learning should be *biased* towards learning knowledge that is *relevant to the goals*. In complex environments, not all possible facts about the environment can be learnt;
- Learning should *cope with noise*, probabilistic environments, etc.;
- Learning should be *autonomous*;
- Learning should be *unsupervised*;
- It must be able to *add prior knowledge* – learning everything from scratch is a timely process.

Maes (1994) further describes three different classes of learning that can be implemented in reactive agent architectures, namely reinforcement learning, classifier systems and model builders, as follows: Given an agent with:

- A set of *actions* it can perform;
- A set of *situations* it can find itself in; and
- A *scalar reward signal* that it receives when the agent does something.

Reinforcement learning involves the learning of an action policy, or a mapping from situations to actions so that an agent that follows that action selection policy maximizes the cumulative discounted reward it receives over time. Q-learning is reinforcement learning in which the agent tries to learn for every situation-action pair what the "value" is of taking that action in that situation (Maes, 1994), (Maes & Brooks, 1990). These techniques have been applied to multi-agent systems as well (Claus & Boutilier, 1998).

Classifier systems can be viewed as a special case of reinforcement learning systems. The agent attempts to learn how it can optimise the reward it receives for taking certain actions in

certain situations. An agent has a set of rules or "classifiers", and data about each rule's performance. The system maintains a "strength" for each rule that represents how "good" that rule is (Maes, 1994).

Model builders learn a causal model of their actions. A probabilistic model is built representing what the effects are of taking an action in a particular situation. This causal model can then be used during arbitration to decide what action is the most relevant given a certain situation and a certain set of goals (Maes, 1994).

Adaptive agent architectures that use Bayesian networks as hyperstructures in their internal models satisfy all of Maes' requirements for learning described above.

4.6.5 A Comparison between the Different Agent Architectures

In Table 2, the different types of agent architectures that we discussed are compared in terms of the Holland's properties and mechanisms.

Agent Architectures	**Deliberative**		**Reactive**	
	BDI	**BDJI**	**SA**	**BN**
Aggregation	X	√	√	X
Tagging	X	√	√	√
Non-linearity	X	X	√	√
Flows	X	√	√	√
Diversity	X	X	X	X
Internal Models	√	√	√	√
Building Blocks	√	√	√	√

Table 2: Comparison between Agent Architectures

In Table 2, the following abbreviations are used:

- BDI – Belief-Desire-Intention architecture;
- BDJI - Belief-Desire-Joint-Intention architecture;
- SA – Subsumption Architecture (no learning capabilities);
- BN – Agent Architecture using Behaviour networks (no learning capabilities).

4.7 Conclusion

Even though there are different interpretations of the meaning of the term "agent", simple and complex agents share one common concept, namely the concept of agency. Agents are grouped into agencies according to the functionality that they collectively achieve and agencies are organised into hierarchies or heterarchies. A heterarchy is more powerful than a hierarchy.

The difference between deliberative, reactive and adaptive agent architectures can be described as *knowledge vs. control*. In a deliberative agent architecture, the states of the environment *are separate from* the control, which make it impossible for these architectures to act upon emergence in the environment. In a reactive or adaptive agent architecture, the states of the environment *are part of* the control, to which these architectures react immediately.

The Belief-Desire-Intention (BDI) and Belief-Desire-Joint-Intention (BDJI) deliberative agent architectures both lack Holland's non-linearity property, due to the absence of emergence in these architectures. These architectures also lack Holland's diversity property due to their inability to learn from experience. The BDI architecture consists of a single complex agent. This architecture only satisfies Holland's internal model and building blocks mechanisms. The BDJI architecture consists of a multiple complex agents joined into agencies by joint intentions, satisfying the aggregation property. It has the same internal model mechanism as the BDI architecture, which can be implemented using re-usable building blocks. The BDJI architecture additionally satisfies the flows property and the tagging mechanism, as the agents communicate with each other and collaborate amongst themselves to achieve their joint intentions.

The Subsumption Architecture (SA) and behaviour networks (BN) reactive agent architectures consist of multiple simple agents organised into one or more agency. The BN architecture lacks the aggregation property, as it is a single agency consisting of multiple simple agents. The SA architecture consists of multiple simple agents, organised into a hierarchy of agencies, therefore satisfying the aggregation property. Both these architectures lack Holland's diversity property due to their inability to learn from experience. A BN architecture that can learn from experience will still lack the aggregation property, but a SA that supports learning will satisfy all of Holland's properties and mechanisms, and will therefore be able to function as a complex adaptive system.

Chapter 5

Agent-Oriented Software Engineering

5.1 Introduction

Agent-oriented software engineering refers to the software engineering methodology followed in an agent architecture. These methodologies currently involve analysis, design and implementation of complex software systems as collections of autonomous agents (Jennings, 2001) and are currently used in deliberative agent architectures to engineer complex systems in such a way that the correct behaviour results.

The next section discusses the tools of the trade, namely decomposition, abstraction and organization, employed by agent-oriented software engineering methodologies in order to address complexity.

Complex agents are commonly viewed as next-generation or "smart" components, and agent-oriented software engineering is viewed as an extension of conventional component-based software engineering. This chapter provides the necessary background to understand component-based software engineering, and then discusses the difference between an object, a component and a complex agent. It then describes examples of component-based agent architectures that use complex agents as next-generation components.

Agent-oriented software engineering currently makes no provision for the engineering of emergence. Complex systems are engineered in such a way that the correct behaviour results. The aim of this research effort is to extend agent-oriented software engineering to include the engineering of emergence. This chapter describes agent-oriented software engineering and gives a brief overview of two representative agent-oriented software engineering methodologies, namely the Gaia methodology and the coordination-oriented methodology.

5.2 Managing Complexity

Jennings (2001) describes the fundamental tools of the trade to help manage complexity, which we summarize below:

- **Decomposition:** The process of dividing a large problem into smaller, more manageable chunks, each of which can then be dealt with in relative isolation. This helps to manage complexity as it limits the designer's scope;

- **Abstraction:** The process of defining a simplified model of the system that emphasizes some of the details or properties while suppressing others. This process limits the designer's interest at a given time;
- **Organization:** The process of defining and managing the interrelationships between the various problem-solving components. This enables the designer to group basic components together, which is treated as a higher-level unit of analysis. It also provides a means of describing the high-level relationships between various units.

Agent-oriented software engineering manages complexity using the tools described above. These methodologies address decomposition using components or agents and abstraction by defining software engineering models at different levels of abstraction. These models include detailed specifications of the components, the interactions between different components as well as organizational relationships between components.

Agent-oriented software engineering extends existing paradigms such as the object-oriented and component-based software engineering approaches. According to Griss & Pour (2001), an agent is a

> *Proactive software component that interacts with its environment and other agents as a surrogate for its user, and reacts to significant changes in the environment.*

Griss & Pour (2001) refer to agents as "next-generation" components and to agent-oriented software engineering as an extension to conventional component-based software engineering approaches. The next section will give a brief overview of component-based software engineering, and subsequent sections will describe what the differences are between component-based and agent-oriented software engineering.

5.3 Component-based Software Engineering – A Brief Overview

Component-based software engineering uses component-based design strategies. A *design strategy* can be viewed as an architectural style consisting of high level design patterns described by the types of components in a system and their patterns of interaction (Bachman et al., 2000). Bachman et al. defined a reference model for component-based concepts, which we summarize below, using Figure 14:

A *component* (1) is a software implementation that can be executed on a physical or a logical device. A component implements one or more *interfaces* that are imposed on it (2). By doing this, the component satisfies certain obligations, called a *contract* (3). These contractual obligations ensure that independently developed components obey certain rules so that components can interact (or not interact) in predictable ways, and can be deployed into standard run-time environments (4). A component-based system is based upon a small number of distinct component-types, each of which plays a specialized role in a system (5) and is described by an interface (2). A *component model* (6) is the set of component types, their interfaces, and additionally, a specification of the allowable *patterns of interaction* among component types. A *component framework* (7) provides a variety of runtime services (8) to support and enforce the component model.

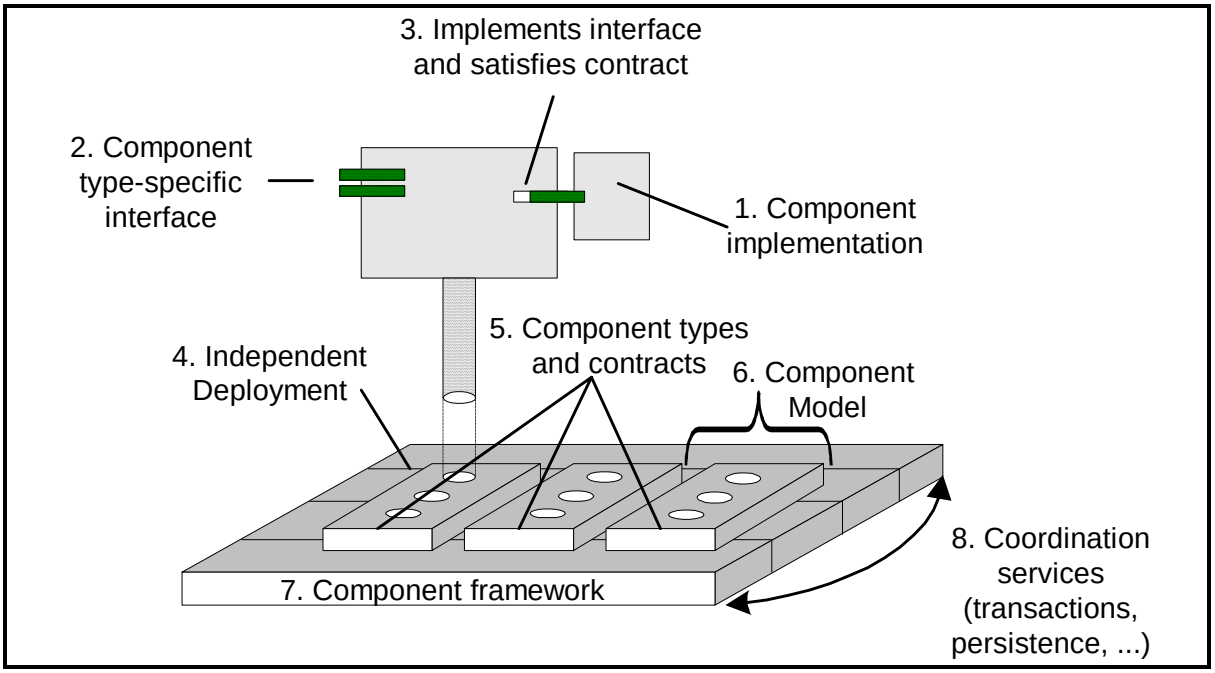

Figure 14: The Component-Based Design Pattern (Bachman et al., 2000)

Hopkins (2000) defines a component as follows:

> *A software component is a physical packaging of executable software with a well-defined and published interface.*

In the current UML specification, UML1.4 (UML Revision Task Force, 2001), a component is described as follows:

> *A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.*

Hopkins further identifies the engineering drivers in the development of a component-based system as:

- **Reuse:** The ability to reuse existing components to create a more complex system;
- **Evolution:** A componentized system is easier to maintain. In a well-designed system, components can be changed without affecting other components in the system.

Components publish their interfaces and communicate with each other within component models such as Microsoft's DCOM (Distributed Component Object Model), the Object Management Group's CORBA (Common Object Request Broker Architecture) and Sun's Enterprise JavaBeans.

5.4 A Comparison between Objects, Components and Complex Agents

5.4.1 Overview

As complex agents are commonly viewed as "next-generation components", we need to understand the similarities and differences between objects, components and complex agents.

5.4.2 Similarity between Objects, Components and Complex Agents

Components, objects and agents have *identity,* their own *state* and *behaviour* (Object Management Group, 2000). Components and agents have interfaces through which they can communicate with each other, and with other entities.

5.4.3 Difference between Objects, Components and Complex Agents

Components within complex agents have been developed using components as simple agents (micro level). The research community, however, is experiencing difficulties in implementing complex agents in their entirety using conventional component-based

approaches. Jennings (2001) describes the differences between objects, components and complex agents, which we summarize below:

- Objects are generally passive in nature – they need to be sent a message before they become active;
- Although objects encapsulate state and behaviour, they do not encapsulate behaviour activation (action choice), as is the case in complex agents. An object can invoke any publicly accessible method on any other object. Once the method is invoked, the actions can be performed;
- Abstraction mechanisms such as design patterns and components and frameworks are still insufficient for the development of (complex) agent systems;
- Object-oriented approaches provide only minimal support for specifying and managing organizational relationships as static inheritance hierarchies define these relationships.

The next section discusses component-based agent architectures based on complex agents as next-generation components.

5.5 Component-Based Agent Architectures

Griss & Pour (2001) describe a component-based agent architecture illustrated in Figure 15.

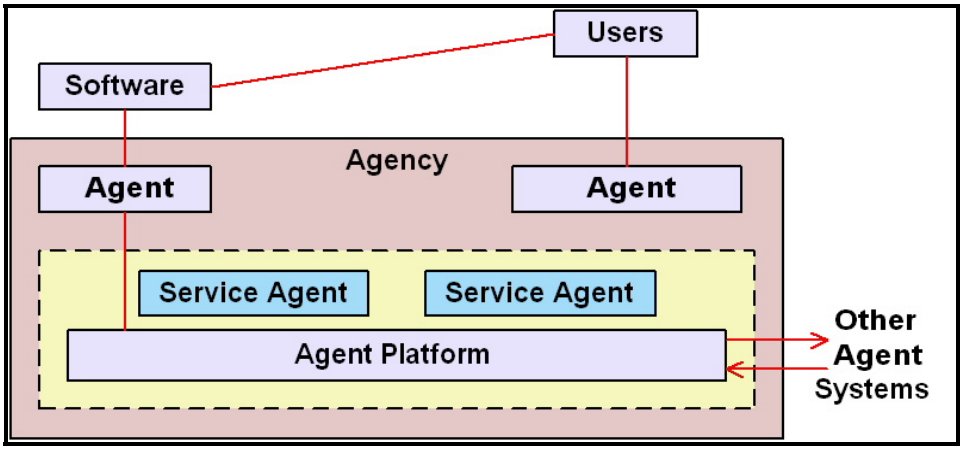

Figure 15: Component-Based Agent Architecture (Griss & Pour, 2001)

We summarize Griss & Pour's description of this architecture below:

In this component-based architecture, complex agents are grouped into agencies according to conceptual and physical locations. These agencies have the responsibilities to locate and send messages to mobile and detached agents and to collect knowledge about groups of agents. The agency's core is the agent platform, which is a component model that provides local services for agents and includes proxies to access remote services such as agent management, security, communication persistence and naming. The agent platform provides agent transport for mobile agents as well as specialized agents that reside in other, possibly remote agencies. The agent infrastructure can be augmented by standard service agents, such as broker agents, auctioneer agents, and so forth. The agent platform and the service agents can monitor and control message exchanges between agents, and detect any violation of rules of engagement.

Agents interact with each other using interaction standards such as agent-communication languages (ACL). These languages are declarative and define the overall structure and interaction pattern between agents. The agent communication language is associated with the component model, and messages are broken up into independent parts, namely message type, addressing, context and content. Message parts can describe the domain; others can describe an expected conversation pattern. The component model manages the messages and the agents. Example agent communication languages include KQML (Finin, Fritzson, McKay & McEntire, 1994) and FIPA's ACL (Foundation for Intelligent Physical Agents [FIPA], 2000).

Agent architectures that use agent-communication languages to interact, use different approaches to identify components for re-use. Skarmeas & Clark (1999) describe the use of an active message-board for inter-component (inter-agent) communication. This message-board forwards messages to other components depending on the message content. The components in this agent-architecture are illustrated in Figure 16. These components include domain dependent components, the agent head, a meta component, a message board and a knowledge base.

The agent head communicates with other agents or non-agent applications. Circles in Figure 16 indicate domain dependent components. These components implement the behaviours or actions that an agent must execute. The meta-component manipulates and reconfigures the domain dependent components, depending on environmental conditions.

The knowledge base component maintains information shared by all the other components. This includes, for example, the beliefs, intentions and plans of the agent, together with

knowledge of other agents, and the capabilities of the domain dependent or behavioural components.

Agents communicate with each other through the shared knowledge maintained by the knowledge base component, as well interacting through messages using the active message board component. Each agent registers itself on the message board, and "advertises" itself on this messages board by sending active message patterns to the message board. "White Pages" are maintained with agent identifications, and "Yellow Pages" are maintained containing advertisements. Incoming messages are forwarded to all applicable agents by consulting the "Yellow Pages" and the "White Papers". A receiving agent can reply directly to a sender agent, or via the message board.

The agent head component deals with all incoming messages. It fulfils a security function, and places messages on the message board to be forwarded to the appropriate agent components, depending on the message contents.

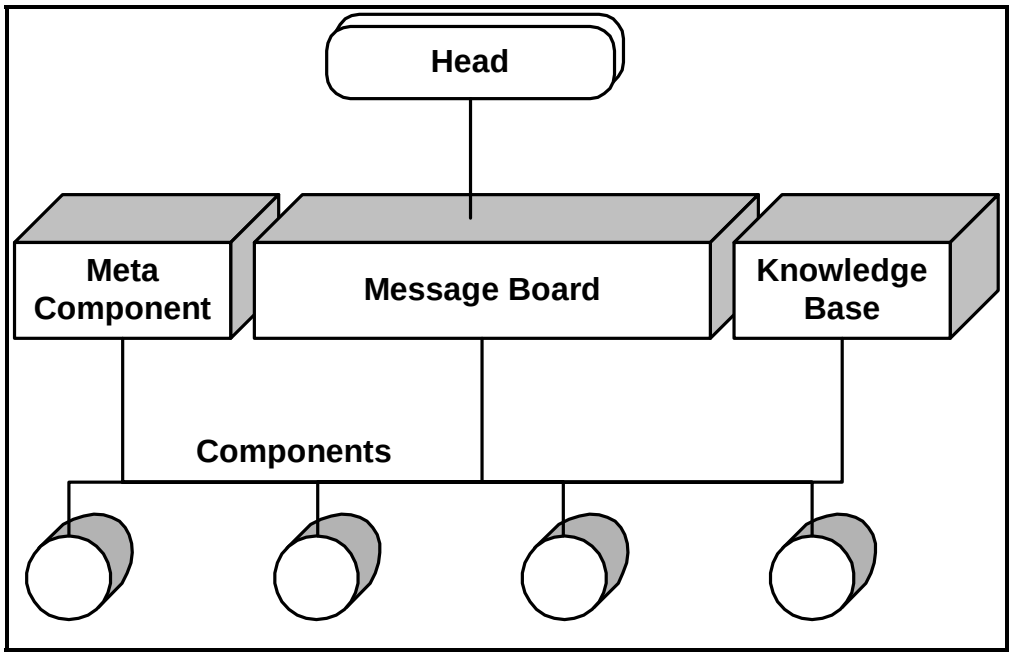

Figure 16: Internal Agent Components (Skarmeas & Clark, 1999)

5.6 Agent-Oriented Software Engineering Methodologies

5.6.1 Overview

Abstraction, the focus on relevant details while ignoring others, forms an important part of agent-oriented software engineering. With increasing complexity in systems, visualization, modelling and the use of well-principled methodologies are essential. Rumbaugh first applied this way of modelling and design to object-orientation with his well-known Object Modelling Technique (OMT), using three models, namely an object model, a dynamic model and a functional model (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991).

Agent-oriented software engineering involves the modelling of different aspects of the agent-based system, at different levels of abstraction. Different approaches construct different models, but they all have the same end-goal in sight, namely addressing decomposition, abstraction and organization using a set of models. Each of these models provides a different view of the agents in the system and the interactions between them, at different levels of abstraction.

UML (Unified Modelling Language) defined by the Object Management Group (OMG) has become widely accepted as a visual modelling tool for object-oriented and component based systems. As agents are commonly viewed as next generation objects or components, a number of researchers are exploring ways to extend UML in order to make it more suitable to agent-oriented software analysis and design. One example is Agent UML or AUML (Odell, Van Dyke Parunak and Bauer, 2001), which will make UML more suitable to the specification of agent interactions and interaction protocols.

Current agent-oriented software engineering methodologies do not make provision for the engineering of emergence. These methodologies attempt to prevent emergence, rather than to engineer it.

The next sections describe two agent-oriented software engineering approaches, namely the Gaia and coordination-oriented software engineering methodologies.

5.6.2 The Gaia Methodology

The Gaia methodology (Wooldridge et al., 2000) is an agent-oriented software engineering methodology that involves a process of generating increasingly detailed models during the analysis and design phases. This approach is not based on any particular agent architecture. The main goal of the Gaia analysis phase is to understand the system and its structure – that

is, understanding the organization of the system. Wooldridge et al. (2000) define an organization as:

> *A collection of roles, that stand in a certain relationships to one another, and that take part in systematic, institutionalized patterns of interaction with other roles.*

Classical design aims to transform abstract models derived during analysis into a sufficiently low level of abstraction so that they can easily be implemented, whereas the Gaia design phase aims to transform the analysis models into a sufficiently low level of abstraction so that traditional object-oriented techniques can be used to implement the agents (Wooldridge et al., 2000).

The Gaia methodology was intended for use in closed systems of distributed problem solvers, in which a fixed number of complex agents collaborate to achieve a global goal (Zambonelli et al., 2000). This methodology attempts to control emergence by engineering it before the fact. It uses interaction protocols in order to specify all possible interactions between the fixed number of complex agents.

The Gaia models are described in more detail in Chapter 7.

5.6.3 Coordination-Oriented Methodology

Zambonelli et al. (2000) claims that most current agent-oriented methodologies are ill suited to open environments, as they do not adequately address agencies. They addressed this problem by developing a coordination-oriented methodology suited to open environments, which makes provision for global laws that agents in an agency must obey when interacting with other agents. This methodology tries to anticipate all potential actions that autonomous agents might take. They impose restrictions on the interactions, which they call "social laws". These "social laws" ensure that agents that adhere to these laws need not worry about undesirable interactions no matter what goals or plans they adopt (Durfee, 2001), therefore restricting emergence.

Zambonelli et al.'s models consist of three elements, namely

- ***the coordinables:*** entities whose mutual interaction is ruled by the model (the agents);

- ***the coordination media:*** the abstractions enabling agent interactions, as well as the core around which components are organized. Examples include semaphores, monitors, channels and blackboards;
- ***the coordination laws:*** the behaviour of coordination media in response to interaction events.

Zambonelli et al. describe two classes of coordination models, namely *data-driven* and *control-driven* behaviour models. We summarize these two models below:

In *data-driven* coordination models, coordinables interact with the external world through the exchange of data-structures through the coordination media. The coordination media acts as a shared dataspace. The coordinables request data, either reading it or extracting it from the coordination media. The coordination laws determine how datastructures are represented and how they are stored, accessed and consumed. The coordination media handles the interaction space through entities interacting using data exchange and synchronization over data occurrences (Zambonelli et al.).

Data-driven coordination models can be applied to deliberative agent architectures. Skarmeas & Clark (1999) describe the use of an active message-board for inter-component (inter-agent) communication. This message-board is the coordination media, forwarding messages to other components (coordinables) depending on the message content using coordination laws.

In *control-driven* coordination models, the coordinables interact with each other and the environment through well-defined input/output ports, representing the coordination media. The observable behaviours of the coordinables are in terms of state changes and events occurring at these ports. The coordination laws establish how events and state changes occur and how they propagate through the coordination media. The coordination media handle the interaction space by controlling how the events that occur in the environment connect to the coordinables and how it propagates through the system. There is no concern for the data exchanged between the coordinables (Zambonelli et al.).

Control-driven coordination models can be extended in order to engineer reactive agent architectures such as the Subsumption Architecture (Brooks, 1985) and autonomous adaptive agents using behaviour networks. (Maes, 1989).

Zambonelli's coordination models are described in more detail in Chapter 7.

5.7 Conclusion

Agent-oriented software engineering uses decomposition, abstraction and organization to engineer complex systems. Decomposition is achieved by using a component-based approach or an extension of a component-based approach. Current agent-oriented software engineering approaches achieve abstraction by defining software engineering models at different levels of abstraction. These models include detailed specifications of the complex agents, the interactions between different agents, as well as the organizational relationships between agents.

The research community is experiencing difficulty in implementing complex agents in their entirety using conventional component-based frameworks. This is due to the fact that these frameworks use objects and components that are passive in nature and that do not encapsulate behaviour activation and because the organizational relationships in these frameworks are static inheritance hierarchies. Re-usable components have been developed for use within complex agents, such as the internal agent components used by Skarmeas & Clark (1999).

Agent-oriented software engineering has so far only been applied in deliberative agent architectures. The Gaia approach (Wooldridge et al., 2000) is applicable to closed systems of deliberative agents in which a fixed number of agents collaborate to achieve a global goal. Zambonelli et al. (2000) extended this methodology to open environments with their coordination-oriented methodology, in which the deliberative agents must adhere to social laws. These methodologies currently attempt to control emergence by engineering it before the fact. This is done by restricting the interactions between the agents by using interaction protocols (Wooldridge et al., 2000) and imposing social laws (Zambonelli et al., 2000).

The coordination-oriented methodology can either be data-driven or control-driven. Data-driven control models can be applied to deliberative agent architectures as described by Skarmeas & Clark (1999). Control-driven models can be extended so that it can handle emergence in order to engineer reactive and adaptive agent architectures.

Chapter 6

BaBe: An Adaptive Agent Architecture

6.1 Overview

An adaptive agent architecture implements a complex adaptive system. Such an architecture has the four properties identified by Brooks (1991), namely situatedness, embodiment, intelligence and emergence. These agents are situated in the world (situatedness). Their actions must be part of a dynamic interaction with the environment (embodiment). They are observed to be intelligent (intelligence). The intelligence of agencies emerges from the interaction between the simple agents and their environment and between themselves and other agents (emergence). In addition to these characteristics, such an agent architecture is able to learn from experience.

An adaptive agent architecture can be constructed from simple agents, organised into agencies that collectively learn from and adapt to their environments. If these agents have the same functionality, they can be implemented as re-usable components. In this research, we implemented the BaBe agent architecture using three re-usable components, together with a set of behavioural components. The three re-usable BaBe components are used to assemble distributed Bayesian behaviour networks that incrementally learn from the environment and activate component behaviours depending on environmental states.

We refer to the Bayesian networks used in BaBe as Bayesian behaviour networks. These networks are used in a similar way as the behaviour networks defined by (Maes, 1989). Behaviour networks, however, allow for causal modelling using Booleans only, whereas Bayesian behaviour networks allow for powerful probabilistic reasoning in the presence of uncertainty. We use Judea Pearl's belief propagation algorithm for Bayesian inference in these networks. This chapter describes our Bayesian behaviour networks, and gives the necessary background on the belief propagation algorithm.

We assemble a Bayesian behaviour network using the BaBe components. These components are simple agents, organised into agencies, which in turn are organised into a heterarchical structure. These agencies are adaptive aggregates that collectively learn from and act in their environments. This chapter describes and illustrates the Bayesian agencies in our prototype implementation. We further describe how these agencies can be applied in a

web personalization example and we analyse our BaBe architecture with respect to Holland's properties and mechanisms.

6.2 Bayesian Behaviour Networks

We define a *Bayesian behaviour network* as a Bayesian network that models the regularities in the input stream of an adaptive agent architecture. The nodes in such a Bayesian behaviour network are grouped into what we call *competence sets*, where each competence set has an associated set of actions that must be performed depending on the states of the nodes in the competence set.

In Figure 17, we illustrate a simple Bayesian behaviour network, born out of my frustration whilst seeking for papers or books authored by Professor Michael Jordan, the well-known statistician, and ending up with hundreds of papers or books on Michael Jordan, the well-known basket-ball player.

Figure 17 illustrates a fictitious model of the browsing behaviour of users visiting an electronic bookstore website. This network models the relationships between the type of user that browses the site (*A*), their interests (*B*), the sequence of hyperlinks that they clicked to access the pages (*C*), content categories of all the pages on the website (*D*), the information content of the advertisements on the web pages (*E*), the pages they view (*F*), the pages that they will visit next (*H*) and the buying behaviour per page (*G*).

Our example website has hyperlinks to the following pages:

- Page 1: books by Judea Pearl on causality and probabilistic reasoning;
- Page 2: books by Professor Michael Jordan on graph theory and probability theory;
- Page 3: books by / related to Michael Jordan, the well-known basketball player;
- Other Page: any other page on the website.

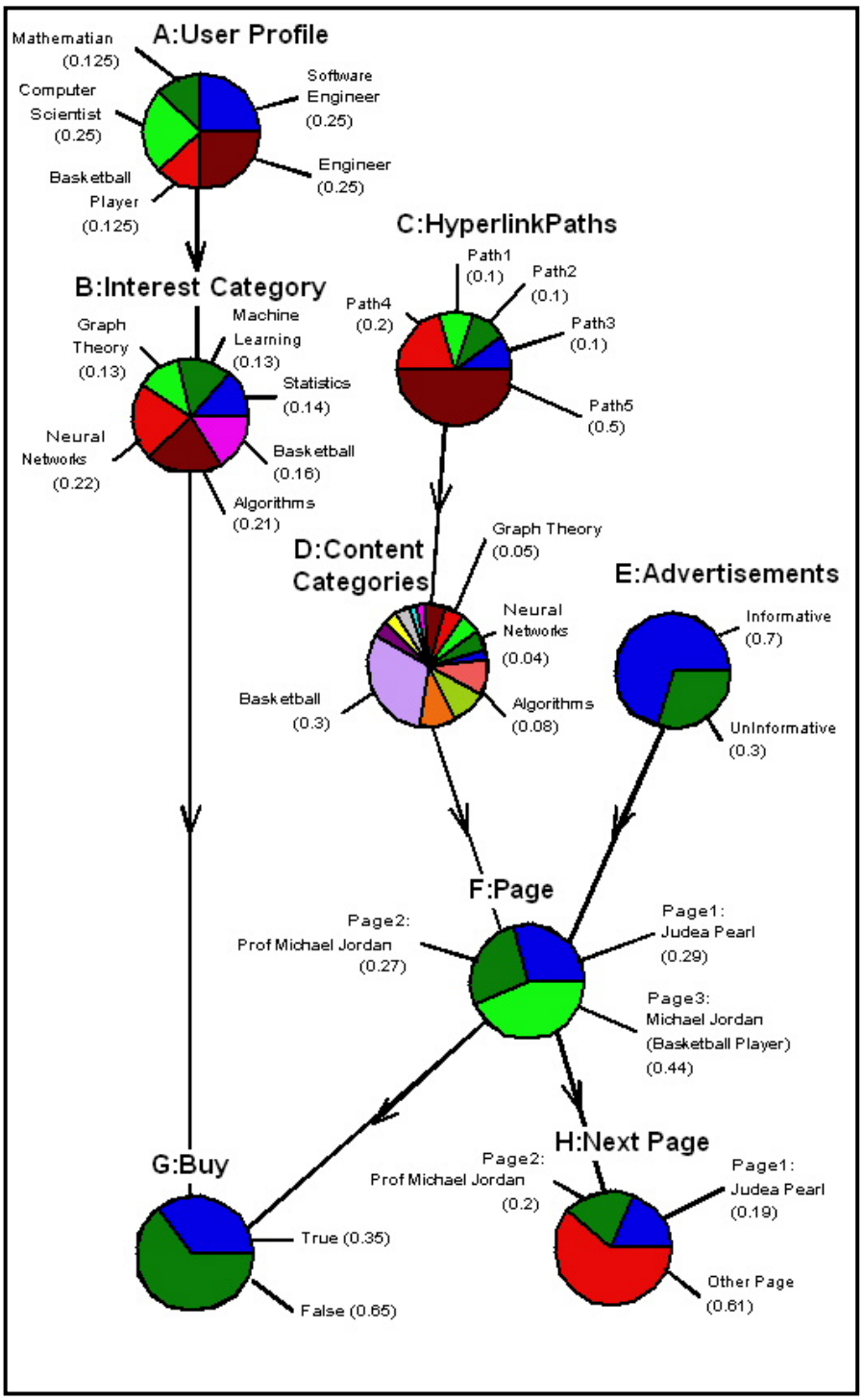

Figure 17: A Bayesian Behaviour Network

The hyperlink paths to these pages are:

- Path 1: Engineering and Science → Mathematics → Graph Theory → Page 1 or 2;
- Path 2: Engineering and Science → Mathematics → Probability and Statistics → Page 1 or 2;
- Path 3: Computers and Internet → Artificial Intelligence → Machine Learning → Neural Networks → Page 2;
- Path 4: Computers and Internet → Programming → Software Engineering → Algorithms → Page 1 or 2;
- Path 5: General Interest → Sports and Adventure → Basketball → Page 3.

Each node has a conditional probability table, for example, the conditional probability matrix associated with node *B* is presented in Table 3.

P(*B*:InterestCategory | *A*:UserProfile)					
***B*: Interest Category**	***A*:UserProfile**				
	Engineer	**Mathematician**	**Computer Scientist**	**Software Engineer**	**Basketball Player**
Graph Theory	0.1	0.25	0.1	0.2	0.01
Statistics	0.2	0.34	0.1	0.1	0.01
Machine Learning	0.1	0.1	0.25	0.1	0.01
Neural Networks	0.3	0.2	0.25	0.2	0.01
Algorithms	0.25	0.1	0.25	0.3	0.01
Basketball	0.05	0.01	0.05	0.1	0.95

Table 3: Conditional Probability Matrix for Node B

In our simple network in Figure 17, buying behaviour (*G*) depends on the current page that is being browsed (*F*), and the categories of interest of a particular user (*B*), which in turn depends on the user profile (*A*). For simplicity, users are profiled on their profession only. The website contents are categorized into content categories (*D*), which are distributed between different pages (*F*). In order not to clutter the diagrams, only a few content categories are indicated next to node *D*. The choice of a page (*F*) depends on how well its contents matches the content categories (*D*) that the user is looking for and how well the content categories were advertised to the user (*E*). The content categories (*D*) that a user is looking for is related to the hyperlinks (*C*) to the page that the user is browsing. The relationship between the current page (*F*) that is being viewed and the next page (*H*) that will most probably be browsed next is also modelled in this network.

The marginal probabilities (beliefs) are indicated next to each of the nodes in Figure 17. For example, the beliefs of the user profile node (*A*) indicate that mathematicians and basketball players browse this site with equal probability of 0.125. The beliefs of the hyperlink paths node (*C*) indicate that Path 5 will most probably be chosen (0.5) and the beliefs of the content categories (*D*) indicate that the basketball category is most likely to be searched for (0.3). The beliefs of the page node (*F*) show that the Michael Jordan (the well-known basketball player) page will most probably be viewed (0.44). The beliefs of the advertisements node (*E*) show that the advertisements that led the user to this page were informative with a probability of 0.7. The beliefs of node (*G*) show that the probability that a user will buy a book when visiting a page is 0.35. The probability that a user would be interested to view books by Michael Jordan (the professor) next is 0.2.

Figure 18 illustrates the results of Bayesian inference in the presence of evidence. A mathematician that browses a website listing books by Judea Pearl (the evidence) is most probably interested in statistics (0.34), graph theory (0.25) and neural networks (0.2). He would have chosen hyperlink path 4 with the highest probability (0.28) in order to search for algorithms related to his field if interest. He will buy a book from this page with a probability of 0.55. The probability that this user will be interested to view books by Professor Michael Jordan next has now risen to 0.6 and the probability that the advertisements were informative has now increased to 0.74.

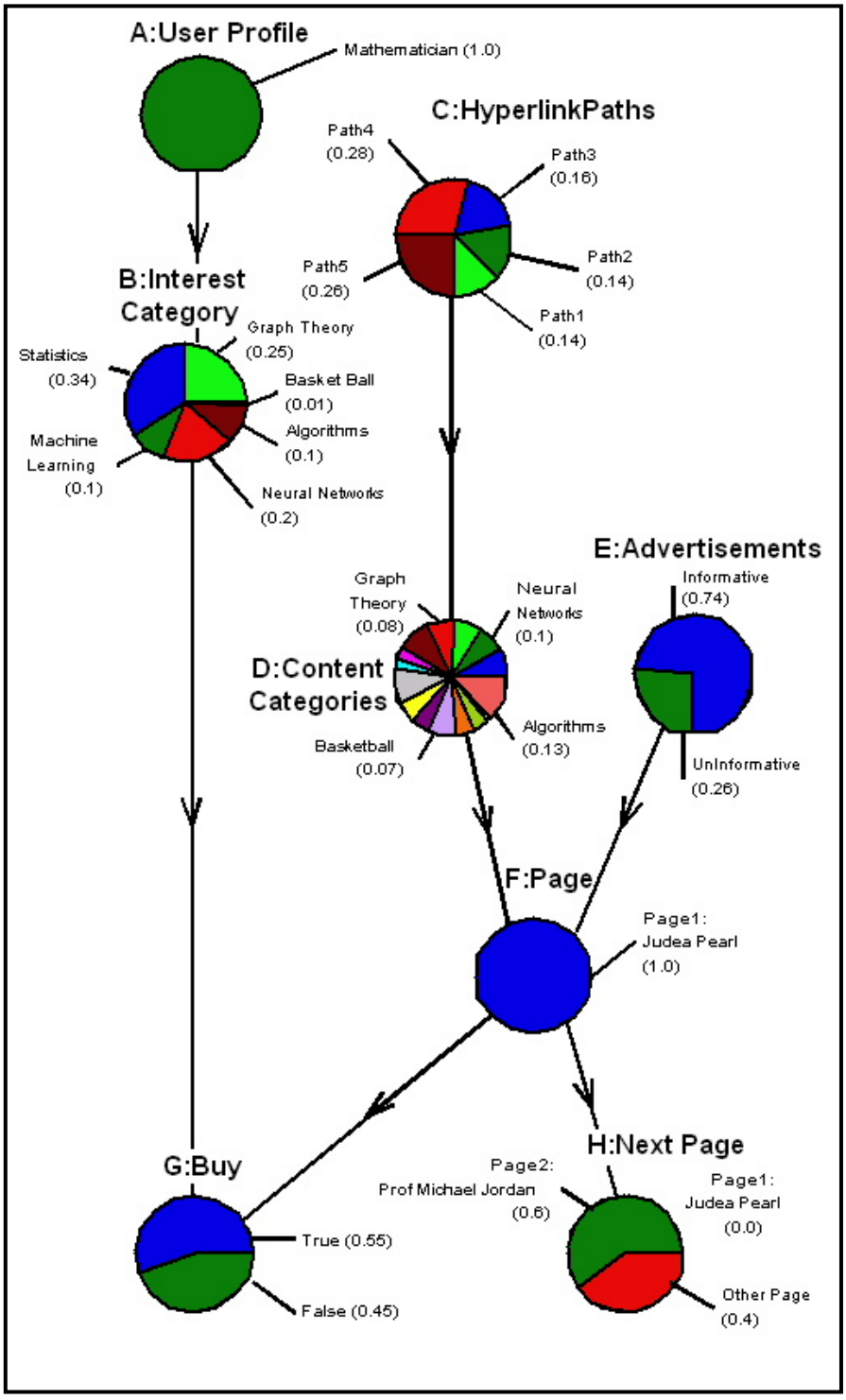

Figure 18: Belief Propagation in a Bayesian Behaviour Network

6.3 Competence Sets

We define a *competence set* Θ_i as a tuple (C_i, A_i), where C_i is a set of constraints on a subset of nodes and their states in a Bayesian behaviour network, and A_i is the set of actions that must be executed if all the constraints in C_i are met.

A possible collection of competence sets for Figures 17 and 18 is as follows:

Let $\Theta = \{\Theta_1, \Theta_2, \Theta_3, \Theta_4\}$, where

- Θ_1 = {{}, {the set of actions associated with the personalization of the web pages depending on the beliefs of nodes *B* and *D*}. This set specifies that the specified actions must be executed unconditionally as there are no constraints in this set;
- Θ_2 = {{}, {the set of actions associated with the dynamic creation of hyperlinks to web pages, depending on the beliefs of nodes *C* and *D*}}. This set specifies that the specified actions must be executed unconditionally as there are no constraints in this set;
- Θ_3= {{the belief that the advertisements were uninformative (node *E*) must be greater than 0.4}, {inform the marketing department how informative advertisements were – display the beliefs of node *E* and how it influenced the buying (the beliefs of node *G*)}}. This set specifies that the specified actions must be taken if the belief that the advertisements were uninformative exceed a threshold of 0.4;
- Θ_4= {{}, {the set of actions associated with the displaying of links to web pages that might interest the user next, depending on the beliefs of node *H*}}. This set specifies that the specified actions must be executed unconditionally as there are no constraints in this set.

Using Bayesian behaviour networks, systems can "observe" their environmental states as well as their own behavioural states and adapt their behaviours accordingly.

6.4 Pearl's Belief Propagation Algorithm

We use Judea Pearl's belief propagation algorithm for inference in our Bayesian behaviour networks. In a singly-connected Bayesian network, that is a network without loops, an arbitrary node X divides the evidence into that connected to its causes $e^+{}_X$ (prior evidence) and that connected to its effects $e^-{}_X$ (observed evidence). A link XY divides the evidence into the evidence above the link, $e^+{}_{XY}$ and the evidence below the link, $e^-{}_{XY}$. All the evidence is denoted by e.

In Judea Pearl's belief propagation algorithm, the evidence is propagated through a Bayesian network using messages propagated between each parent and its children, and between each child and its parents. The messages propagated in a network, as described by Pearl (1998), are given by Eqs. (27) to (31), and illustrated in Figure 19.

The prior probabilities vector is the summarized effect on the belief of X by prior evidence e^+, and is represented by:

$$\pi(x) \equiv P(x \mid \boldsymbol{e}^+{}_X) = \sum_{w_1, w_2, \ldots w_n} P(x \mid w_1, w_2, \ldots, w_n) \prod_{i=1}^{n} \pi_X(w_i) \tag{27}$$

The likelihood vector is the summarized effect on the belief of X by diagnostic (or observed) evidence, e^-, and is represented by:

$$\lambda(x) \equiv P(e^-{}_X \mid x) = \prod_{j=1}^{m} \lambda_{Y_j}(x) \tag{28}$$

The summarized effect of evidence above link XY_j is represented by:

$$\pi_{Y_j}(x) \equiv P(x, e^+{}_{XY_j}) = \pi(x) \prod_{k \neq j} \lambda_{Y_k}(x) \tag{29}$$

The summarized effect of evidence below link XY_j is represented by:

$$\lambda_{Y_j}(x) \equiv P(e^-{}_{XY_j} \mid x) = \sum_{y_j} \left[\lambda(y_j) \sum_{v_1, \ldots v_p} P(y_j \mid x, v_1, v_2 \ldots v_p) \prod_{k=1}^{p} \pi_{Y_j}(v_k) \right] \tag{30}$$

where $V_1, \ldots, V_p$ are causes of Y_j other than X.

The belief of node X is represented by:

$$BEL(x) \equiv P(x \mid e) = \alpha\pi(x)\lambda(x) \tag{31}$$

where α is a normalization constant to be computed after finding $\pi(x)$ and $\lambda(x)$.

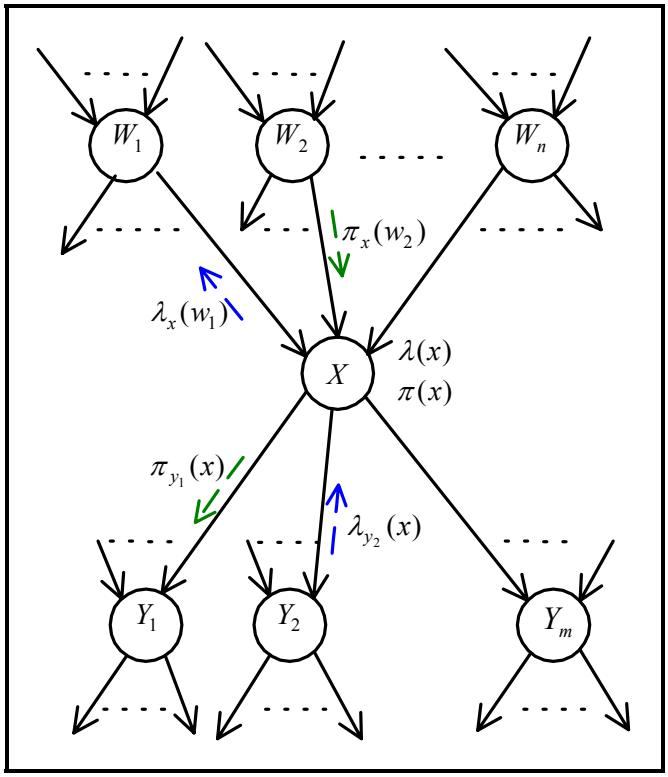

Figure 19: Belief Propagation

Pearl (1988) describes the boundary conditions for belief propagation as follows:

1. *Root nodes*: If node X has no parents, then $\pi(x)$ is the prior probability $P(x)$.
2. *Anticipatory nodes*: If node X has no children, then $\lambda(x) = (1,1,\ldots 1)$.
3. *Evidence nodes*: If evidence X=x' was received for any node X in the network (not necessarily a leaf node) then $\lambda(x) = (0,,\ldots 1\ldots 0)$ with 1 at the x'-th position.

6.5 The BaBe Components

We assemble our Bayesian behaviour networks from BaBe components. These components include node components, link components, belief propagation agents grouped into belief

propagation agencies, and competence components grouped into competence agencies. The components, agents and agencies in the BaBe architecture are illustrated in Figure 20.

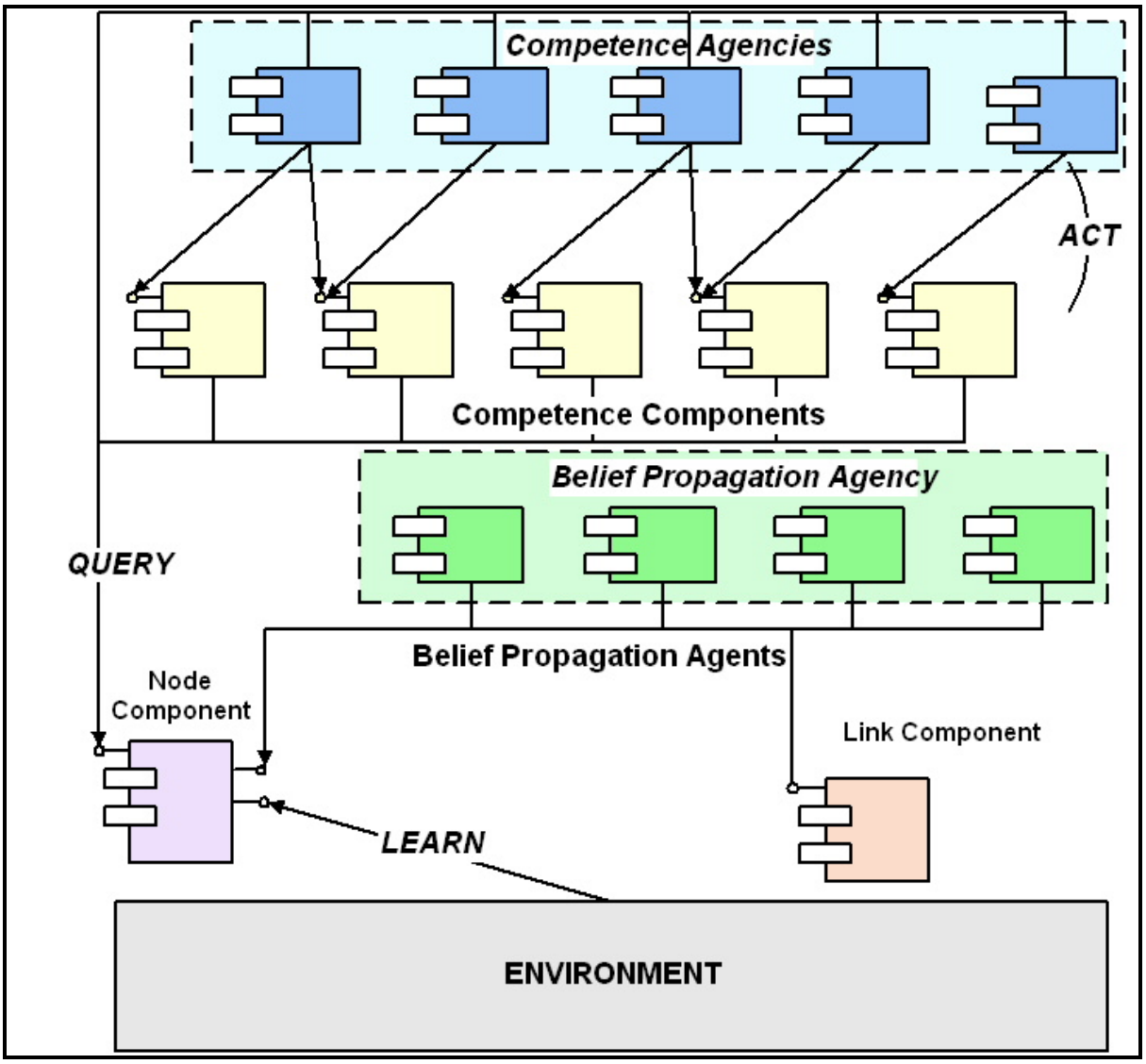

Figure 20: BaBe Agent Architecture Components

Collectively the belief propagation agents propagate evidence through the network and activate component behaviours depending on the beliefs of the Bayesian behaviour network nodes.

We implemented prototype node and link components using EJB entity beans, administering and ensuring persistence for the evidence, π's and λ's, beliefs and

conditional probability matrices for the underlying Bayesian behaviour network nodes. The node components reflect beliefs about the current environmental states, as environmental evidence is presented to the node components as soon as it occurs. As soon as evidence is received from the environment, the history data is updated, and the conditional probability matrices are incrementally updated using Bayesian learning.

We developed prototype belief propagation agents. These components are homogeneous as they are identical EJB message beans, listening on different JMS queues corresponding to the links of the Bayesian behaviour network links. These agents propagate beliefs amongst themselves using Judea Pearl's belief propagation algorithm.

Belief propagation agents communicate with each other through simple tags, as well as through data in a database administered by the node and link components. Using only the link and node components, belief propagation agents collectively propagate beliefs in a distributed Bayesian behaviour network that learns from the environment and that can be queried by competence agencies to determine which component behaviours to activate next. Each competence agency queries the beliefs of a set of node components and activates one or more component behaviours, depending on the beliefs of the queried node components. Each behaviour component executes a particular behaviour, and queries one or more node components in order to use the beliefs in actions that must be executed.

Each of the above components will be discussed in more detail in the next sections.

6.5.1 The Node Component

The node components are homogeneous components. Node components are identical, but each node component corresponds to a different node in the underlying Bayesian behaviour network. The component diagram for any node X is illustrated in Figure 21. This component maintains and administers:

- the conditional probability matrix (CPM) for each node X;
- the prior probabilities vector $\pi(x)$ - see Eq. (27);
- the likelihood vector $\lambda(x)$ - see Eq. (28);
- history of occurrences for each state;
- the latest evidence received for the states of the node;
- a list of the incoming links to the node;

- a list of the outgoing links to the node;
- the name of the node, as well as a list of the states of the node.

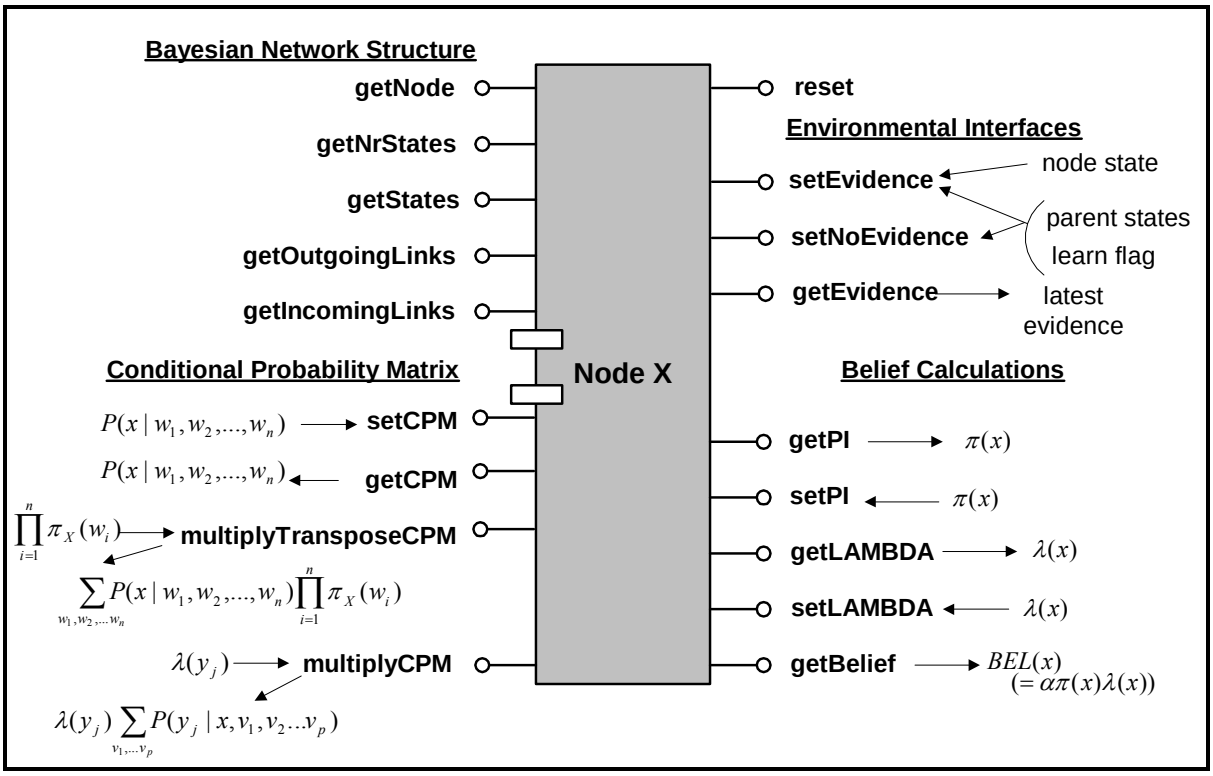

Figure 21: Node Component Diagram for Node *X*

In Figure 21, the interfaces are grouped into four groups, namely the Bayesian network structure interfaces, the conditional probability matrix interfaces, the belief calculation interfaces and the environmental interfaces.

The Bayesian network structure interfaces enable access to the name of the node that the component administers, retrieval of the number of states of this node, a list of descriptions of the states and a list of the incoming links and the outgoing links of this node.

The conditional probability matrix interfaces enable access to the conditional probability matrix, and using these interfaces, calculations can be performed on the conditional probability matrix or its transpose - see Eqs. (27) and (30).

The belief calculation interfaces allows access to $\pi(x)$, $\lambda(x)$ and $BEL(x)$ - see Eqs. (27), (28) and (31).

The environmental interfaces enable interaction with the environment. If the learn flag is true, the evidence received from the environment is added to the history data, and the conditional probability matrix is updated. This is an incremental learning process. As a detailed study of Bayesian learning falls outside the scope of this research, our prototype implementation includes learning in networks with known structure and no hidden variables. In future research, in the case of learning in the presence of missing data, the interface setNoEvidence must activate the execution of the EM algorithm. Russell & Norvig (2003) describe this algorithm in detail.

If the learn flag is false, no learning takes place, but the evidence is taken into account during belief propagation. This form of evidence setting is done when "what-if" queries are executed against the Bayesian behaviour network.

6.5.2 The Link Component

The link components are homogeneous components. Link components are identical, but each link component corresponds to a different link in the underlying Bayesian behaviour network. The link component for any link XY_j is illustrated in Figure 22.

The link component maintains and administers $\pi_{Y_j}(x)$ - see Eq. (29), $\lambda_{Y_j}(x)$ - see Eq. (30) and synchronization flags (PIFlag and LAMBDAFlag).

In Figure 22, the interfaces are grouped into three groups, namely the Bayesian network structure interfaces, the synchronization interfaces and the belief calculation interfaces.

The Bayesian network structure interfaces enable access to the name of the parent node and child node of the link that the component administers, retrieval of a list of the other outgoing (sibling) links of the parent node as well as retrieval of a list of the other incoming links of the child node.

The belief calculation interfaces allows access to $\pi_{Y_j}(x)$ and $\lambda_{Y_j}(x)$ - see Eqs. 29 and 30.

The synchronization interfaces are used by the belief propagation agents to synchronize the calculation of products of π's or λ's of sibling links. The PIFlag keeps track if link XY_j has

calculated $\pi_{Y_j}(x)$ or not, and the LAMBDAFlag keeps track if link XY_j has calculated $\lambda_{Y_j}(x)$ or not.

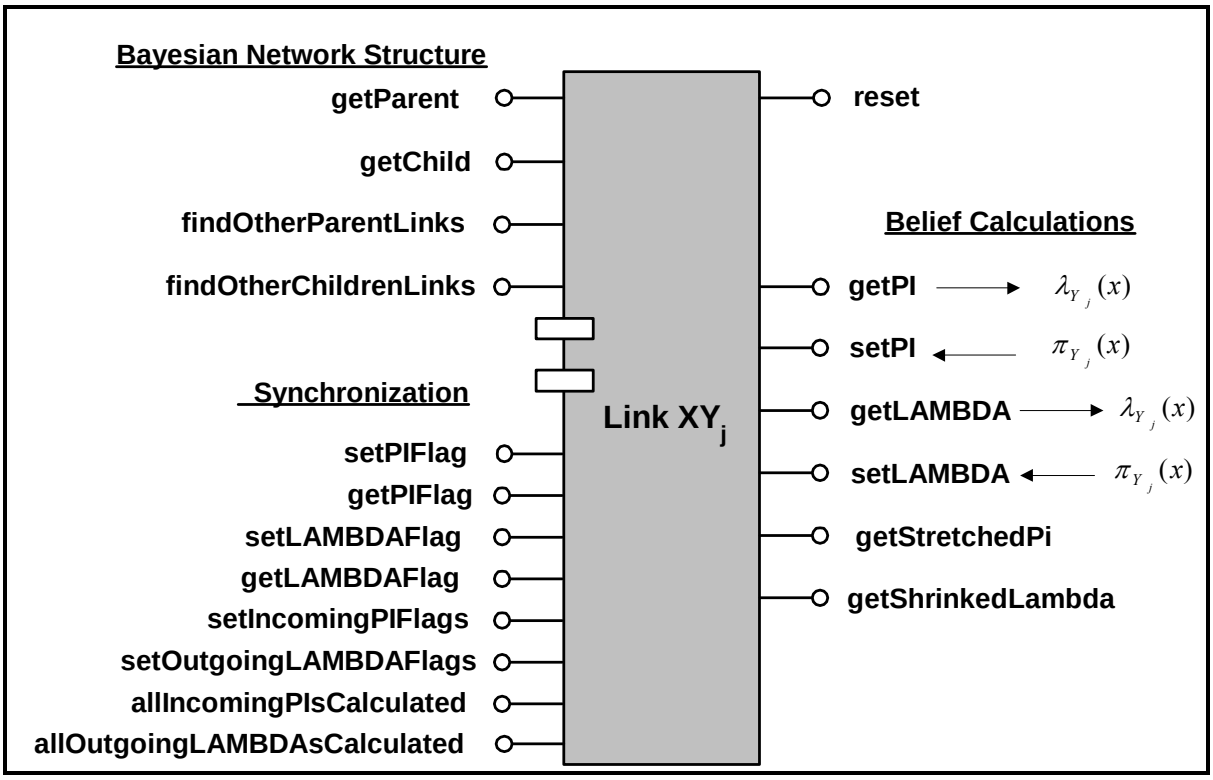

Figure 22: Link Component Diagram for Link XY_j

The allIncomingPIsCalculated interface enables access to a flag that indicates if all the siblings of this link, that are also incoming links of this link's child node, have calculated their link π's yet. As soon as this flag is true, the product of π's of all the incoming links of the child node can be calculated. As soon as this product is calculated, the setIncomingPIFlags interface is used to set setIncomingPIsFlag in the link component to true. As soon as this flag is set, the link component will clear all the PIFlags of all the child node's incoming links and then set setIncomingPIsFlag to false again – ready for the calculation of the next product of π's.

The allOutgoingLAMBDAsCalculated interface enables access to a flag that indicates if all the siblings of this link, that are also outgoing links of this link's parent node, have calculated their link λ's yet. As soon as this flag is true, the product of λ's of all the outgoing links can

be calculated. As soon as this product is calculated, the setOutgoingLAMBDAFlags interface is used to set setOutgoingLAMBDAsFlag in the link component to true. As soon as this flag is set, the link component will clear all the LAMBDAFlags of all the parent node's outgoing links and then set setOutgoingLAMBDAsFlag to false again - ready for the calculation of the next product of λ's.

6.5.3 The Belief Propagation Agents

The belief propagation agents are homogeneous components. These agents are identical EJB message beans, but agent listens on a different JMS queue implementing a link in the underlying Bayesian behaviour network. The messages communicated on the JMS queues are simple tags – LAMBDA tags or PI tags. These tags determine the direction of propagation in the Bayesian network. Figure 23 is a state diagram for a belief propagation agent that illustrates the processes triggered by these tags.

As soon as a belief propagation agent receives a tag, it first identifies the queue it received the tag on, in order to know which link in the Bayesian network the queue corresponds to. Once the link is known to the belief propagation agent, it creates the link and node components needed to access the underlying Bayesian behaviour network information.

A PI tag will trigger the calculation of the link's π. If all the child node's incoming links have calculated their link π's, then the child node's π is calculated. As soon as the child node's π is updated, PI tags are sent to the queues corresponding to its outgoing links if it is not a leaf node. The belief propagation agent will then go into a wait state for the next tag to arrive on its queue.

A LAMBDA tag will trigger the calculation of the link's λ. If all the parent node's outgoing links have calculated their link λ's, then the parent node's λ is calculated. As soon as this node's λ is updated, LAMBDA tags are sent to the queues corresponding to the parent node's incoming links if it is not a root node, otherwise PI tags are sent to the queues corresponding to its outgoing links – reversing the propagation direction. The belief propagation agent will then go into a wait state for the next tag to arrive on its queue.

Each belief propagation cycle is a two-phase process, which is activated as soon as a set of evidence is received from the environment. The propagation of LAMBDA tags upwards from the leaf nodes through the network gathers evidence from the environment, followed by the flow of PI tags downwards from the root nodes, gathering a priori information. A LAMBDA tag will cause propagation of LAMBDA tags upwards in the direction of

predecessor nodes, except if the parent of the link is a root node. In this case, the direction will be reversed and PI tags will be propagated towards the children nodes. A PI tag will cause propagation of PI tags in the direction of children nodes, except if the child of the link is a leaf node.

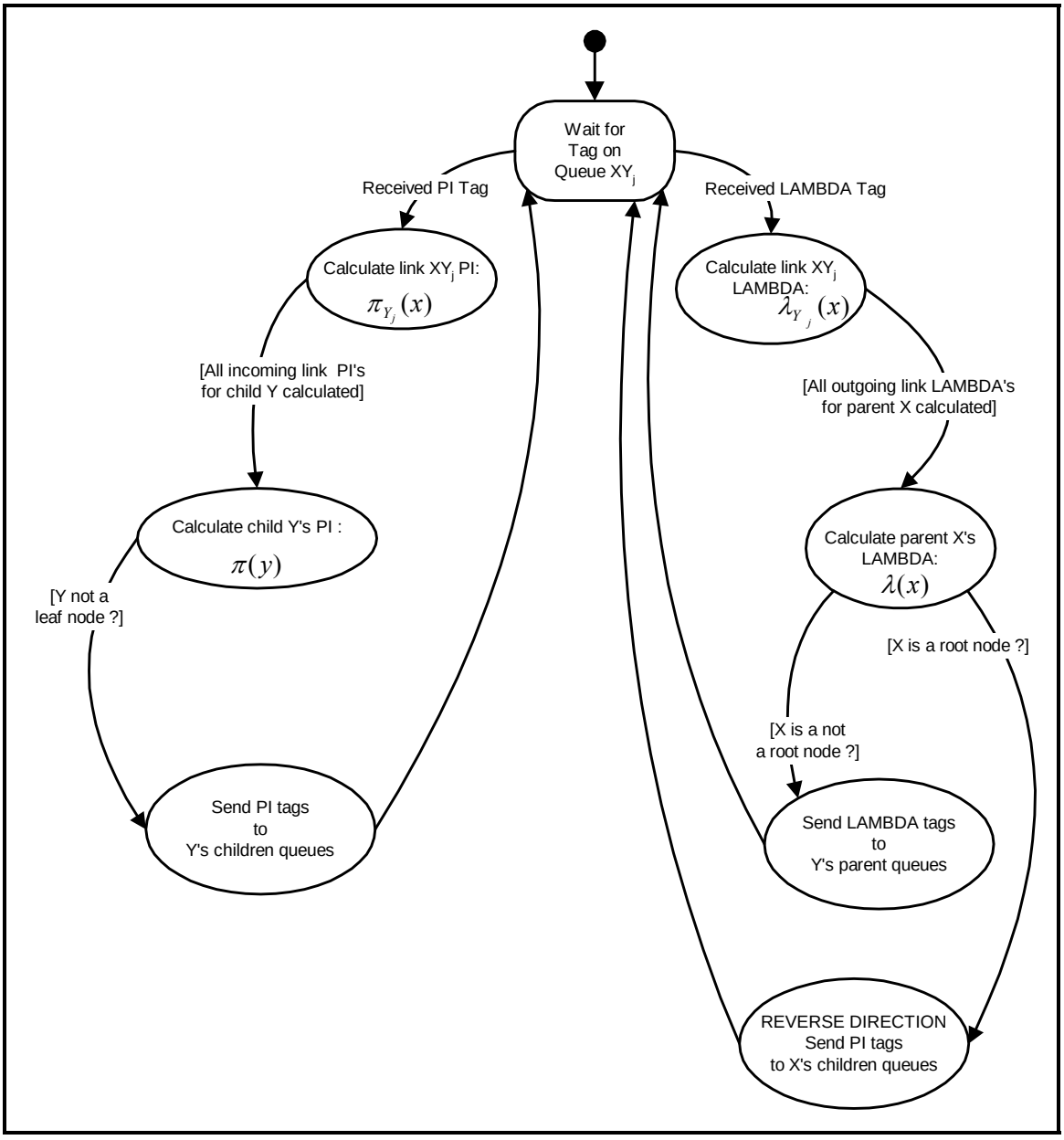

Figure 23: The Belief Propagation Agent

In the next two sections, the processes triggered by the different tags are described in more detail. These processing steps must be studied in conjunction with Judea Pearl's belief propagation algorithm, as well as the node and link component diagrams in Figures 21 and 22.

6.5.3.1 Handling PI tags

Each PI tag received on a queue representing link XY_j, triggers the following processing steps:

1. Create a link component corresponding to the link XY_j
   a. Use the getParent and getChild node component interfaces to determine the names of the parent (X) and the child node (Y_j) of this link.
   b. Use the findOtherParentLinks interface to retrieve a list of the names of all siblings of this link that have the same parent as this link – in other words, all the other outgoing links of the parent node. The returned list of links will be $\{XY_1, XY_2 \ldots XY_{j-1}, XY_{j+1} \ldots XY_m\}$.
   c. Use the findOtherChildrenLinks interface to retrieve a list of the names of all siblings of this link that have the same child as this link – in other words, all the other incoming links of the child node. The returned list of links will be $\{V_1Y_j, \ldots V_pY_j\}$.
2. For each of the other sibling links that are outgoing links of the parent link X, create a link component and use the getLAMBDA interface to retrieve $\lambda_{Y_k}(x), k = 1 \ldots m$. Calculate the product of these link λ's, namely $\prod_{k \neq j} \lambda_{Y_k}(x)$.
3. Use the getBelief interface to retrieve the belief of node X, namely $BEL(x)$, and calculate $\pi_{Y_j}(x) = BEL(x) \prod_{k \neq j} \lambda_{Y_k}(x)$, using the product of λ's calculated in the previous step. Note that this equation differs from Eq. (29) in that $BEL(x)$ is used instead of $\pi(x)$.
4. Use the link component for link XY_j to:
   a. set $\pi_{Y_j}(x)$ using the interface setPI. Use the setPIFlag interface to set the flag to indicate that this link has now calculated its π.

b. Test if the other incoming sibling links have calculated their $\pi's$ yet, by accessing the allIncomingPIsCalculated interface.

c. If this link is the last link to calculate its π, the π for child node Y_j can now be calculated. Use the interface setIncomingPIFlags to set a flag that will indicate to the link component to clear all the PIFlags for all the incoming links to node Y_j – ready for the next belief propagation cycle.

   i. For each of the other sibling links that are incoming links of the child node Y_j, create a link component and retrieve $\pi_{Y_j}(v_i), i = 1...p$, using the getPI interfaces of these link components. Calculate the product of the link π's.

   ii. Create a node component for child node Y_j and use its multiplyTransposeCPM interface to calculate the product of the conditional probability matrix and the product of the incoming link π's, calculated in the previous step. The result is $\pi(y_j)$ which is updated using node component Y_j's setPI interface.

   iii. If node Y_j has children nodes, then use node component Y_j's getOutgoingLinks interface to get a list of names of all the outgoing links of this child node. Place a PI tag on the JMS queue corresponding to each of these link names.

6.5.3.2 Handling LAMBDA tags

Each LAMBDA tag received on a queue, representing link $XY_{j,}$ triggers the following processing steps:

1. Create a link component corresponding to the link XY_j.
2. Use the getParent and getChild node component interfaces to determine the names of the parent (X) and the child node (Y_j) of this link.
3. Use the findOtherParentLinks interface to retrieve a list of the names of all siblings of this link that have the same parent as this link – in other words, all the other outgoing links of the parent node. The returned list of links will be $\{XY_1, XY_2 \ldots XY_{j-1}, XY_{j+1} \ldots XY_m\}$.

4. Use the findOtherChildrenLinks interface to retrieve a list of the names of all siblings of this link that have the same child as this link – in other words, all the other incoming links of the child node. The returned list of links will be $\{V_1Y_j, \ldots V_pY_j\}$.

5. For each of the other sibling links that are incoming links of the child node Y_j, create a link component and retrieve $\pi_{Y_j}(v_i), i = 1...p$. Calculate the product of the link π's.

6. Create a node component for child node Y_j and use the getLAMBDA interface to determine child node's $\lambda(y_j)$. Use the multiplyCPM to calculate the product of $\lambda(y_j)$ and the conditional probability matrix. Multiply this product with the product of the incoming link π's calculated in the previous step. The result is $\lambda_{Y_j}(x)$.

7. Use the link component for link XY_j to:

    a. set $\lambda_{Y_j}(x)$ using the interface setLAMBDA. Use the setLAMBDAFlag interface to set the flag to indicate that this link has now calculated its λ.

    b. Test if the other outgoing sibling links have calculated their λ's yet, by accessing the allOutgoingLAMBDAsCalculated interface.

    c. If this link is the last link to calculate its λ, the λ for parent node X can now be calculated. Use the interface setOutgoingLAMBDAFlags to set a flag that will indicate to the link component to clear all the LAMBDAFlags for all the outgoing links from node X, ready for the next belief propagation cycle.

        i. For each of the other sibling links that are outgoing links of the parent node X, create a link component and retrieve $\lambda_{Y_j}(x), j = 1...m$. Calculate the product of these link λ's. The result is $\lambda(x)$ which must be updated using the setLAMBDA interface.

        ii. If node X has parent nodes, then use parent node X's getIncomingLinks interface to get a list of all the incoming links of this parent node. Place a LAMBDA tag on the communication queue corresponding to each of these link names.

        iii. If node X has no parent nodes, then use parent node X's getOutgoingLinks to get a list of names of all the outgoing links of this

node. Place a PI tag on the communication queue corresponding to each of these link names.

6.6 Bayesian Agencies

There are two types of Bayesian agencies in the BaBe agent architecture, namely belief propagation agencies and competence agencies. In Figure 24, a state diagram for these agencies is presented.

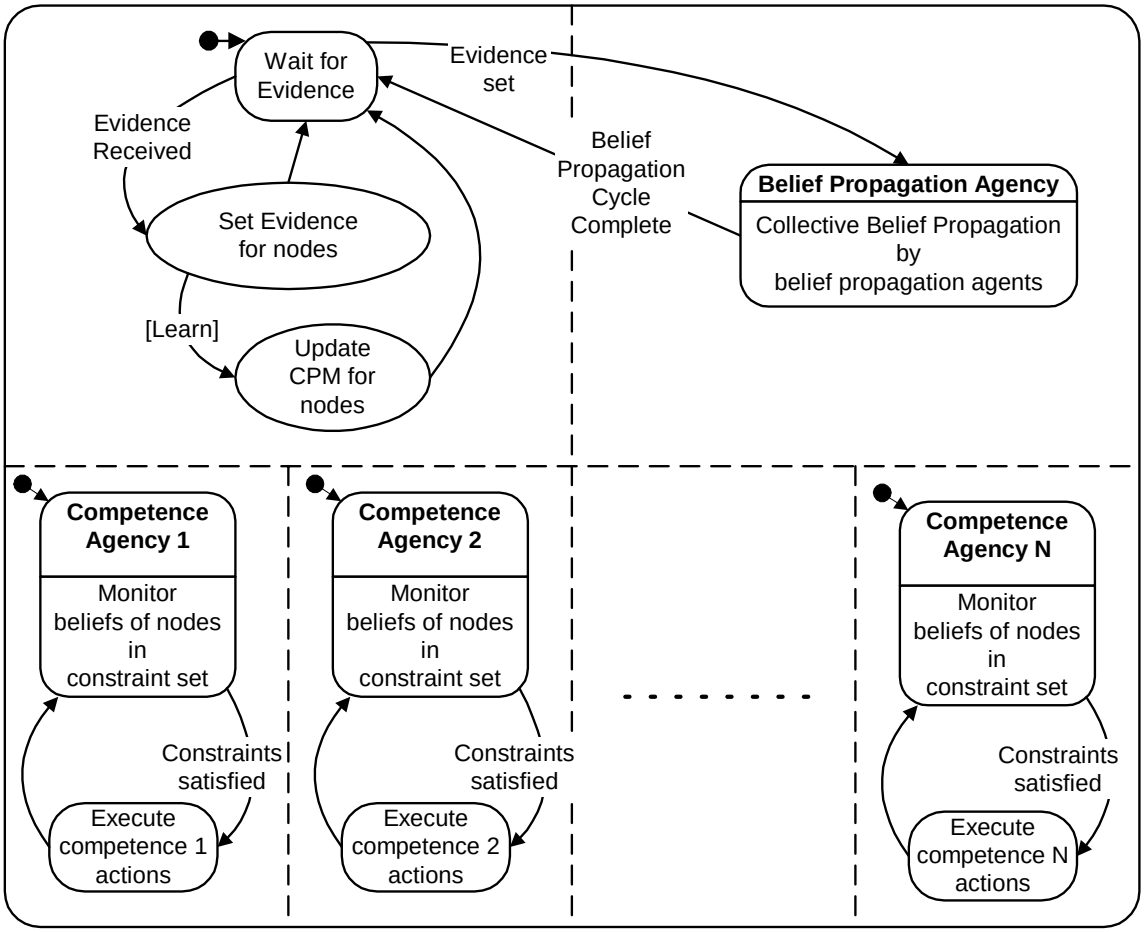

Figure 24: Bayesian Agencies State Diagram

Collective belief propagation by the belief propagation agents in response to the environmental evidence is a continuous process. As soon as evidence is received from the environment, it is presented to the appropriate nodes via node component interfaces, and the

conditional probability matrices are updated if learning is activated. As soon as all the node components received their evidence, the belief propagation agencies perform a belief propagation cycle – gathering evidence from the node components and updating the beliefs of the nodes. The competence agencies use these beliefs to determine if certain component behaviours or actions must be executed or not. Each competence agency monitors a set of constraints on the beliefs of a subset of nodes – the constraint set. If all the constraints in a constraint set are met, the competence agency can execute its actions.

Figure 25 illustrates how the Bayesian agencies are organized into a heterarchy in order to collectively implement the Bayesian behaviour network illustrated in Figures 17 and 18.

The belief propagation agency consists of all the belief propagation agents (ABAgent, BGAgent, etc.). The heterarchy in Figure 25 consists of a belief propagation agency and four competence agencies, namely the personaliseAgency, the marketingAgency, the hyperLinkAgency and the nextPageAgency. These agencies overlap and "share" node components with the belief propagation agency, thus forming a heterachy.

The BaBe adaptive agent architecture achieves self-awareness by using Minsky's model of an A-Brain and a B-Brain. The belief propagation agencies can be viewed as A-Brains, connected to the environment and continuously inferring beliefs about and learning from the latest environmental states. The competence agencies can be viewed as B-Brains, inspecting the beliefs of the Bayesian behaviour network nodes underlying the belief propagation agencies (the A-Brains) and acting according to these beliefs.

Each competence agency implements a competence set $\Theta_i = (C_i, A_i)$, where C_i is a set of constraints on a subset of nodes and their states in the Bayesian behaviour network, and the A_i is a set of component actions that must be executed if all the constraints in C_i are met. In an enterprise, example component actions will include parts of the business processes or workflows in the enterprise. These actions are packaged into one or more re-usable components, called competence components.

Each competence agency accesses the node components for the nodes in the constraint set. The beliefs of these nodes are accessed using the getBelief node interfaces, and tested to determine if the beliefs satisfy all the constraints in the constraint set. If all the constraints are met, the competence components are executed.

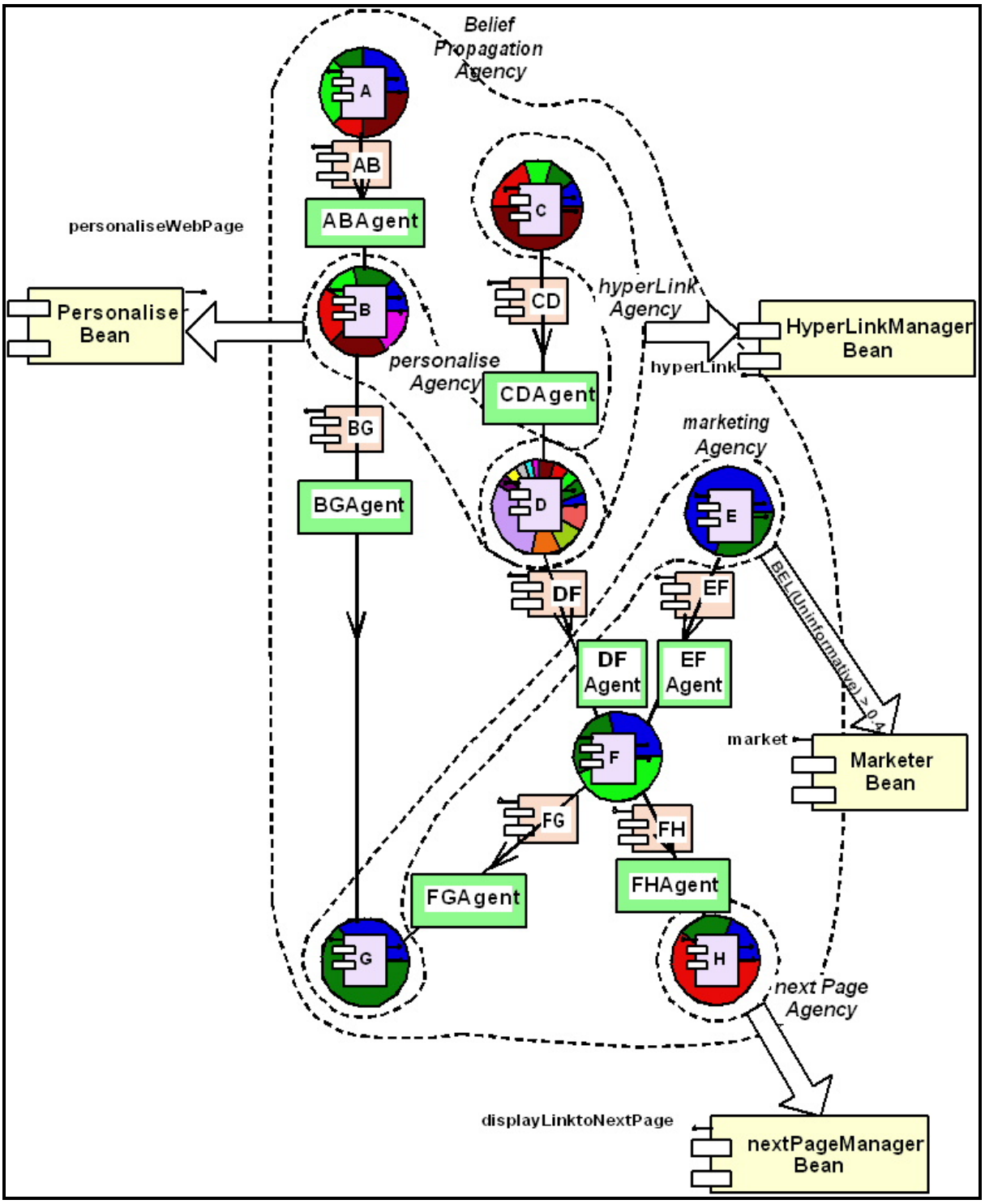

Figure 25: Heterarchy of Bayesian Agencies

In Figure 26 below, the components for the Bayesian behaviour network in Figure 25 is given. The BeliefPropagationAgentJAR contains all the belief propagation agents, all identical, but listening on different JMS queues. The NodeBeanJAR contains the node component, and the LinkBeanJAR contains the link component. The CompetencesJAR contains the competence components, namely MarketerBean, NextPageManagerBean, HyperLinkManagerBean and PersonaliserBean. The competence components have interfaces to the behaviours or actions that the competence agencies can execute.

The personaliseAgency implements competence set Θ_1 = {{}, personaliseWebPage}. It (unconditionally) calls the personaliseWebPage interface of the PersonaliserBean.

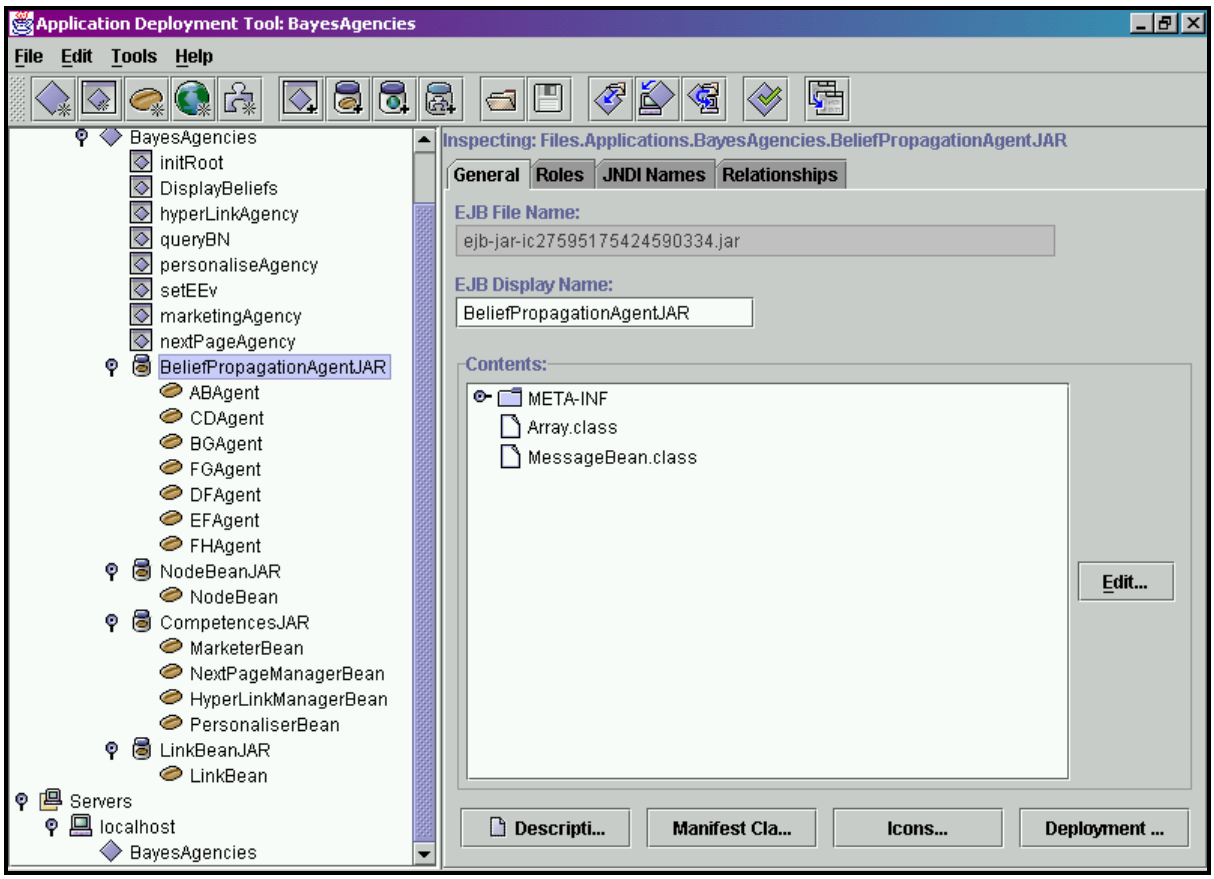

Figure 26: Example BaBe Components

Figure 27 is a screen dump of the output of personaliseWebPage, which in this simple example displays the beliefs of nodes *B* and *D*, after belief propagation in the presence of a

mathematician browsing a website listing books on Bayesian networks by Judea Pearl. (Note that the probabilities are the same as in Figure 18).

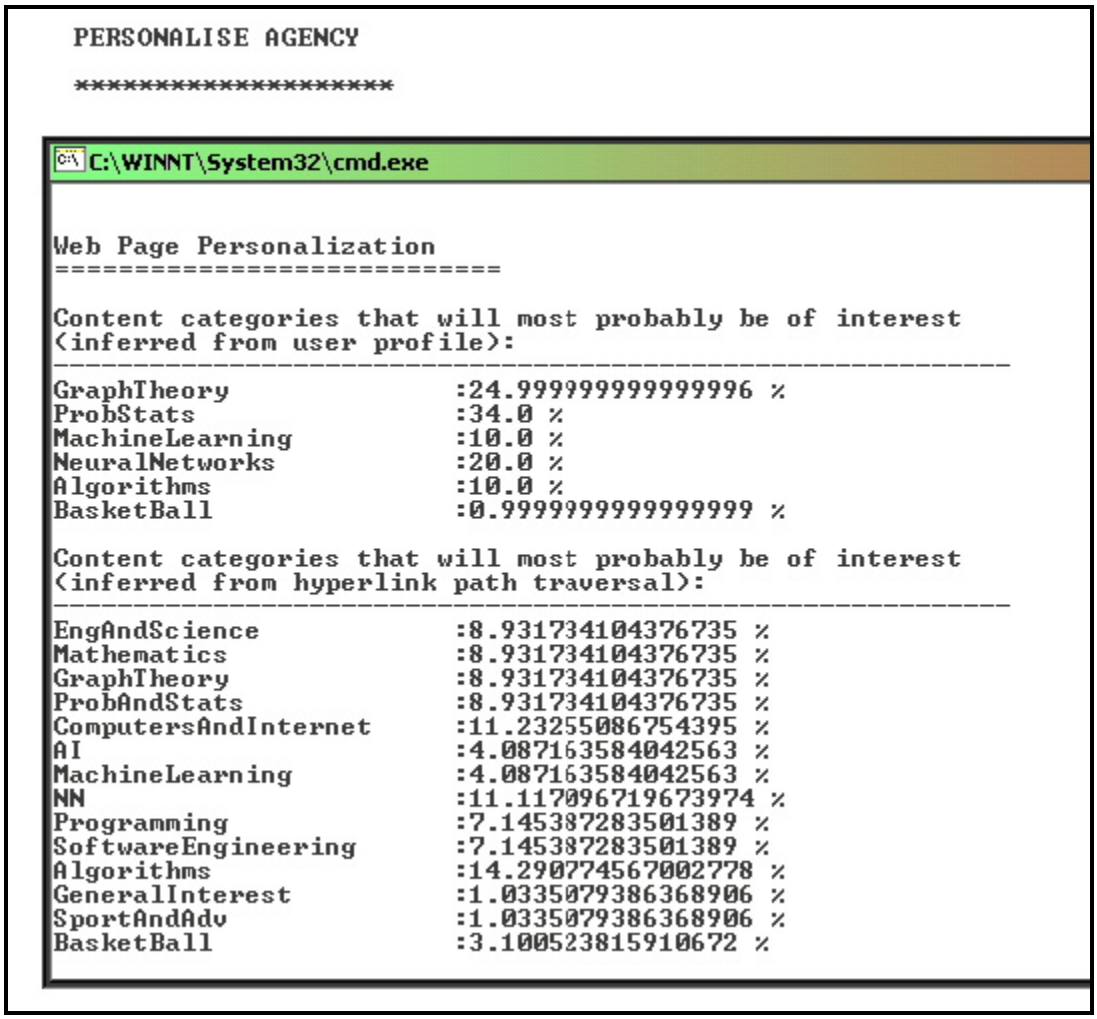

Figure 27: PersonaliseAgency Output

The hyperLinkAgency implements competence set Θ_2 = {{}, hyperLink}. It (unconditionally) calls the hyperLink interface of the HyperLinkManagerBean, which accesses the beliefs of nodes *C* and *D*.

The marketingAgency implements competence set Θ_3 = {{*BEL*(*E* = Uninformative) > 0.4}, market}. It calls the market interface of the MarketerBean if the belief that the advertisements are uninformative exceeds 0.4, which accesses the beliefs of nodes *E* and *G*.

The nextPageAgency implements competence set Θ_4 = {{ }, displayLinktoNextPage}. It (unconditionally) calls the displayLinktoNextPage interface of the NextPageManagerBean, which uses the beliefs of node *H*.

6.7 Bayesian Agencies in Web Personalization

Figure 28 illustrates how Bayesian agencies can be used in a simplified web personalization application. There are two sets of Bayesian agencies in this example, namely the clickstream and the content Bayesian agencies. The clickstream Bayesian agencies collectively implement a dynamic Bayesian network, modelling a two-way contents-product aspect model at each time step.

The content Bayesian agencies collectively implement a Bayesian behaviour network that models a hierarchical concept model, representing the relationships between words extracted from web pages and higher-level concepts, at different levels of abstraction. In Figure 28, during time step t, a bag of words, BOW(t) = {w2, w4, w5} is extracted from the PageView(t) that a user browses. BOW(t) is then presented to the content Bayesian agencies that collectively reduce the dimensionality of the words to a bag of concepts BOC(t) = {co1, co4}. Content agency 1 adds concept co1 to BOC(t) and content agency 2 adds co4 to BOC(t) because their beliefs exceeded a certain threshold as a result of the collective belief propagation by the content Bayesian agencies. Each BOC is "filled" through the emergent behaviour of agencies 1, 2 and 3.

Clickstream agency(t) uses BOC(t) as evidence together with products purchased from PageView(t) to predict the contents and products that might interest the user next. The behaviour associated with Clickstream agency(t) is the personalization of PageView(t+1). BOW(t+1) is next extracted from the personalized PageView(t+1) viewed by the user. This process is repeated until the session ends.

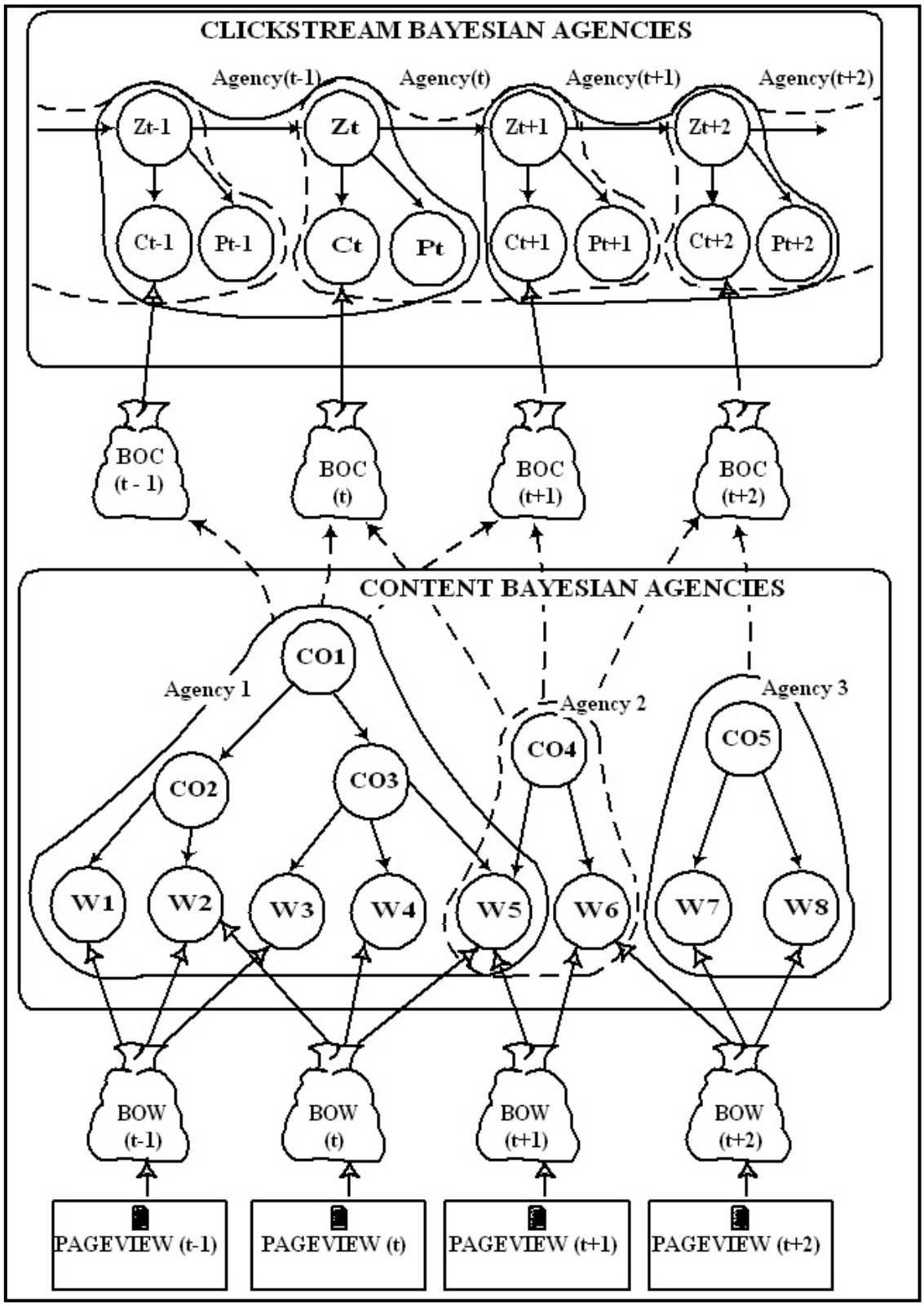

Figure 28: Bayesian Agencies in Web Personalization

6.8 Applicability, Use and Value of the BaBe Agent Architecture

6.8.1 A General Application of the BaBe Agent Architecture

A general J2EE-based application of the BaBe agent architecture is illustrated in Figure 29. This application consists of three tiers, namely a client-tier, a business-tier and an enterprise information system (EIS) – tier. The client-tier consists of a Bayesian network administration client and one or more user clients, running on one or more client machines. The business-tier consists of the Bayesian agencies, competence and environment components deployed on one or more J2EE servers. The EIS-tier consists of one or more database or legacy machines.

Figure 29 illustrates the distribution of the BaBe system components between the three tiers.

1. The Business-Tier:

   - Bayesian Agencies

     The Bayesian agencies form the heart of the BaBe system. These agencies collectively implement a Bayesian behaviour network. The knowledge engineer uses the Bayesian Network Administration Client to initialize or to modify the Bayesian behaviour network. The Bayesian agencies update this network from relationships learnt from evidence received from the environment components in an ongoing process of responding to environmental changes.

   - Competence Components

     These components contain the business logic of the enterprise, and are packaged by code developers in collaboration with the system and knowledge engineers.

   - Environment Components

     The environment components are generated by the administration client from specifications supplied by the administrator of the system. These components ensure persistence and act as an interface between the Bayesian agencies and the data sources in the environment.

2. The Client-Tier:

   - The Administration Client

     The administrator of the system uses the administration client to specify all relevant data sources. The administration client then generates and deploys the environment components.

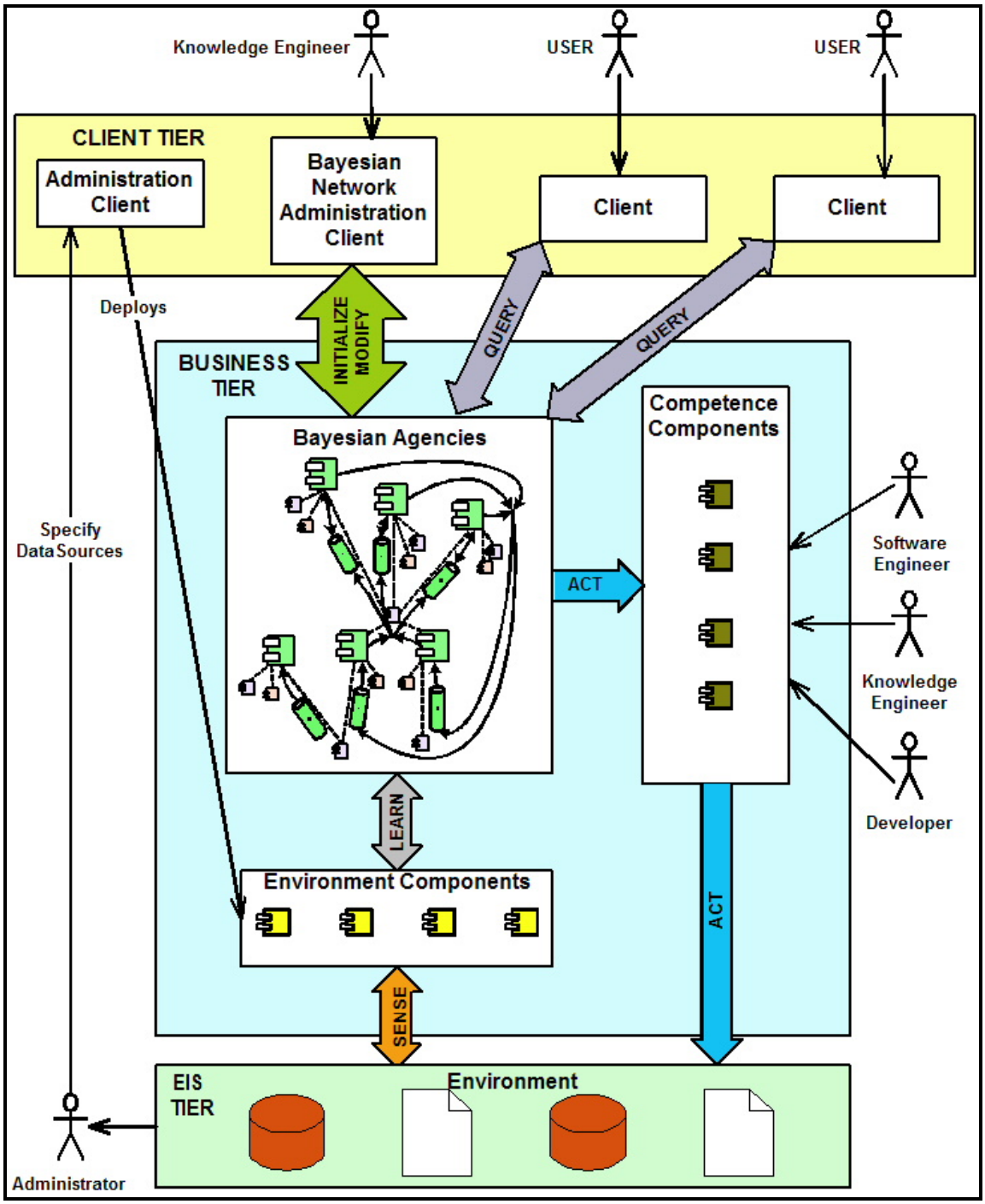

Figure 29: BaBe – A General Application

- The Bayesian Network Administration Client

  Using this client, a knowledge engineer can capture his/her prior knowledge about

the domain and relationships within the domain. He/she can also view the Bayesian behaviour networks collectively implemented by the Bayesian agencies, and change it if necessary. This client allows the knowledge engineer to associate competence sets and Bayesian network node sets. Each competence set defines one or more actions packaged into one or more competence components.

- User Clients
  The users of the system use these as interfaces to query the system to determine relationships between data entities, query trend predictions, etc.

3. The EIS-Tier:
   This tier consists of legacy systems, databases, documents, media files, and other data sources in the distributed environment of the enterprise.

The Bayesian agencies are by default in learning mode. As soon as any of the data sources in the environment changes, the environment components gather evidence from all related sources and present the evidence to the node components of the Bayesian agencies. This in turn triggers the learning in the Bayesian agencies. Belief-propagation in the agencies will cause certain competence components to be activated, depending on the beliefs of the nodes in the underlying Bayesian behaviour network. The competence components, in turn act in the environment, effecting changes to data sources or documents.

When a user client queries the Bayesian network for patterns of interest, the Bayesian agencies temporarily change their mode from learning mode to query mode. Using the evidence supplied by the user client, the Bayesian agencies perform belief propagation, and return the result to the user client. The Bayesian agencies immediately return to learning mode, and process all the evidence that was received whilst processing the query.

6.8.2 The Limitations of the BaBe Architecture

One of the limitations of the BaBe application is that it implements Bayesian inference in singly-connected Bayesian networks only. The belief propagation agencies propagate beliefs through the networks according to Judea Pearl's belief propagation algorithm for singly-connected networks. It is possible to apply this algorithm to multiply-connected networks, as the algorithm is an approximation method for multiply-connected networks. The results will not necessarily be correct, and messages might circulate indefinitely around loops, but the values obtained are usually very close to the true values (Pearl, 1988) (Russell & Norvig, 2003).

The other limitations of the BaBe architecture are caused by the static nature and inflexibility of the J2EE platform. This platform requires the complete system functionality and adaptation logic to be specified at build time, after which the system is deployed. For example, when the Bayesian network administration client changes the Bayesian behaviour network structure, or when the Bayesian agencies discover new relationships between data sources, new queues must be created, and new belief propagation agents, node and link components must be deployed. If the Bayesian agencies find that relationships between data sources become insignificant, queues must be deleted, belief propagation agents must be destroyed and the associated node and link components must be destroyed. The competence agencies activate competence components as emergent responses to the belief propagation in the underlying Bayesian behaviour network. If a competence component must be changed, it must be recompiled and redeployed. If a new competence component is added, it must be deployed. The J2EE platform does not allow any of these activities during runtime. Any one of the above activities will require a new deployment of the entire BaBe architecture – a cumbersome process.

A new generation of component-based frameworks is emerging, namely reflective component-based frameworks. These frameworks will allow all of the above activities to be performed during execution time. BaBe will only realise its full potential when deployed in a reflective component-based framework. Kon, Costa, Blair & Campbell (2002) describe a reflective component-based system as a collection of components that can be configured and reconfigured by the application in order to adapt to changes in the environment. Examples of reflective component-based systems include for example dynamicTao (Kon, Román, Liu, Mao, Yamane, Magalhães & Campbell, 2000), X-Adapt (McGuren & Conroy, 2002) and Chisell (Keeney & Cahill, 2003).

6.8.3 The Benefits of the BaBe System

BaBe is unique. As the first component-based complex adaptive computer system, it will set a new standard in the modelling of the dynamics of change. It implements an adaptive agent architecture with an internal model that is dynamic and that can change automatically in response to patterns mined from environmental changes. Traditional systems have either static models or models that must be updated manually. For example, the BDI (Belief-Desire-Intention) (Rao & Georgeff, 1995) and BDJI (Belief-Desire-Joint-Intention) (Jennings, 1993) deliberative agent architectures have static models, and the Subsumption Architecture (Brooks, 1985) and the behaviour networks of Maes (1990) have to be

maintained manually. The Bayesian behaviour networks implemented by the Bayesian agencies sense changes in real-time by the environment components from the different data sources in the environment and the patterns are integrated into the Bayesian behaviour networks as they emerge.

Bayesian agencies can be integrated into the static BDI and BDJI models. One way to do this is to place the possible worlds model under the control of the Bayesian agencies, so that this model always reflects the current state of the environment. Rather than maintaining an input queue of events that is serviced on a cyclic basis, the Bayesian agencies will react to environmental changes as they occur and activate competence components to update the data structures representing the agent's beliefs, desires, intentions and joint intentions.

A major benefit of BaBe is its component-based architecture. For example, when the administrator defines new data sources using the administration client, the necessary environment components are deployed, and the Bayesian agencies immediately take these new data sources into account when mining patterns in the data. The business logic of the application is packaged into the heterogeneous competence components, which are controlled by homogeneous components assembling the Bayesian agencies. The component-based architecture and the re-usability of the components make this system scalable, and able to evolve.

The BaBe application benefits from the underlying support provided by the J2EE server, which includes services such as security, transaction management, Java Naming and Directory Interface (JNDI) lookups, and remote connectivity.

6.8.4 How and where can the general BaBe application be used?

Organisations are complex systems that are characterised by high numbers of component entities, and a high degree of interactions. The outcome of any change to the system cannot always be predicted. Also, a sequence of changes can occur over time, following an initial disturbance. Humans find such changes and sequences difficult to understand and anticipate.

The BaBe application can be deployed in an environment to highlight the impact of emergent properties and to control the behaviour of the system in response to emergence. The Bayesian agencies organize themselves into distributed Bayesian behaviour networks, collectively modelling the dynamics of change. The Bayesian agents can collectively "mine" the complex web of interrelationships including for example internal relationships within the company and interrelationships implicit in knowledge regarding products and services,

business processes and business units; specific projects and project implementations; customers and the marketplace. The enterprise can then use this self-understanding to adapt its business processes and to formulate new knowledge or business strategies in response to the ever-changing marketplace in order to sustain its competitive advantage.

One use of the general BaBe application is to implement a dynamic, self-defining trading network. A supply chain is a sequence of logistical activities that move raw materials, transform them into goods, and move them to the end consumer or customer. A value chain is the whole process of creating value across the product life cycle, from conceiving the product, through manufacture, onward distribution, sales and into the hands of a customer.

A value chain consists of a network of commercial relationships that are fluid, and form in response to consumer demand, shaping themselves in real time to best satisfy the consumer demand, faster and at lower cost. The general BaBe application can ensure that all participants in such a network are included, and that business decisions, negotiations and transactions can occur at machine speed, independent of human intervention.

6.9 The BaBe Agent Architecture: A Complex Adaptive System

Humans find it difficult to understand and anticipate the effects of emergence in complex computer systems. Complex adaptive systems, a branch of complexity theory research, can be used to model the dynamics of change caused by emergence. A complex adaptive system is characterized by complex behaviours that emerge as a result of interactions among individual system components and among system components and the environment.

The BaBe agent architecture can function as a complex adaptive system, as it satisfies all of Holland's properties and mechanisms as discussed below:

1. The Aggregation Property – the simple agents in the BaBe architecture are organised into adaptive aggregates. Belief propagation agents are organised into belief propagation agencies, and competence components are organised into competence agencies. These agencies are organised into a heterarchy through shared node components in the underlying Bayesian behaviour network. The belief propagation agencies collectively propagate belief in the presence of evidence. If the network is in learning mode, the conditional probability tables of the nodes are updated. The competence agencies query the network nodes and activate competence components if certain conditions are met.

2. The Tagging Mechanism – The component interfaces facilitate selective interaction. Evidence from the environment is presented to nodes using the node component interfaces. Belief propagation agents communicate with their neighbours by placing LAMBDA and PI tags on queues, representing the Bayesian behaviour network structure. Belief propagation agents update the beliefs of the nodes by accessing the link and node component interfaces. The competence agencies query the beliefs of nodes, using the node interfaces, and depending on the beliefs of the nodes, activate behaviours via the competence component interfaces.
3. The Non-linearity Property – The BaBe agent architecture exhibits emergent behaviour. Its overall behaviour is more than what can be predicted from the behaviours of the individual competence components and belief propagation agents.
4. The Flows Property – The belief propagation agents collectively propagate LAMBDA and PI tags through the Bayesian behaviour network given the current environmental state. This is a continuous process of receiving environmental evidence and sending LAMBDA and PI tags on message queues representing the Bayesian behaviour network topology.
5. The Diversity Property – The Bayesian agencies can incrementally adapt by using Bayesian learning to discover new regularities or destroying redundant ones. Each new adaptation in the underlying Bayesian behaviour network can cause the competence agencies to activate different or new actions and interactions.
6. The Internal Models Mechanism – The BaBe agent architecture uses Bayesian behaviour networks as adaptive hyperstructures in its internal model. The powerful inference capabilities of these networks enable the Bayesian agencies to collectively anticipate and predict. The ability of Bayesian learning to discover structure from data, enable the Bayesian agencies to collectively "mine" regularities from the input stream and integrate it into the Bayesian behaviour network topology. The modified internal model will then enable the Bayesian agencies to anticipate and act upon the consequences that follow when a similar pattern is encountered.
7. The Building Blocks Mechanism – The Bayesian behaviour networks in the internal model are assembled from BaBe components. These components include homogeneous node components, link components and belief propagation agents grouped into belief propagation agencies, as well as heterogeneous competence components grouped into competence agencies.

The ability to act as a complex adaptive system empowers BaBe to model and act upon the dynamics of change. This technology enables BaBe to develop robust strategies in a rapidly evolving environment, highlighting the possible impact of emergent properties and acting in response to this knowledge.

6.10 Conclusion

The BaBe agent architecture can function as a complex adaptive system, learning from and acting upon emergence. BaBe can be deployed in an uncertain environment to model and act upon the dynamics of change caused by emergence. Humans find it very difficult to comprehend emergence in computer systems and to anticipate the consequences of emergence. The BaBe architecture can assist the human by modelling and adapting to the effects of emergence in uncertain and unpredictable environments such as value chains and knowledge management applications.

The BaBe agent architecture is a component-based adaptive agent architecture. It consists of simple components, grouped into Bayesian agencies. These agencies, in turn, are organised into heterarchies, exploiting the power inherent in these organisational structures.

The BaBe components include node components, link components, belief propagation agents grouped into belief propagation agencies, and competence components grouped into competence agencies. Collectively the belief propagation agents, node components, link components and competence components assemble the Bayesian behaviour networks as hyperstructures in BaBe's internal model.

The belief propagation agents are homogeneous components that collectively propagate beliefs, given the current environmental state, in singly-connected Bayesian behaviour networks. The node components incrementally learn from environmental states by keeping history data and updating the conditional probability matrices in response to evidence received from the environment. The competence agencies overlap and "share" node components with the belief propagation agencies, thus forming a heterachy. Each competence agency activates one or more competence components as emergent properties of the belief propagation in the underlying Bayesian behaviour network. The competence components are heterogeneous components that capture domain-specific behaviours such as parts of business processes or workflows in an enterprise. The node components and belief propagation agents collectively ensures situatedness of the Bayesian agencies as the underlying Bayesian behaviour network always reflects beliefs about the current environmental states.

BaBe can function as a complex adaptive system as it satisfies all of Holland's properties and mechanisms. The BaBe components collectively implement the internal model and building blocks mechanism. The Bayesian agencies are adaptive aggregates, which have non-linear properties due to their capability to exhibit emergent behaviour. The overall behaviour of the Bayesian agencies are more than what can be predicted from the individual behaviours of the belief propagation agents and the competence components. The Bayesian learning contributes to the diversity property of this architecture. The belief propagation algorithm implements the flows property, as it is a continuous process of receiving environmental evidence and sending tags on message queues representing the Bayesian behaviour network topology. These tags trigger the update of the beliefs of each node given the environmental states.

The BaBe component interfaces implement the tagging mechanism that enables the belief propagation and competence agencies to interact with each other in accordance to the structure of the underlying Bayesian behaviour network. The belief propagation agencies and competence agencies interact through the BaBe component interfaces. Environmental evidence is represented to the node components via their interfaces, which is used to update the conditional probability tables of the nodes. The node component interfaces supply access to the beliefs of nodes, queried by the competence agencies. Belief propagation agents communicate with each other using queues representing links in the underlying Bayesian behaviour network.

The Bayesian agencies are self-aware. A belief propagation agency can be viewed as an A-Brain that is connected to the real world. As soon as evidence is received from the environment, the belief propagation agents collectively perform belief propagation. The competence agencies can be viewed as constituting the "B-Brain". These agencies can "see" inside the "A-Brain" by inspecting the beliefs of nodes and acting upon these beliefs by activating competence components. The actions of the competence components can change the state of the environment, influencing the collective belief propagation in the belief propagation agency – the "A-Brain". These actions are therefore part of the dynamic interaction with the environment, ensuring embodiment of the Bayesian agencies.

We described a general application of the BaBe architecture based on the J2EE platform. This platform does not support runtime configuration of components and middleware services as can be done in the new generation of component-based frameworks, namely reflective component-based frameworks. BaBe will only realize its full potential if implemented using a reflective component-based framework.

Chapter 7
The BaBe Methodology

> *The presence of emergence in engineering may be a natural consequence of the modern trend which is leading engineering into areas where we expect machines to do things which we cannot really specify, but, like intelligence and life, can only say "I will know it when I see it!"*
>
> - E. Ronald & M. Sipper

7.1 Overview

According to Minsky (1988) the process of understanding any large and complex "thing" (system) can be broken up into three subprocesses, as follows:

> *First we must know how each separate part works. Second we must know how each part interacts with those to which it is connected. And third, we have to understand how all these local interactions combine to accomplish what the system does – as seen from the outside.*

Agent-oriented software engineering currently focuses only on Minsky's first two steps. These methodologies design the individual agents and the interactions between them, but fail to address the understanding of emergent behaviour – Minsky's third step above.

Minsky described a model of how the human mind observes its own emergent behaviour, by the division of the brain into an A-Brain and a B-brain in order to allow the mind to "watch itself". The A-brain is connected to the real world in order to sense and act upon what is happening there. The B-brain, in turn, is connected to the A-brain, so that the A-brain is the B-brain's "world". The B-brain can then "see" and influence what happens inside the A-brain.

The BaBe agent architecture is based on Minsky's model and "watches itself" in the way described above. A belief propagation agency can be viewed as an A-Brain that is connected to the real world. As soon as evidence is received from the environment, belief is propagated

through the underlying Bayesian behaviour network. The competence agencies can be viewed as constituting the "B-Brain". These agencies can "see" inside the "A-Brain" by inspecting the beliefs of nodes and acting upon these beliefs and possibly changing the state of the environment, influencing the beliefs propagated by the belief propagation agency – the "A-Brain".

The BaBe methodology involves the design of the initial topology of the Bayesian behaviour network used by the belief-propagation agency (the "A-Brain") and the design of the competence agencies (the "B-Brain"). This methodology extends the Gaia methodology and the coordination-oriented methodology in order to cope with emergence. This chapter describes the models in the Gaia methodology and the coordination-oriented methodology, which form the basis of the BaBe methodology. It then describes how these models are adapted in the BaBe methodology, and describes the added emergence model that copes with emergence.

7.2 The Gaia Models

The Gaia methodology involves a process of generating increasingly detailed models during the analysis and design phase. These models are concerned with the design of individual agents and the communication between them.

Figure 30 illustrates the different models in the Gaia analysis and design phases. During the Gaia analysis phase, the following models are defined (Wooldridge et al., 2000):

- roles model - A description of each role in terms of responsibilities, permissions, interaction protocols, and activities;
- interactions model – description of each protocol in terms of data exchanged and partners involved.

During the Gaia design phase, the following models are defined (Wooldridge et al., 2000):

- agent model - the complex agent types and instances composing the system, where each agent type can be viewed as a set of agent roles;
- services model - the services to be provided by each complex agent in order to realize its role;
- acquaintance model - a description of the lines of communication between different complex agents, in terms of symbolic communication protocols.

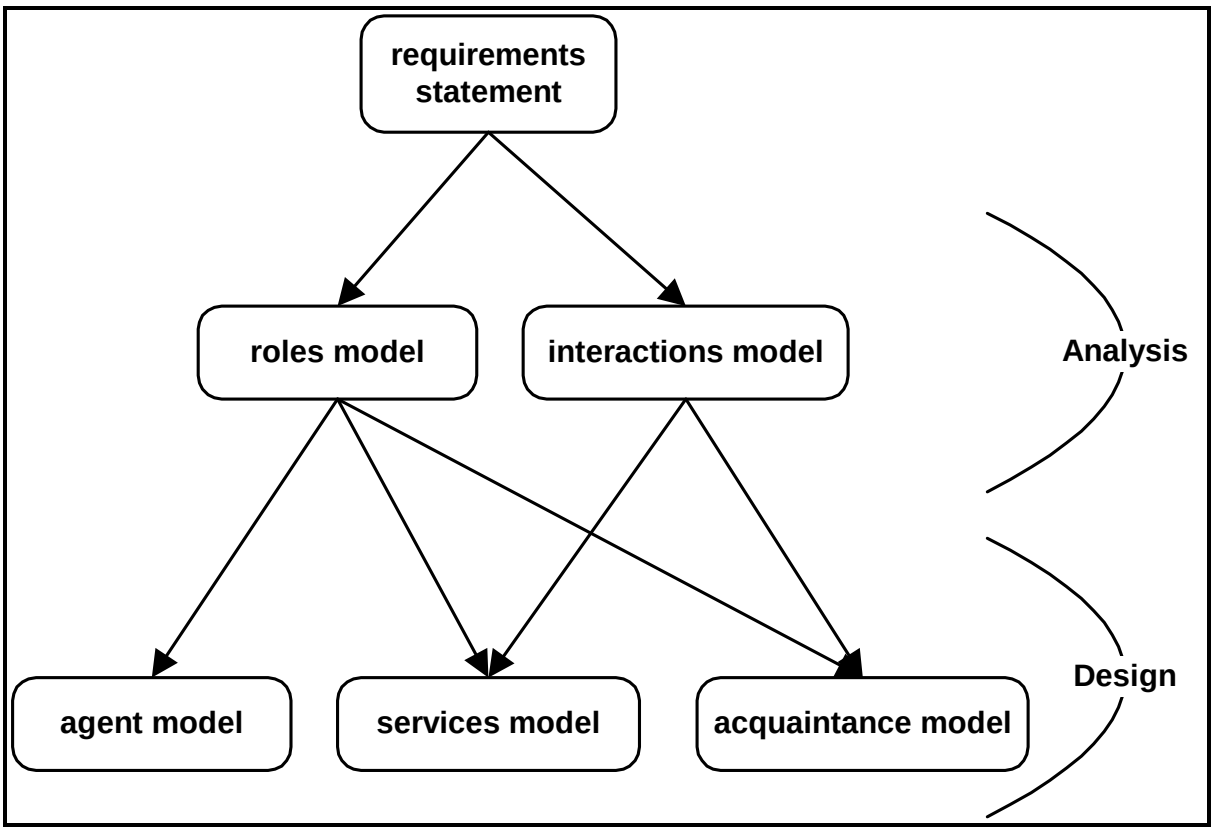

Figure 30: Relationships between the Gaia Models (Wooldridge et al., 2000)

We summarize the Gaia methodology, as described by Wooldridge et al. (2000), below:

The Analysis Process:

1. Identify the *roles* of the system. These roles will correspond to:
   - Individuals, either within an organization or acting independently;
   - Departments within an organization;
   - Organizations themselves.

   Output: A prototypical roles model – a list of key roles that occur in the system, described informally.
2. For each role, document the associated *protocols.* These protocols are the patterns of interaction that occur between the various roles.

Output: An interaction model, capturing the inter-role interactions.

3. Using the protocols as basis, elaborate the roles model.

   Output: A fully elaborated roles model, documenting the key roles in the system, their permissions and responsibilities, together with the protocols and activities in which they participate.

4. Iterate steps 1-3 above.

The Design Process:

5. Create a *complex agent model:*
   - Aggregate roles into complex agent types, and refine to form an agent type hierarchy;
   - Document the instances of each agent type.
6. Develop a services model from activities, protocols and roles.
7. Develop an acquaintance model from the interaction model and agent model.

7.3 Coordination Models

The coordination-oriented methodology is based on the Gaia methodology. Zambonelli et al. (2000) extended the Gaia methodology in order to make it more suitable to open environments as they claimed that most current agent-oriented methodologies are ill suited to open environments. The coordination-oriented methodology added global laws that agents in an agency must obey when interacting with other agents.

Zambonelli et al.'s models consist of three elements, namely

- ***the coordinables:*** entities whose mutual interaction is ruled by the model (the agents);
- ***the coordination media:*** the abstractions enabling agent interactions, as well as the core around which components are organized. Examples include semaphores, monitors, channels and blackboards;
- ***the coordination laws:*** the behaviour of coordination media in response to interaction events.

Figure 31 illustrates the coordination models in the coordination-oriented methodology. These models are the same as in the Gaia methodology with the social model added in the

analysis phase, and the acquaintance model replaced by a model of the behaviour of the coordination media in the design phase. The social model defines social laws that must be obeyed by agents. The social model, together with the interactions model, forms the basis for the model defining the expected behaviour of the coordination media.

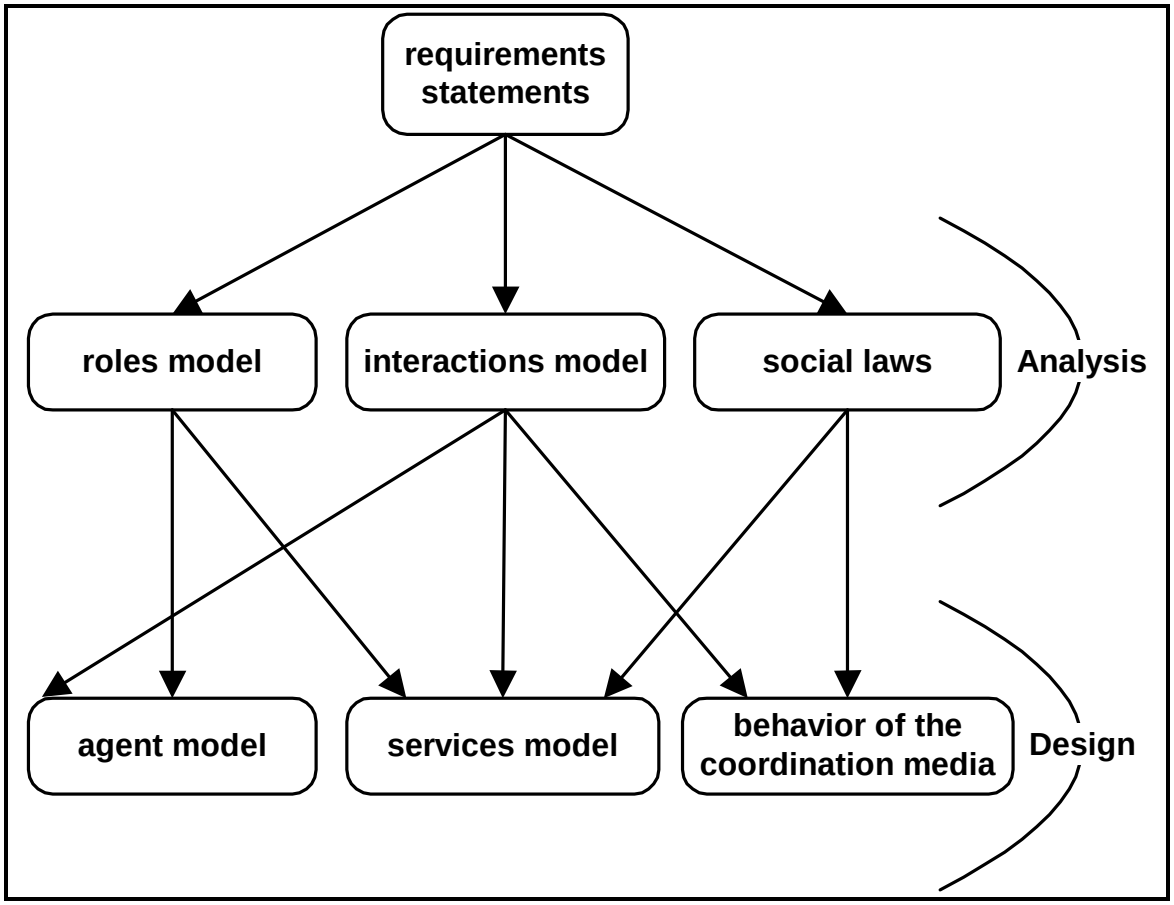

Figure 31: Coordination Models (Zambonelli et al., 2000)

7.4 The BaBe Models

7.4.1 Overview

The BaBe methodology involves the modelling of different aspects of the system at different levels of abstraction, extending the Gaia and coordination-oriented models.

The BaBe coordination model elements are:

- ***the coordinables:*** the entities involved in mutual interaction, are:
  - the belief propagation agents;
  - the node components;
  - the link components; and
  - the competence agencies and the competence components.

  The belief propagation agents, node components and link components are part of the BaBe agent architecture. Only the competence agencies and the competence components need to be addressed in the BaBe methodology.
- ***the coordination media:*** The components that enable agent interactions form part of the BaBe agent architecture. These components include:
  - belief propagation agents, communicating with each other using simple tags placed on communication queues implementing Bayesian behaviour network links; and
  - node and link components administering shared data in a database.
- ***the coordination laws:*** the behaviour of coordination media in response to interaction events. The BaBe agent architecture provide interfaces enabling the following interactions:
  - ***interactions with the environment:*** evidence from the environment is presented to applicable node components as soon as it is received. If the Bayesian behaviour network is in learning mode, the conditional probability matrices are incrementally updated with the new evidence. Each node component administers evidence history, the newest evidence as well as the conditional probability matrices for that node.
  - ***interactions between belief propagation agents, node and link components:*** evidence received from the environment triggers belief propagation. The belief propagation agents calculate beliefs using Judea Pearl's belief propagation algorithm. Belief propagation agents communicate by sending and receiving simple tags on message queues representing the Bayesian behaviour network topology. These tags trigger the update of π and λ values for each link and each node given the

environmental evidence. The π and λ values are stored in a database, administered by node and link components. The belief of each node is a normalized product of the π and λ values for that node.

- ***interactions between competence agencies and node components:*** the competence agencies monitor the beliefs of nodes, revised by the belief propagation agents during each belief propagation cycle. Depending on the beliefs of the nodes the competence agencies decide if component behaviours must be activated or not.

The BaBe models and the relationships between them are illustrated in Figure 32.

During the BaBe analysis phase, the following models are defined:

- *environment model* – description of observable behaviours of coordinables and events as objects with states.
- *roles model* – the description of each role in terms of responsibilities, permissions and activities.
- *interactions model* – interactions between roles, and between roles and the environment.

During the BaBe design phase, the following models are defined:

- *agency model* – this model describes the competence agencies;
- *internal behaviour model* – this model describes the initial configuration of the Bayesian behaviour network(s) and describes how evidence from the environment will be presented to the network(s);
- *external behaviour model* – this model describes the competence of each competence agency.

During the BaBe execution phase, the emergence model is implemented by the Bayesian agencies. This model functions as the internal model of the complex adaptive system.

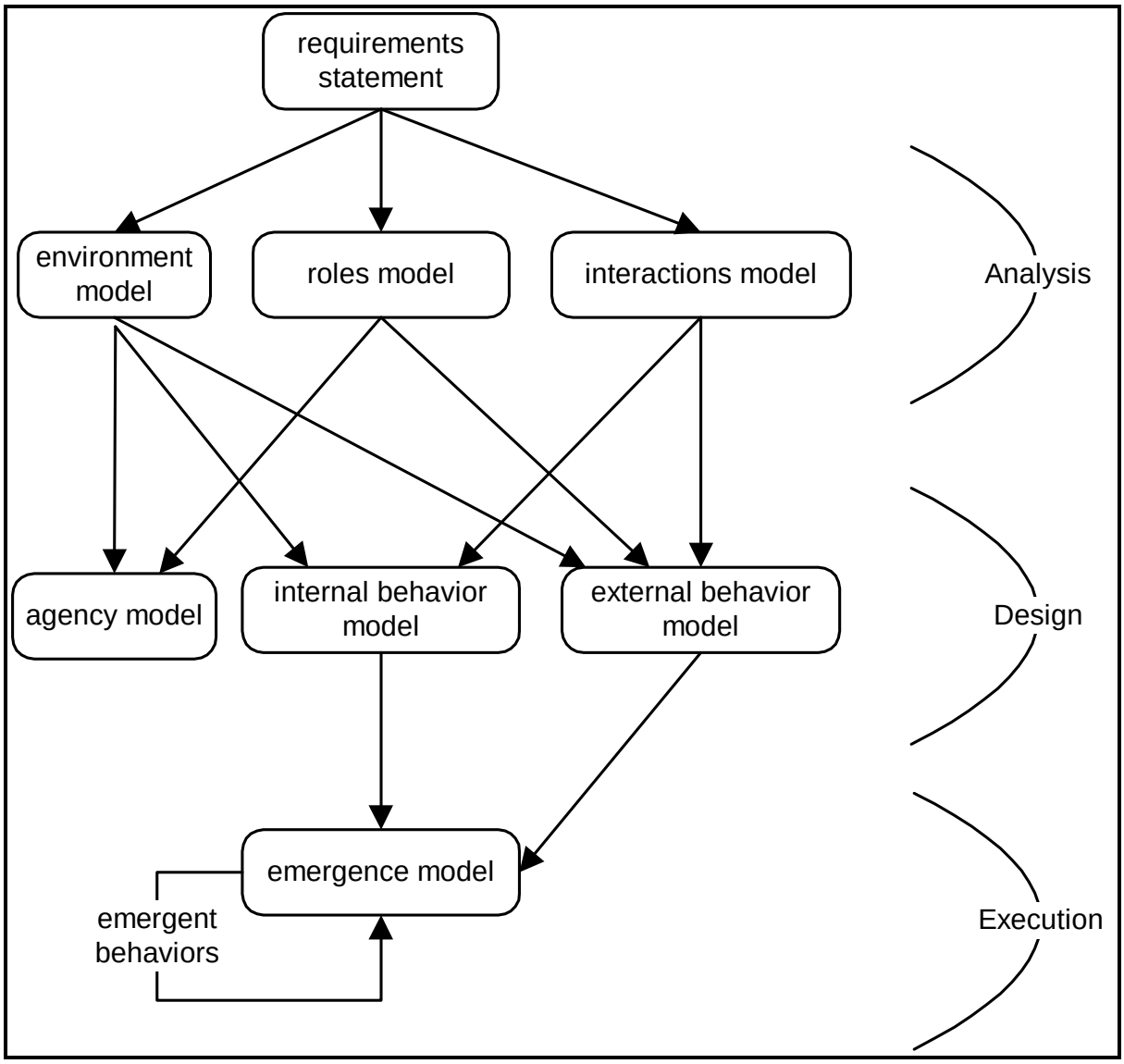

Figure 32: The BaBe Methodology

7.4.2 The Environment Model

The environment model must capture all the known state changes and events that can occur in the environment as objects with states.

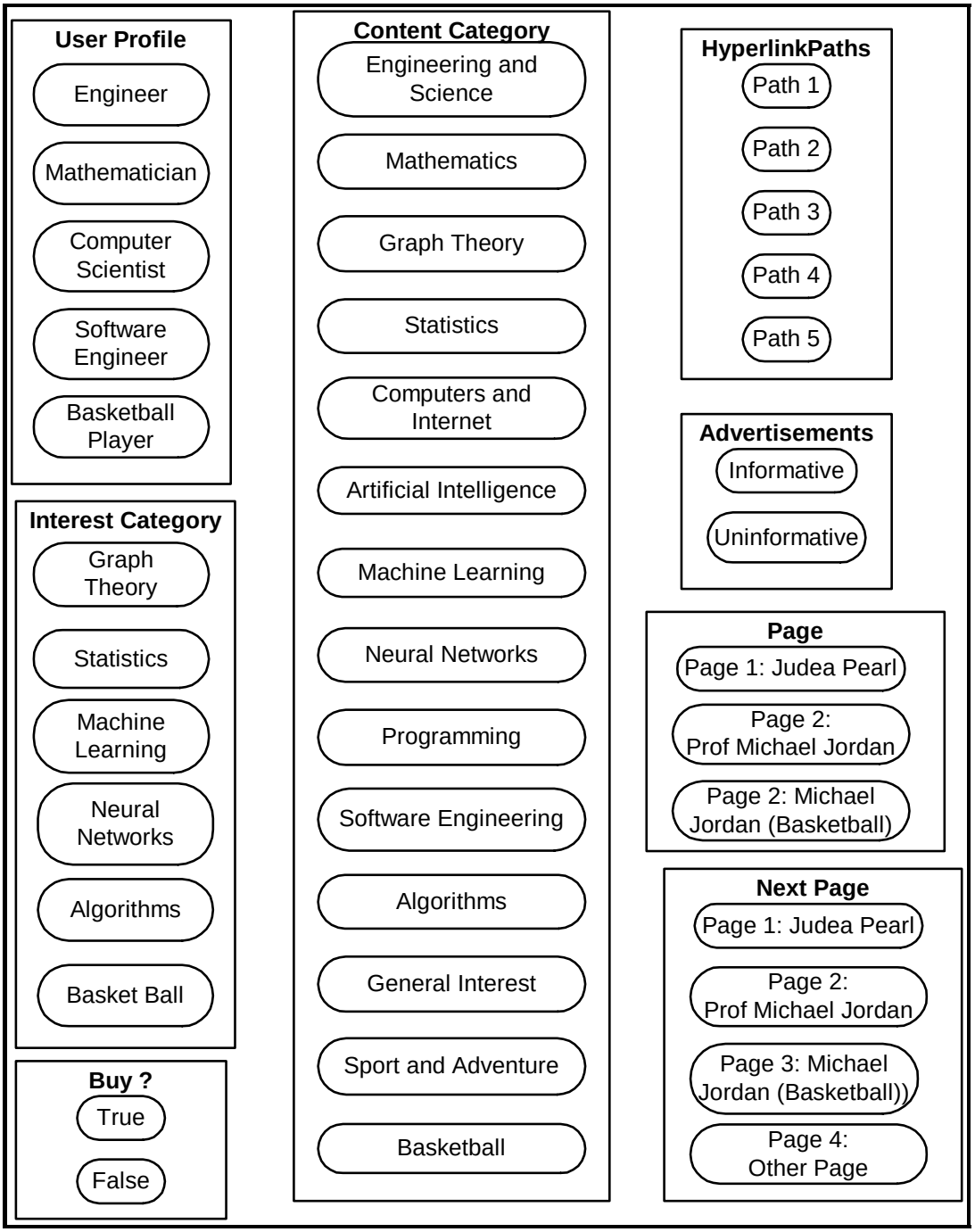

Figure 33: Example Environment Model

For each object, the states of the objects that are known can be indicated. In the analysis phase, the states will most probably not be known, as some states might only emerge during the execution phase. In Figure 33, example environmental objects are illustrated.

7.4.3 The Roles Model

This model describes each role in terms of responsibilities, permissions and activities. A role is created to do something in reaction to environmental states. In the submission for UML2.0, Financial Systems Architects, MEGA International, Mercury ComputerSystems & TogetherSoft (2002) describe a role as follows:

> *A role is used to specify an action, association or relationship without identifying a particular participant in that action, association or relationship. That is, a role serves as a placeholder, used when the modeller does not want to be more specific at that place in the model.*

The BaBe roles model is nearly the same as the Gaia roles model, but differs with respect to one characteristic. In Gaia, a role is described in terms of four attributes, namely responsibilities, permissions, activities and protocols. In BaBe, a role is described in terms of three attributes, namely responsibilities, permissions and competence. These characteristics are illustrated in Figure 34.

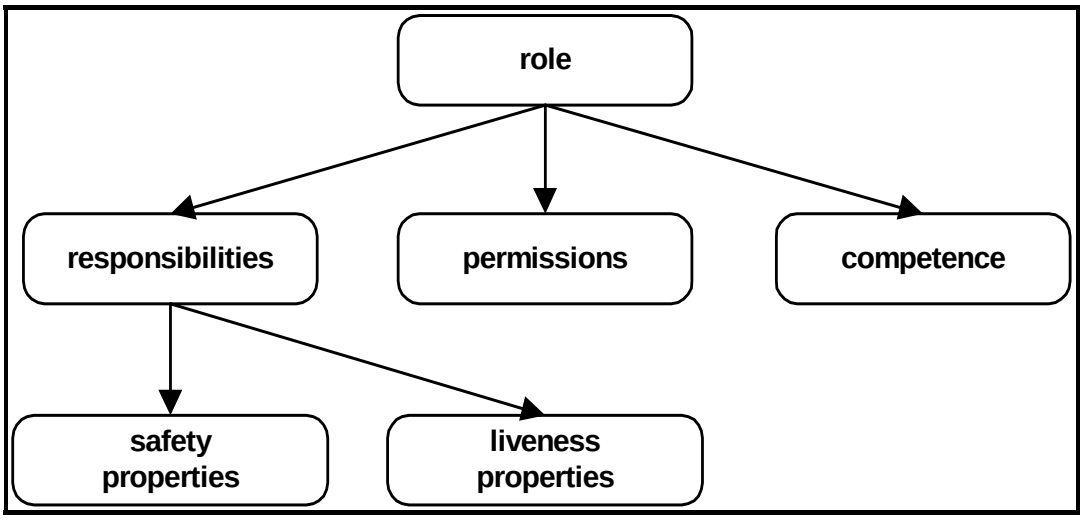

Figure 34: The BaBe Roles Model

A responsibility determines functionality. Wooldridge et al. distinguish between two types of responsibilities, namely liveness and safety properties, and describe these properties as follows:

- *Liveness properties* describe what must be achieved by a given role, given certain environmental conditions; and
- *Safety properties* impose constraints on the behavioural states of a role.

Permissions are the "rights" associated with each role, that is a specification of the resources available to the role in order to realise its responsibilities. These resources are the objects described by the environment model.

A *competence* specifies the computations associated with each role. (In Gaia, *activities* refer to computations carried out without interacting with other roles, and *protocols* are computations carried out through interaction with other roles).

The BaBe roles model can be documented using a set of role schemata. These role schemas are similar to the Gaia role schemas. See Figure 35.

Role Schema	***name of role***
Description	*short English description of role*
Competence	*activities in which the role plays a part*
Permissions	*"rights" associated with the role*
Responsibilities	
Liveness	*Liveness responsibilities*
Safety	*safety responsibilities*

Figure 35: BaBe Role Schema

Object Process Diagrams (Sight Code Inc., 2001) are ideally suited to visualize role models. Figures 9 and 12 are examples of Object Process Diagrams (OPD's). In these diagrams, processes are represented by ellipses, informational objects by rectangles, and states of these objects by rounded rectangles within the rectangles that represent the objects.

Liveness responsibilities can be represented using an OPD as in Figure 36. In this OPD, the example role achieves states p1 and q1, given environmental states m2 and n2.

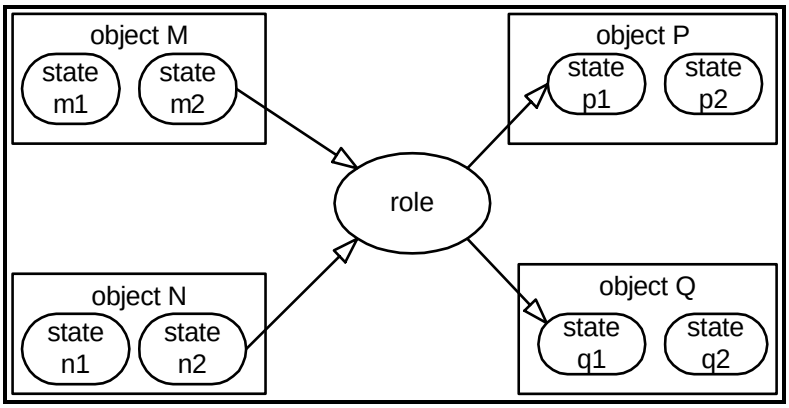

Figure 36: Liveness Properties OPD

Safety responsibilities can be represented using an OPD as in Figure 37. In this OPD, object M must be in state m2 and object N must be in state n2 for the role to achieve its competence.

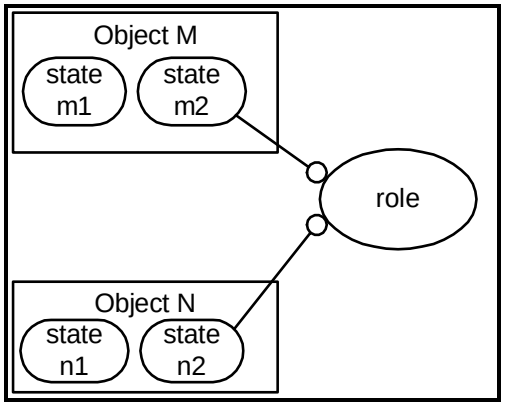

Figure 37: Safety Properties OPD

Figure 38 illustrates an example role for the bookstore example. Using the states of the interest category object, as well as the states of the content categories determined by traversed hyperlinks, the role personalizes the web page and outputs the changed web page contents.

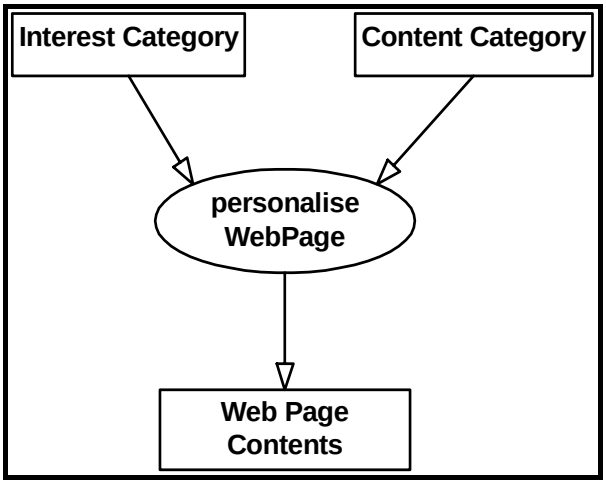

Figure 38: Example Role

7.4.4 Interactions Model

The interactions model captures the interactions between roles. These interactions are not restricted to symbolic communication protocols. Roles can interact through states of objects, as illustrated in Figure 12. Roles can also interact through the exchange of messages, which can be illustrated using OPD diagrams or any of the UML behaviour diagrams.

A number of researchers view agents as next generation objects or components (Odell, Van Dyke Parunak and Bauer, 2001) and is extending the UML to Agent UML (AUML), which will make it more suitable to the specification of agent interactions and interaction protocols.

The BaBe agent architecture enables causal interaction between objects with states. In the interactions model, one or more Bayesian behaviour networks can be constructed, using the objects in the environment model. The system designer need not worry about the interactions between agents during belief propagation, as it is inherent in the BaBe agent architecture.

7.4.5 Agency Model

The agency model consists of the definition of the competence sets to be implemented by the competence agencies. Each competence agency implements a competence set $\Theta_i = (C_i, A_i)$, where C_i is a set of constraints on a subset of Bayesian behaviour network nodes and their states, and the A_i is a set of component actions that must be executed if all the constraints in C_i are met. Each competence agency consists of a set of competence components. The

agency accesses the node component interfaces of the nodes in the constraint set in order to determine if the component behaviours must be executed or not. (As an example, see Figures 25-27 and the accompanying descriptions).

7.4.6 Internal Behaviour Model

This model consists of one or more Bayesian behaviour network(s) and the sources of evidence from the environment that will be presented to Bayesian behaviour network nodes, using the interfaces of the node-components.

7.4.7 External Behaviour Model

This model describes the competences of the competence agencies. Each of the component actions in the competence sets of the competence agencies must be described in detail. In an enterprise, example component behaviours will include parts of the business processes or workflows in the enterprise. These behaviours must be packaged into one or more re-usable components - the competence components.

7.4.8 The Emergence Model

The emergence model consists of Bayesian behaviour networks, initialised by the software engineer and implemented by the Bayesian agencies. The belief propagation agencies will incrementally update the networks during learning by updating the conditional probability matrices or discovering new links with each new set of evidence received from the environment. The competence agencies inspect the beliefs of nodes and act upon these beliefs by activating competence components. The actions of the competence components can change the state of the environment, influencing the collective learning and belief propagation in the belief propagation agencies. The feedback arrow to the emergence model in Figure 32 indicates this process.

The Bayesian behaviour networks can be viewed as the language $\mathcal{L}_1$ described in the emergence test defined by Ronald et al. (1999). Bridging the $\mathcal{L}_1$ - $\mathcal{L}_2$ gap involves the learning of the conditional probability matrices from the observed external and internal behaviours, as well as the discovery of new relationships between behaviours.

7.5 The BaBe Methodology

7.5.1 Graphical Diagrams

The BaBe models can be defined using any of the following, where applicable:

- Use case diagrams (UML)
- BaBe Role Schemas (derived from Gaia Role Schemas) - Example: Figure 35
- Class diagrams (UML)
- Behaviour Diagrams
  - Object Process Diagrams (Sight Code Inc., 2001) - Examples: Figures 9, 12, 36, 37, 38
  - State chart diagrams (UML) - Examples: Figures 10, 11, 23 and 24
  - Activity diagrams, Interaction Diagrams, Sequence Diagrams, Collaboration Diagrams (UML)
  - Bayesian behaviour networks (BaBe) - Example: Figure 17
- Implementation Diagrams
  - Component Diagrams (UML) - Examples: Figures 21 and 22
  - Deployment Diagrams (UML)

7.5.2 The Analysis Process

1. Identify observable behaviours of coordinables and events in terms of objects with states.
   Output: An environment model.
2. Identify the *roles* of the system.
   Output: A prototypical roles model – a list of key roles that occur in the system, described informally.
3. For each role, document the associated patterns of interactions that occur between the various roles.
4. Design the causal relationships between objects.

   Output: An interaction model, capturing the inter-role interactions and a prototypical Bayesian behaviour network.

5. Elaborate the roles model using the Bayesian behaviour network as basis.
   Output: A fully elaborated roles model, documenting the key roles in the system, their permissions and responsibilities, together with their competences.
6. Iterate steps 1-5 above.

7.5.3 The Design Process

1. Create an *agency model*
   a. Aggregate roles into competence sets, and assign a competency agency to each competence set;
   b. Document each competence agency.

   Output: An Agency Model, documenting the competence sets and their associated competence agencies.
2. Elaborate the Bayesian behaviour network defined in the interactions model, as well as specifications of all interactions presenting evidence to the Bayesian behaviour network nodes.

   Output: An Internal Behaviour Model documenting the Bayesian behaviour network and its interfaces with the environment.
3. Design the competences to be provided by each agency in order to realize its role.

   Output: An External Behaviour Model, documenting all the component actions in the competence sets of the competence agencies.
4. Iterate steps 1-3 above

7.5.4 The Execution Phase

7.5.4.1 A General Application implementing the Execution Phase of the BaBe Methodology

A general application implementing the execution-phase of the BaBe methodology is illustrated in Figure 39. This is a subset of the general Babe application illustrated in Figure 29.

Using the Bayesian Network Administrator Client, as well as the Agency Model generated during the design phase, the software engineer manually initializes or modifies the Bayesian

behaviour network configuration and configures the competence agencies, so that it checks the constraints on the Bayesian network nodes to decide if competence components must be activated or not.

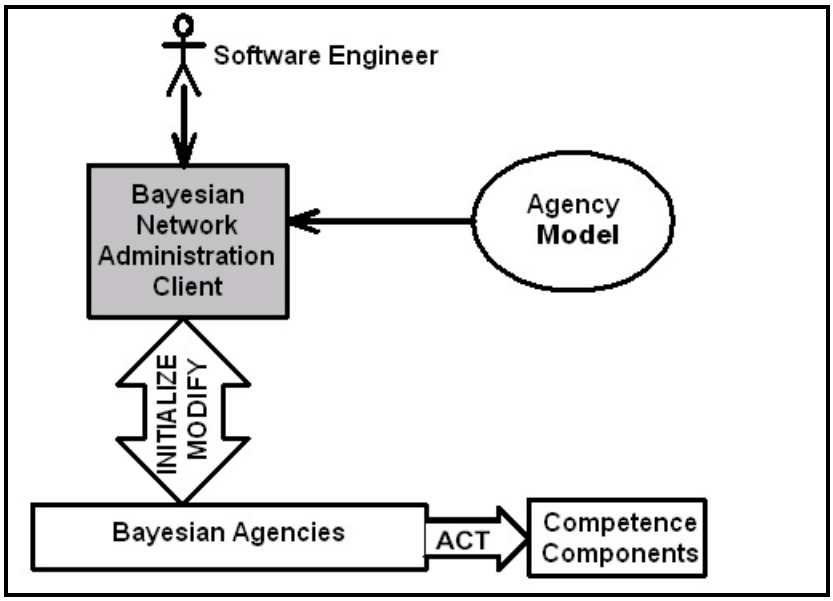

Figure 39: General Application for the BaBe Methodology Execution Phase

The Bayesian agencies consist of the competence agencies and belief propagation agencies that are deployed together. They share node components, updated by the belief propagation agents and queried by the competence agencies to determine if their associated competence components must be activated or not.

7.5.5 The Limitations of the BaBe Execution Phase Application

The limitations of the execution phase application are caused by the static nature and inflexibility of the J2EE platform. If the structure of the Bayesian behaviour network underlying the Bayesian agencies is modified, or if the competence agencies are changed, the entire BaBe agent architecture must be redeployed. These limitations will be overcome by deploying BaBe using a reflective component-based framework.

7.5.6 The Benefits of the BaBe Execution Phase Application

Using the BaBe execution phase application, the user will be able to visualize the emergence model. The software engineer will be able to view the effect of the emergent behaviours on the emergence model illustrated in Figure 32. This interface will enable the software

engineer to update the Bayesian behaviour network and competence agencies if iterating through the analysis and design phases causes the agency model and internal behaviour model to change.

7.5.7 How and where can the BaBe Execution Phase Application be used?

The BaBe execution phase application can be used to engineer the emergence model in an interactive fashion in any environment that the BaBe agent architecture is deployed, assisting it to control the behaviour of the system in response to emergence. The software engineer can visualize how the Bayesian agencies organize themselves into distributed Bayesian behaviour networks, collectively modelling the dynamics of change.

Using this application, the software engineer can view the complex web of interrelationships that the Bayesian agencies mined from their environment, including for example relationships between products and services, business processes and business units; specific projects and project implementations; customers and the marketplace. The software engineer can then use this understanding to adapt the competence agencies assisting the enterprise to adapt its business processes and to formulate new knowledge or business strategies.

7.6 Conclusion

The BaBe methodology modifies and extends the Gaia and the coordination-oriented methodologies. It includes Gaia models and coordination models, and has an additional emergence model.

The BaBe methodology uses the Gaia and coordination-oriented roles and interactions models. The coordination-oriented methodology added the social laws model to the Gaia methodology, which our BaBe methodology replaced with an environmental model. This model captures all the known states of the environment. BaBe modifies the Gaia roles model by replacing Gaia's activities and protocols with competences. These competences include general interactions and activities and are not restricted to symbolic communication protocols.

The BaBe agency model replaces the Gaia and coordination-oriented agent model. The competences apply to agencies rather than individual agents. The agency model consists of the competence sets defined in terms of constraints on the beliefs of the Bayesian behaviour network nodes and actions that must be performed should the constraints be met.

The BaBe internal behaviour model replaces the Gaia and coordination-oriented services model. This model defines all the interactions between the Bayesian behaviour network nodes and the environment, for example how environmental evidence will be presented to the network nodes, and how the network will be queried.

The BaBe external behaviour model replaces the Gaia acquaintance model and coordination model that defines the behaviour of the coordination media. This model refines the competences of the competence agencies.

The BaBe analysis and design models can be constructed using any suitable graphical diagrams, such as the UML use case diagrams, state chart diagrams and so forth.

The BaBe methodology has an additional run-time model, namely the emergence model. This model consists of Bayesian behaviour networks, initialised by the software engineer and implemented by the Bayesian agencies. The belief propagation agencies incrementally update the networks during learning by updating the conditional probability matrices with each new set of evidence received from the environment. The competence agencies inspect the beliefs of nodes and act upon these beliefs by activating competence components. The actions of the competence components can change the state of the environment, influencing the collective learning and belief propagation in the belief propagation agencies. The collective self-awareness of these agencies assists the human software engineer to bridge the gap between the implementation and the understanding of emergent behaviour of the BaBe adaptive agent architecture.

We described a J2EE-based general application that implements the execution phase of the BaBe methodology. This application can be used by the software engineer to interact with the emergence model, visualizing the effects of emergence and adapting the behaviour of the architecture if necessary. The J2EE platform is static because it requires the entire BaBe agent architecture to be redeployed if the software engineer modifies the configuration of the Bayesian behaviour network, the competence agencies or competence components.

If the BaBe architecture is deployed in a reflective component-based framework that allows runtime configuration of components and middleware services, the modifications to the emergence model made by the software engineer can take immediate effect as a reflective component-based framework allows dynamic reconfiguration and deployment.

Chapter 8

BaBe: The Prototype

8.1 Overview

With the prototype implementation of the BaBe agent architecture, we wanted to prove that it is possible to use a commercially available component architecture to implement an adaptive agent architecture.

As a detailed study of Bayesian belief propagation and learning falls outside the scope of this research, our prototype implementation includes learning in Bayesian networks with known structure and no hidden variables and belief propagation in singly-connected polytrees using Judea Pearl's belief propagation algorithm.

We implemented prototype node and link components using EJB entity beans. The prototype node component implements the component specified in Figure 21. The prototype link component implements the component specified in Figure 22. The prototype belief propagation agents are all identical EJB message beans, operating in accordance with the specification in Figure 23. These agents listen on different JMS queues, corresponding to the links of the Bayesian behaviour network.

In this chapter, we illustrate the operation of our prototype BaBe components using the Bayesian behaviour network illustrated in Figures 17 and 18. We describe how we manually configured this Bayesian behaviour network and how we manually deployed the belief propagation agents. We include screen dumps of the original beliefs (marginal probabilities) of the nodes. We then illustrate how we set evidence and the resulting belief propagation, taking this evidence into account, without any learning taking place. After this, we illustrate the setting of evidence and the resulting belief propagation, taking this evidence into account, this time with learning.

In Chapter 6, we described the competence sets and the competence agencies. We illustrated the functioning of an example competence component, and will not repeat the description in this chapter.

8.2 Configuring the Bayesian Behaviour Network

The configuration of the Bayesian behaviour network was done manually. The JMS queues for the links in Figure 17 were created using the j2eeadmin tool, as follows:

```
j2eeadmin -addJmsDestination DF queue
j2eeadmin -addJmsDestination EF queue
j2eeadmin -addJmsDestination CD queue
j2eeadmin -addJmsDestination AB queue
j2eeadmin -addJmsDestination FG queue
j2eeadmin -addJmsDestination FH queue
j2eeadmin -addJmsDestination BG queue
 j2eeadmin -addJmsFactory DFCF queue
 j2eeadmin -addJmsFactory EFCF queue
j2eeadmin -addJmsFactory CDCF queue
 j2eeadmin -addJmsFactory ABCF queue
j2eeadmin -addJmsFactory FGCF queue
 j2eeadmin -addJmsFactory FHCF queue
j2eeadmin -addJmsFactory BGCF queue
```

Figure 40: JMS Link Definitions

Figure 41 shows the assignment of the belief propagation agents to the JMS queues, using the J2EE deployment tool.

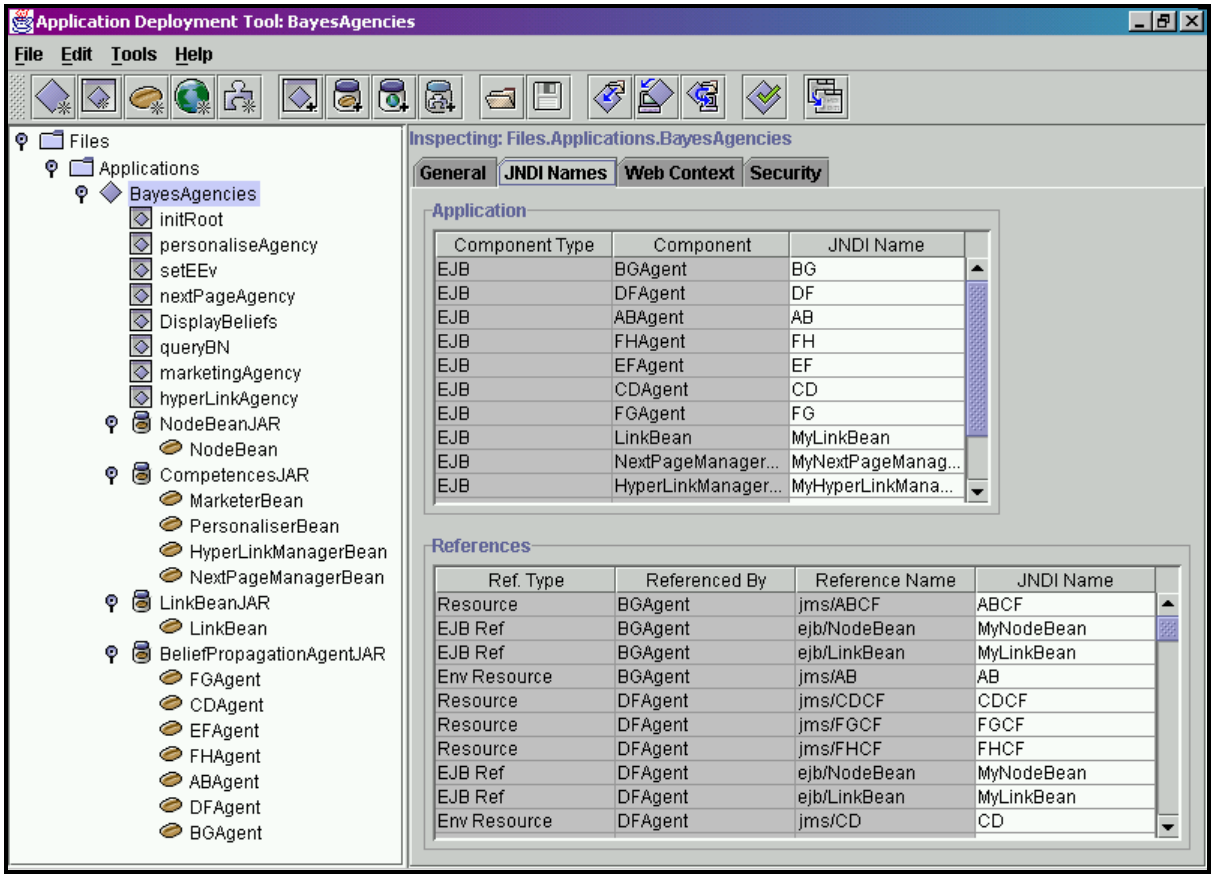

Figure 41: Bayes Components

Figure 42 shows the output of the J2EE server on start-up, with the belief propagation agents listening on the JMS queues representing the links in the Bayesian behaviour network.

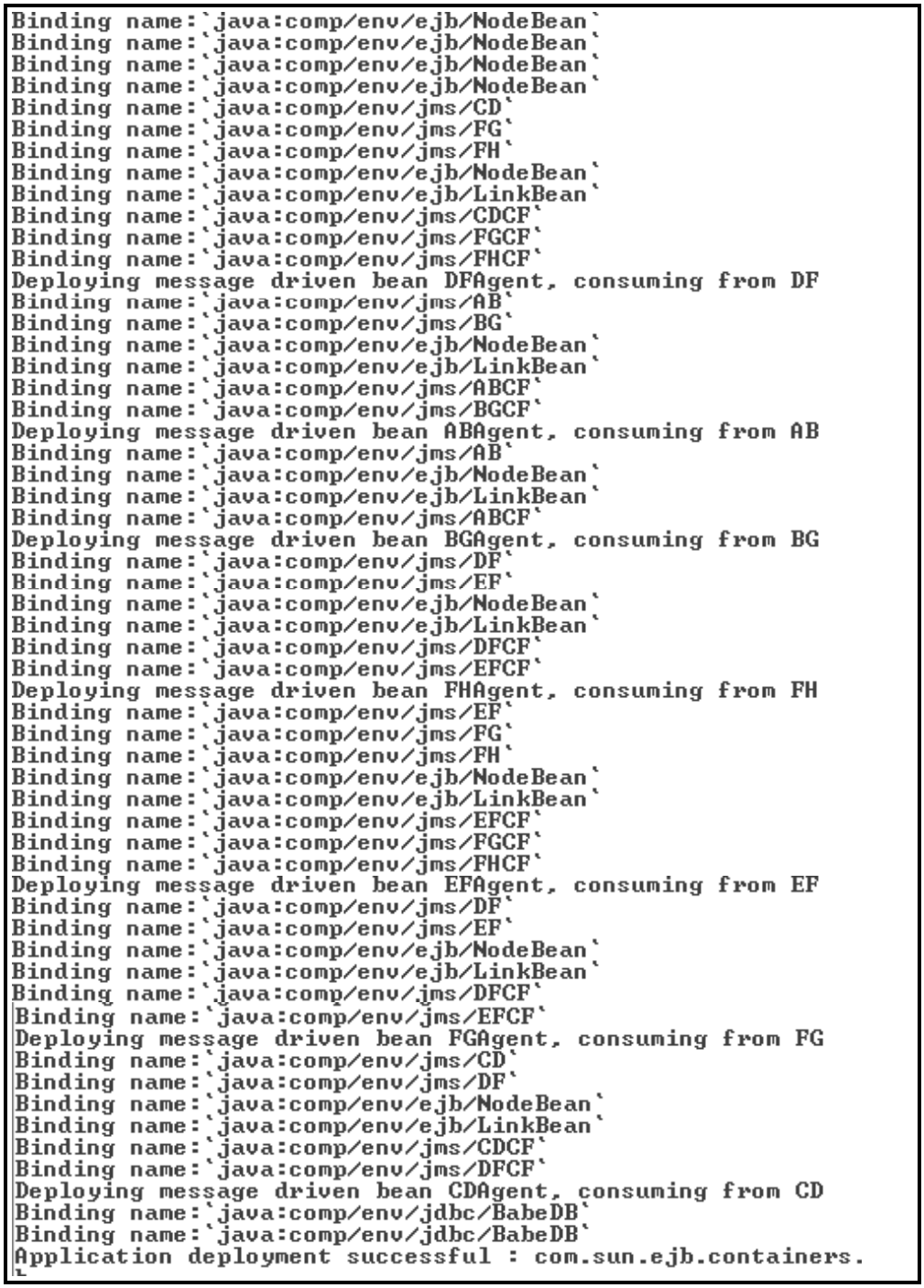

Figure 42: J2EE Server Start-up Trace

Figure 43 displays the beliefs of the Bayesian network in the absence of evidence. These beliefs are also illustrated in Figure 17.

```
Initiating login ...
Binding name: `java:comp/env/ejb/NodeBean`

*******Beliefs : NODE: A : User Profile
Engineer                    : 0.25
Mathematician               : 0.125
ComputerScientist           : 0.25
SoftwareEngineer            : 0.25
BasketballPlayer            : 0.125

*******Beliefs : NODE: B : Interest Category
GraphTheory                 : 0.1325
ProbStats                   : 0.14375
MachineLearning             : 0.12625
NeuralNetworks              : 0.21375000000000002
Algorithms                  : 0.21375000000000002
BasketBall                  : 0.16999999999999998

*******Beliefs : NODE: C : Hyperlink Paths
Path1                       : 0.1
Path2                       : 0.1
Path3                       : 0.1
Path4                       : 0.2
Path5                       : 0.5

*******Beliefs : NODE: D : Content Categories
EngAndScience               : 0.04999999999999999
Mathematics                 : 0.04999999999999999
GraphTheory                 : 0.04999999999999999
ProbAndStats                : 0.04999999999999999
ComputersAndInternet        : 0.06
AI                          : 0.02
MachineLearning             : 0.02
NN                          : 0.04
Programming                 : 0.04
SoftwareEngineering         : 0.04
Algorithms                  : 0.08
GeneralInterest             : 0.09999999999999998
SportAndAdv                 : 0.09999999999999998
BasketBall                  : 0.29999999999999993

*******Beliefs : NODE: E : Advertisements
Informative                 : 0.7
NotInformative              : 0.3

*******Beliefs : NODE: F : Page
BayesianNetworks            : 0.2868998606995821
MJordan(Professor)          : 0.2700998102994309
MJordan(Basketball)         : 0.443000329000986 96

*******Beliefs : NODE: G : Buy
True                        : 0.3531260131186103
False                       : 0.6468739868813898

*******Beliefs : NODE: H : Next Page
JudeaPearl                  : 0.1890175784899392
MJordan(Professor)          : 0.2007743614422307
MJordan(BasketBall)         : 0.0
OtherPage                   : 0.6102080600678301
```

Figure 43: Original Beliefs – No Evidence

8.3 Querying the Bayesian Behaviour Network

Figure 44 displays the output of a client setting evidence in order to query the Bayesian behaviour network. In query mode, the node components do not learn from the evidence presented to them.

```
Initiating login ...
Binding name:`java:comp/env/jms/FG`
Binding name:`java:comp/env/jms/FH`
Binding name:`java:comp/env/jms/BG`
Binding name:`java:comp/env/ejb/NodeBean`
Binding name:`java:comp/env/ejb/LinkBean`
Binding name:`java:comp/env/jms/BGCF`
Binding name:`java:comp/env/jms/FGCF`
Binding name:`java:comp/env/jms/FHCF`

        QUERYING BAYES BEANS

        ====================


    ******    NODE A : User Profile
    Setting Evidence for Query to : Mathematician

    ******    NODE B : Interest Category
    No Evidence for Query

    ******    NODE C : Hyperlink Paths
    No Evidence for Query

    ******    NODE D : Content Categories
    No Evidence for Query

    ******    NODE E : Advertisements
    No Evidence for Query

    ******    NODE F : Page
    Setting Evidence for Query to : BayesianNetworks

    ******    NODE G : Buy
    No Evidence for Query

    ******    NODE H : Next Page
    No Evidence for Query
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
Sending LAMBDAto queue java:comp/env/jms/BG
Sending LAMBDAto queue java:comp/env/jms/FG
Sending LAMBDAto queue java:comp/env/jms/FH
```

Figure 44: Setting Evidence for a Query

Figure 45 is the output trace of the belief propagation agency, in response to the evidence presented to it in Figure 44 above. This evidence is also illustrated in Figure 18.

```
MessageBean : BG received a MessageLAMBDA
MessageBean : FG received a MessageLAMBDA
MessageBean : FH received a MessageLAMBDA
BG: LAMBDA : [
3.0,
3.0,
3.0,
3.0,
3.0,
3.0,
FH: LAMBDA : [
1.25,
1.0,
1.0,
BG:Can now calculate B's LAMBDA !
FG: LAMBDA : [
6.0,
6.0,
6.0,
FH:Can not calculate F's LAMBDA yet !
FG:Can now calculate F's LAMBDA !
B: LAMBDA : [
3.0,
3.0,
3.0,
3.0,
3.0,
3.0,

 Propagating LAMBDA message to AB from BG
MessageBean : AB received a MessageLAMBDA
F: LAMBDA : [
7.5,
6.0,
6.0,

 Propagating LAMBDA message to EF from FG

 Propagating LAMBDA message to DF from FG
MessageBean : EF received a MessageLAMBDA
MessageBean : DF received a MessageLAMBDA
AB: LAMBDA : [
3.0,
3.0000000000000004,
3.0,
3.0,
2.9999999999999996,
AB:Can now calculate A's LAMBDA !
EF: LAMBDA : [
DF: LAMBDA : [
43.725000000000001,
6.249975,
34.999649999999995,
6.249975,
6.249975,
6.249975,
6.249975,
6.249975,
6.249975,
8.499975,
6.249975,
6.249975,
6.249975,
2.5749750000000002,
2.5749750000000002,
2.5749750000000002,
EF:Can now calculate E's LAMBDA !
DF:Can now calculate D's LAMBDA !
A: LAMBDA : [
3.0,
3.0000000000000004,
3.0,
3.0,
2.9999999999999996,
```

```
E: LAMBDA : [

 Propagating PI message to AB from AB
D: LAMBDA : [
6.249975,
6.249975,
6.249975,
6.249975,
6.249975,
6.249975,
6.249975,
43.725000000000001,
34.999649999999995,
8.499975,
6.249975,
6.249975,
6.249975,
2.5749750000000002,
2.5749750000000002,
2.5749750000000002,
MessageBean : AB received a MessagePI

 Propagating LAMBDA message to CD from DF

 Propagating PI message to EF from EF
MessageBean : EF received a MessagePI
MessageBean : CD received a MessageLAMBDA
AB: PI : [
0.0,
1.0,
0.0,
0.0,
0.0,
Can now calculate child B's PI !
EF: PI : [
0.7445740602147133,
0.25542593978528677,
EF: Can not calculate F's PI yet !
B's PI : [
0.25,
0.34,
0.1,
0.2,
0.1,
0.01,

 Propagating PI message to BG from AB
CD: LAMBDA : [
6.249975,
6.249975,
7.149975,
6.2499750000000001,
2.25998,
MessageBean : BG received a MessagePI
CD:Can now calculate C's LAMBDA !
C: LAMBDA : [
6.249975,
6.249975,
7.149975,
6.2499750000000001,
2.25998,

 Propagating PI message to CD from CD
BG: PI : [
0.24999999999999997,
0.33999999999999997,
MessageBean : CD received a MessagePI
0.1,
0.2,
0.1,
0.009999999999999998,
BG: Can not calculate G's PI yet !
CD: PI : [
0.14384366777503452,
0.14384366777503452,
0.16455723878892353,
0.28768733555006903,
0.2600680901109384,
Can now calculate child D's PI !
```

```
D's PI : [
0.07192183388751726,
0.07192183388751726,
0.07192183388751726,
0.07192183388751726,
0.09044891486779852,
0.032911447757784706,
0.032911447757784706,
0.06582289551556941,
0.05753746711001381,
0.05753746711001381,
0.11507493422002762,
0.05201361802218768,
0.05201361802218768,
0.15604085406656304,

 Propagating PI message to DF from CD
MessageBean : DF received a MessagePI
DF: PI : [
0.08385738922966482,
0.08385738922966482,
0.08385738922966482,
0.08385738922966482,
0.10545907201604623,
0.038373160632314385,
0.038373160632314385,
0.10437510743503978,
0.06708591138373186,
0.06708591138373186,
0.13417182276746373,
0.024985749712973356,
0.009703297978804 98,
0.07495724913892006,
Can now calculate child F's PI !
F's PI : [
0.4407391615492795,
0.3941101630323268,
0.16514812115899558,

 Propagating PI message to FG from DF

 Propagating PI message to FH from DF
MessageBean : FG received a MessagePI
MessageBean : FH received a MessagePI
FH: PI : [
FG: PI : [
1.25,
0.0,
0.0,
6.0,
0.0,
0.0,
Can now calculate child H's PI !
Can now calculate child G's PI !
H's PI : [
0.0,
4.5,
0.0,
3.0,
G's PI : [
0.6851249999999999,
0.5648749999999999,
```

Figure 45: Belief Propagation Agency Output

Figure 46 shows the beliefs of the nodes after belief propagation (compare with Figure 18)

```
Initiating login ...
Binding name: java:comp/env/ejb/NodeBean

*******Beliefs : NODE: A : User Profile
Engineer                    : 0.0
Mathematician               : 1.0
ComputerScientist           : 0.0
SoftwareEngineer            : 0.0
BasketballPlayer            : 0.0

*******Beliefs : NODE: B : Interest Category
GraphTheory                 : 0.24999999999999997
ProbStats                   : 0.33999999999999997
MachineLearning             : 0.1
NeuralNetworks              : 0.2
Algorithms                  : 0.1
BasketBall                  : 0.009999999999999998

*******Beliefs : NODE: C : Hyperlink Paths
Path1                       : 0.14384366777503452
Path2                       : 0.14384366777503452
Path3                       : 0.16455723878892353
Path4                       : 0.2876873355006903
Path5                       : 0.2600680901109384

*******Beliefs : NODE: D : Content Categories
EngAndScience               : 0.08259513309466833
Mathematics                 : 0.08259513309466833
GraphTheory                 : 0.08259513309466833
ProbAndStats                : 0.08259513309466833
ComputersAndInternet        : 0.10387165840984969
AI                          : 0.03779555193411501
MachineLearning             : 0.03779555193411501
NN                          : 0.1028040101124178
Programming                 : 0.06607610647573468
SoftwareEngineering         : 0.06607610647573468
Algorithms                  : 0.13215221295146937
GeneralInterest             : 0.024609653865578066
SportAndAdv                 : 0.024609653865578066
BasketBall                  : 0.0738289615967342

*******Beliefs : NODE: E : Advertisements
Informative                 : 0.7445740602147133
NotInformative              : 0.25542593978528677

*******Beliefs : NODE: F : Page
BayesianNetworks            : 1.0
MJordan(Professor)          : 0.0
MJordan(Basketball)         : 0.0

*******Beliefs : NODE: G : Buy
True                        : 0.5481
False                       : 0.4519

*******Beliefs : NODE: H : Next Page
JudeaPearl                  : 0.0
MJordan(Professor)          : 0.6
MJordan(BasketBall)         : 0.0
OtherPage                   : 0.4
```

Figure 46: Beliefs of Nodes after Belief Propagation (Evidence and No Learning)

8.4 Learning the Bayesian Behaviour Network

Figure 47 displays the output of a client presenting evidence to the Bayesian behaviour network. In learning mode, the node components learn from the evidence presented to them, in this case that a user that browsed Michael Jordan's the basketball player's page next, bought a product from the web page that he browsed before.

```
        Learning From Evidence

        ======================


    ******    NODE A : User Profile
    No Evidence

    ******    NODE B : Interest Category
    No Evidence

    ******    NODE C : Hyperlink Paths
    No Evidence

    ******    NODE D : Content Categories
    No Evidence

    ******    NODE E : Advertisements
    No Evidence

    ******    NODE F : Page
    No Evidence

    ******    NODE G : Buy
    Setting Evidence to : True

    ******    NODE H : Next Page
    Setting Evidence to : MJordan(BasketBall)
Java(TM) Message Service 1.0.2 Reference Implementation (build b13)
Sending LAMBDAto queue java:comp/env/jms/BG
Sending LAMBDAto queue java:comp/env/jms/FG
Sending LAMBDAto queue java:comp/env/jms/FH
```

Figure 47: Setting Evidence that must be learnt

Figure 48 displays the results after belief propagation in the presence of the evidence presented in Figure 47.

```
Binding name:`java:comp/env/ejb/NodeBean`

*******Beliefs : NODE: A : User Profile
Engineer                    : 0.2545166613340778
Mathematician               : 0.1390229411706419
ComputerScientist           : 0.2545166613340778
SoftwareEngineer            : 0.251404721449477
BasketballPlayer            : 0.10053901471172558

*******Beliefs : NODE: B : Interest Category
GraphTheory                 : 0.16474500367924177
ProbStats                   : 0.18017135439926857
MachineLearning             : 0.14432472117870188
NeuralNetworks              : 0.2074730325691345
Algorithms                  : 0.18749982365984136
BasketBall                  : 0.11578606451381206

*******Beliefs : NODE: C : Hyperlink Paths
Path1                       : 3.1431596227164465E-4
Path2                       : 3.1431596227164465E-4
Path3                       : 3.1431596227164465E-4
Path4                       : 6.2863192454328 93E-4
Path5                       : 0.9984284201886418

*******Beliefs : NODE: D : Content Categories
EngAndScience               : 3.9934346024041953E-5
Mathematics                 : 3.9934346024041953E-5
GraphTheory                 : 3.9934346024041953E-5
ProbAndStats                : 3.9934346024041953E-5
ComputersAndInternet        : 4.7921215228850344E-5
AI                          : 1.5973738409616 78E-5
MachineLearning             : 1.597373840961678E-5
NN                          : 3.1947476819233 56E-5
Programming                 : 3.1947476819233 56E-5
SoftwareEngineering         : 3.1947476819233 56E-5
Algorithms                  : 6.3894953638467 13E-5
GeneralInterest             : 0.19992013130795191
SportAndAdv                 : 0.19992013130795191
BasketBall                  : 0.5997603939238558

*******Beliefs : NODE: E : Advertisements
Informative                 : 0.5951441098951817
NotInformative              : 0.4048558901048182

*******Beliefs : NODE: F : Page
BayesianNetworks            : 0.0
MJordan(Professor)          : 0.0
MJordan(Basketball)         : 1.0

*******Beliefs : NODE: G : Buy
True                        : 1.0
False                       : 0.0

*******Beliefs : NODE: H : Next Page
JudeaPearl                  : 0.0
MJordan(Professor)          : 0.0
MJordan(BasketBall)         : 1.0
OtherPage                   : 0.0
```

Figure 48: Beliefs of Nodes after Belief Propagation (Evidence and Learning)

Figure 49 illustrates the new beliefs after learning, in the presence of no evidence from the environment. Compare the new beliefs, with the original beliefs in Figure 43. The beliefs of nodes F, G and H have changed, to incorporate the new evidence from the environment.

```
Initiating login ...
Binding name: java:comp/env/ejb/NodeBean

*******Beliefs : NODE: A : User Profile
Engineer                    : 0.25
Mathematician               : 0.125
ComputerScientist           : 0.25
SoftwareEngineer            : 0.25
BasketballPlayer            : 0.125

*******Beliefs : NODE: B : Interest Category
GraphTheory                 : 0.1325
ProbStats                   : 0.14375
MachineLearning             : 0.12625
NeuralNetworks              : 0.21375000000000002
Algorithms                  : 0.21375000000000002
BasketBall                  : 0.16999999999999998

*******Beliefs : NODE: C : Hyperlink Paths
Path1                       : 0.1
Path2                       : 0.1
Path3                       : 0.1
Path4                       : 0.2
Path5                       : 0.5

*******Beliefs : NODE: D : Content Categories
EngAndScience               : 0.049999999999999999
Mathematics                 : 0.049999999999999999
GraphTheory                 : 0.049999999999999999
ProbAndStats                : 0.049999999999999999
ComputersAndInternet        : 0.06
AI                          : 0.02
MachineLearning             : 0.02
NN                          : 0.04
Programming                 : 0.04
SoftwareEngineering         : 0.04
Algorithms                  : 0.08
GeneralInterest             : 0.09999999999999998
SportAndAdv                 : 0.09999999999999998
BasketBall                  : 0.29999999999999993

*******Beliefs : NODE: E : Advertisements
Informative                 : 0.7
NotInformative              : 0.3

*******Beliefs : NODE: F : Page
BayesianNetworks            : 0.2378179211811432
MJordan(Professor)          : 0.278908577898186
MJordan(Basketball)         : 0.4832735009206708

*******Beliefs : NODE: G : Buy
True                        : 0.3410152476439632
False                       : 0.6589847523560368

*******Beliefs : NODE: H : Next Page
JudeaPearl                  : 0.19744258702397247
MJordan(Professor)          : 0.1683540389919771
MJordan(BasketBall)         : 4.5569745485988156E-4
OtherPage                   : 0.6337476765291905
```

Figure 49: New Beliefs after Learning

8.5 The Limitations of the BaBe Prototype

The BaBe prototype is limited in that the configuration of the Bayesian behaviour network is a manual process and the prototype does not implement full Bayesian learning. Learning is limited to Bayesian networks with known structure, and with no hidden variables. Due to the known structure, it is possible for the configuration and deployment process to be done manually. The JMS queues that represent the links in the Bayesian behaviour network are created manually using the j2eeadmin utility. The belief propagation agents are then deployed manually to listen on these links using the J2EE deployment tool.

8.6 Conclusion

In this chapter, we illustrated the operation of our prototype Bayesian agencies, and how the Bayesian behaviour network is configured.

We illustrated collective belief propagation in the absence and presence of evidence from the environment, and how the network can be queried. We showed the flow of LAMBDA and PI messages in the Bayesian behaviour network in response to environmental evidence.

We demonstrated learning and illustrated how new evidence is incrementally integrated into the conditional probability tables. This learning mechanism is limited in that it assumes known structure and no hidden variables or missing data. The configuration and deployment is a manual process, which we demonstrated.

With the prototype implementation of the BaBe agent architecture, we proved that it is possible to use a commercially available component architecture, namely Sun's Enterprise JavaBeans™ component architecture, to implement an adaptive agent architecture. This is possible because the BaBe agent architecture use simple agents as components, rather than attempting to implement complex agents.

Chapter 9
Conclusions and Future Research

9.1 Future Research: The BaBe Architecture

In this thesis, we described the use of Bayesian behaviour networks as internal models in complex adaptive systems. We described prototype EJB components that are assembled into distributed Bayesian behaviour networks, collectively performing Bayesian learning and Bayesian belief propagation in singly-connected Bayesian behaviour networks. As a detailed study of Bayesian belief propagation and learning falls outside the scope of this research, our prototype implementation includes learning in Bayesian networks with known structure and no hidden variables and belief propagation in singly-connected polytrees using Judea Pearl's belief propagation algorithm.

Future research will involve a full implementation of Bayesian learning, where Bayesian agents collectively and incrementally discover structure from data in the presence of known values for variables as well as in the presence of missing data. We will also complete the collective belief propagation capabilities of the Bayesian agencies in order to cope with multiply-connected Bayesian behaviour networks.

The major shortcomings of the current BaBe architecture are due to the "black-box" nature of the J2EE platform. In this platform, configuration between components and middleware services is only supported at deployment-time using a deployment descriptor. This configuration cannot be changed during runtime. BaBe needs to adapt to its environment in a dynamic fashion. This cannot be achieved if the BaBe architecture must be redeployed after any change to the competence components, Bayesian behaviour network configuration or the competence agencies, as enforced by the current J2EE platform. In our future work, we plan to deploy the BaBe architecture in a reflective component-based framework.

A reflective system can reason about and act upon itself. Such a system contains metadata representing some part of itself, object data representing its functional application or domain and a program to manipulate the object and metadata. The metadata can be inspected to describe some part of the system, and changed to adapt the system (Keeney & Cahill, 2003). Figure 50 illustrates our reflective BaBe agent architecture that we plan to implement in our future research.

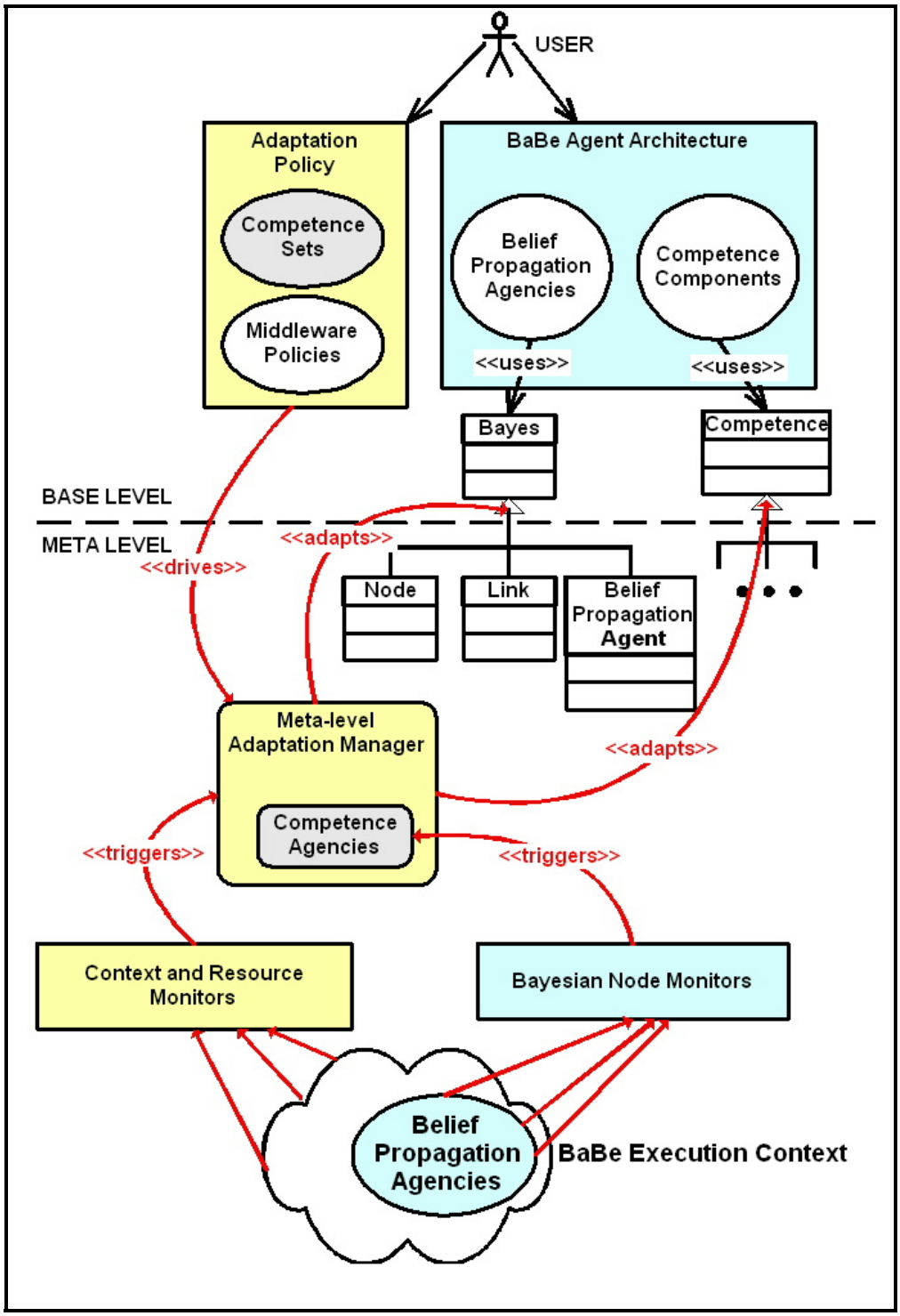

Figure 50: Reflective BaBe Architecture

This architecture is based on the Chisell architecture (Keeney & Cahill, 2003). The reflective BaBe architecture is divided into a base level and a meta level. The base level consists of the components, their interfaces and a policy specification. This policy specification contains adaptation rules for the system. These rules specify how to adapt the system in an application-aware manner but do not contain low-level execution environment information. The competence sets form an integral part of the policy document. In this architecture, the competence agencies are decoupled from the belief propagation agencies. The meta-level adaptation manager contains as a subset the competence agencies. The adaptation manager accesses low-level middleware services using the context and resource monitors, and the competence agencies test their constraints against the Bayesian network nodes using the Bayesian node monitors. The adaptation manager then evaluates the rules in the policy document and modifies middleware services, or activates, destroys or modifies components as specified by the policy rules. The user must be able to view/modify the adaptation policy in an interactive fashion.

Our future research will also involve the implementation of a commercially viable application using the BaBe reflective agent architecture. BaBe can be used to implement applications in any environment in which a system must dynamically adapt to an uncertain and complex environment. Example applications include customer relationships management, knowledge management and supply-chain optimisation. A particularly interesting application is the use of Bayesian agencies to sustain the competitive advantage of an enterprise.

9.2 The Complex Adaptive Enterprise: Sustaining the Competitive Advantage using Bayesian Agencies

9.2.1 Introduction

The challenge facing every modern enterprise is not only how to achieve its competitive advantage, but how to sustain this competitive advantage. In a world where the market, customer profiles and demands change constantly and the events in the global marketplace are unpredictable, it becomes increasingly difficult for an enterprise to sustain its competitive advantage. Under these conditions of uncertainty, complexity and constant change, it becomes very important for an enterprise to be able to learn from its experience and to adapt

its behaviour in order to constantly outperform its competitors. An enterprise that has these characteristics is a complex adaptive enterprise.

The interrelationships between the resources in a complex adaptive enterprise amongst themselves and within the marketplace are not only unlimited but mostly hidden. These interrelationships can therefore affect so many different resources throughout the enterprise that it is impossible for the human mind to comprehend. One of the main challenges of the modern enterprise is to understand this complex web of interrelationships and to integrate this understanding into its business processes and strategies in such a way that it can sustain its competitive advantage.

9.2.2 The Chain of Sustainability

According to the resource-based theory, there are dynamic relationships between enterprise resources, the capabilities of the enterprise and the competitive advantage of the enterprise (April, 2002). The complex adaptive enterprise maintains a *chain of sustainability* that constantly evolves from the interactions between the individual resources and the interactions between the resources and the dynamically changing marketplace in order to sustain a competitive advantage (April, 2002). April's chain of sustainability is illustrated in Figure 51. We summarize April's description of the chain of sustainability very briefly in the rest of this section.

Resources or assets are the basic components in the chain of sustainability. Example resources are products, employee skills, knowledge, and so forth. There are two types of resources, namely tangible and intangible resources. Examples of *tangible resources* include real estate, computer hardware and software. These resources are important to the strategy of the enterprise, but are usually not a source of competitive advantage as these resources can usually be acquired with ease by the competition. Examples of *intangible resources* include organizational cultures, technological knowledge, know-how shared among employees, patented processes and designs, trademarks, and accumulated learning and/or knowledge, as well as experience. Intangible resources often play a critical role in the competitive advantage of the enterprise, as it can rarely be separated from the skills and knowledge of the employees, and can therefore not be acquired by the competition.

The resources in an enterprise are combined into *complementary resource combinations* (CRCs) according to the functionality that these resources collectively achieve. The collective behaviour of the resources in a CRC produces greater value to the enterprise than the individual resource behaviours. The resources in a CRC therefore complement each

other. CRCs are the unique inter-relationships between tangible and intangible resources and are the source of competitive advantage in an enterprise. An enterprise can take the same tangible resources as its competitors, and uniquely combine these resources with intangible resources and produce better quality products and services more efficiently than its competitors. While the pool of assets or resources is the source of the set of CRCs in an enterprise, the CRCs are the main source of its competitive advantage.

The behaviours of the CRCs define the *strategic architecture* of an enterprise, which is defined as the capabilities of an enterprise, when applied in the marketplace. It is the customer's perspective of the capabilities of the enterprise. There are two types of capabilities, namely key capabilities and core capabilities. *Key capabilities* refer to capabilities that are merely necessary for the enterprise to be a player in their market or sector and ensure competitive parity. These include services to support internal customers (for example human resources, legal, accounting skills and processes) as well as those skills and systems that are pre-conditions for doing business in the industry in which the enterprise operates. These capabilities will increase the probability of economic survival, but are typically not the capabilities that account for the real competitive advantage of the enterprise. *Core capabilities*, on the other hand, refer to capabilities that are valuable and profit producing in the marketplace, and are those capabilities that an enterprise relies on for its competitive advantage. The list of core capabilities includes a set of abilities describing efficiency and effectiveness, for example faster, more responsive, more flexible, higher quality, and so forth. These capabilities apply to all the business processes in the enterprise. The CRCs in an enterprise serve as the basis for developing these key and core capabilities.

Social complexity refers to the complex behaviour exhibited by a complex adaptive enterprise, when its CRCs are embedded in a complex web of social interactions. These CRCs are referred to as socially complex resource combinations (SRCs). In social complexity, the source of competitive advantage is known, but the method of replicating the advantage is unclear. Examples include corporate culture, the interpersonal relations among managers or employees in an enterprise and trust between management and employees. Socially complex resource combinations (SRCs) depend upon large numbers of people or teams engaged in co-ordinated action such that few individuals, if any, have sufficient breadth of knowledge to grasp the overall phenomenon.

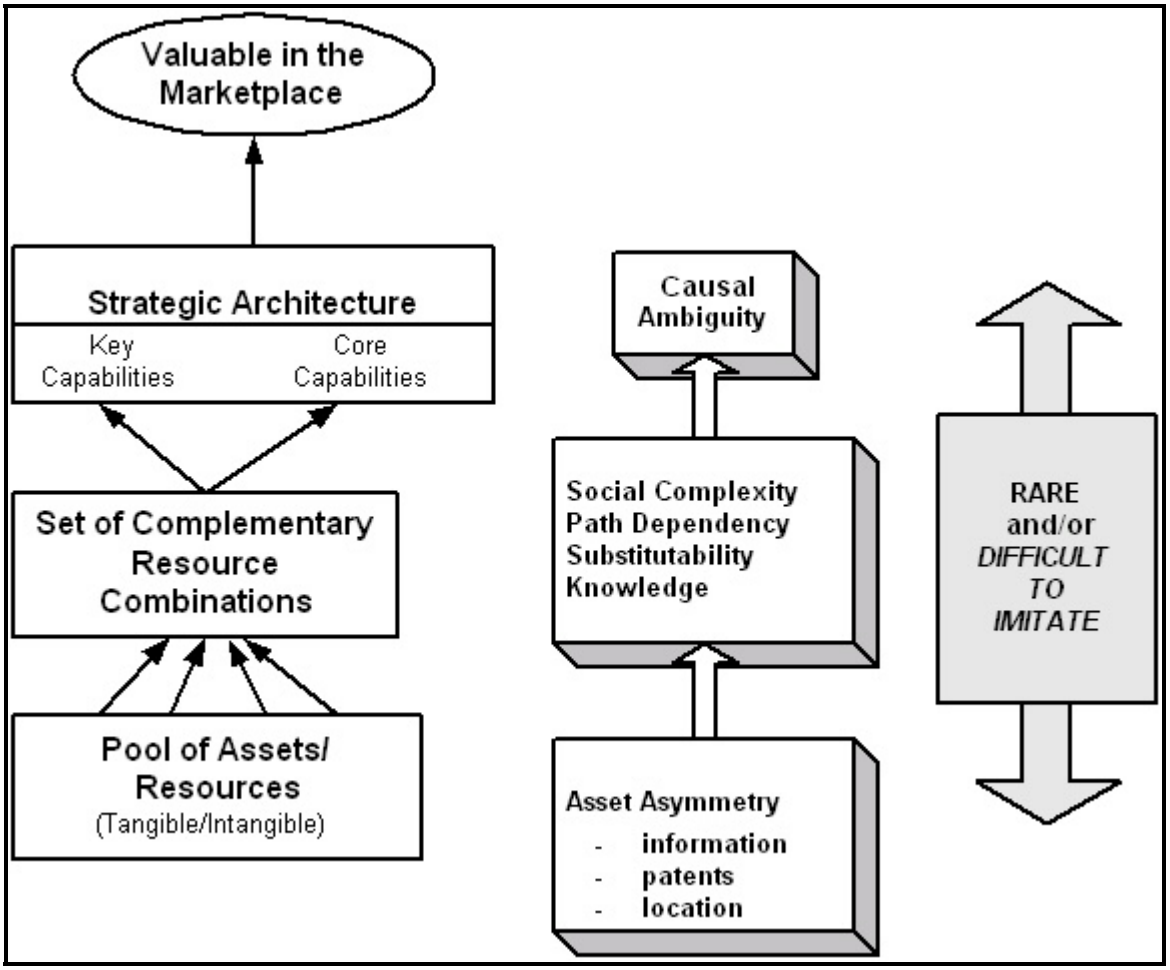

Figure 51: The Chain of Sustainability (April, 2002)

Casual ambiguity refers to uncertainty regarding the causes of efficiency and effectiveness of an enterprise, when it is unclear which resource combinations are enabling specific capabilities that are earning the profits. In order for an enterprise to sustain its competitive advantage, it must initially understand, for themselves, what these clusters of CRCs are that are driving and enabling their strategic capabilities to generate profit and, once understood, to build in more causally ambiguous knowledge components and then embed that understanding within the business processes and thinking within the enterprise.

In moving from basic assets or resources, through to CRCs, and eventually to key and core capabilities, the enterprise wants it all to be rare and difficult to imitate by competitors.

9.2.3 The Relationship between Emergence and Knowledge in the Complex Adaptive Enterprise

Emergence, the most important characteristic of a complex adaptive enterprise and the main contributor to the competitive advantage of the enterprise, is the collective behaviour of interacting resources in the CRCs. New knowledge is produced by the individual resources in the CRCs. The individual resources in a CRC collectively formulate new ideas, validate them, and this knowledge is then propagated across the enterprise, affecting the individual and collective behaviours of the resources within the CRCs.

According to April (2002), examples of this knowledge include:

- knowledge related to internal relationships within the company;
- knowledge related to products and services;
- knowledge related to business processes and business units;
- knowledge related to specific projects and project implementations;
- knowledge related to customers;
- knowledge related to the marketplace.

This knowledge can be *tacit* or *explicit*. According to April, tacit knowledge is usually defined as that which cannot be written down or specified.

Emergence therefore contributes to the creation of knowledge and causes social complexity and causal ambiguity, which are difficult to be imitated by competitors. This emergence must however be understood and engineered.

9.2.4 Future Research: BaBe and the Engineering of Emergence in the Complex Adaptive Enterprise

Self-awareness in a complex adaptive enterprise is instrumental in the maintenance of its chain of sustainability. Enterprises need to understand the interrelationships between the individual behaviours of the resources and the emergent behaviours of the CRCs and SRCs. This will enable the enterprise to understand its own social complexity and causal ambiguity.

In future research, we want to capture the knowledge in the complex adaptive enterprise using Bayesian behaviour networks. We want to package business processes and workflows into re-usable components, which will be selectively activated by the competence agencies depending on the collective belief propagation in the belief propagation agencies.

In order for the employees of an enterprise to understand its own knowledge, the Bayesian agencies must be used to adaptively present knowledge to employees in a way that is determined by the profile of the employee. As an example, knowledge presented to a manager must assist him/her to make strategic decisions, whereas knowledge presented to a technical person must improve his/her learning and skills. Bayesian agencies can also use knowledge about the marketplace for personalised marketing such as illustrated in the simplified web personalization example in Figure 28.

9.3 Conclusion

Cariani (1991) expresses the problem of the engineering of emergence as:

> *how can a designer build a device which outperforms the designer's specification?*

The BaBe agent architecture solves this problem by using Minsky's model of an A-Brain and a B-Brain. The belief propagation agencies can be viewed as A-Brains, connected to the environment and continuously inferring beliefs about and learning from the latest environmental states. The BaBe methodology initialises the structure of the Bayesian behaviour networks that form the basis of the A-Brains.

The competence agencies can be viewed as B-Brains, inspecting the beliefs of the Bayesian behaviour networks underlying the A-Brains and acting according to the beliefs. Collectively the belief propagation agencies and competence agencies learn from and adapt to their changing environmental states, therefore acting as a complex adaptive system.

We implemented Bayesian behaviour networks using a component-based approach. We defined an adaptive agent architecture, namely the BaBe agent architecture, and its implementation using re-usable components. In this architecture, the hyperstructures in the internal model are Bayesian behaviour networks. These networks are assembled from homogeneous node components, link components and belief propagation agents organised into belief propagation agencies. Competence components are domain-specific, heterogeneous components organised into competence agencies.

The belief propagation agencies and competence agencies are organised into a heterarchical structure through shared nodes in the underlying Bayesian behaviour network collectively implemented by these agencies. Each competence agency activates one or more competence component, depending on the collective inference by the belief propagation agencies.

We described a general application of the BaBe agent architecture. This general application is based upon the J2EE platform and can be deployed in any uncertain environment to understand and act upon the effects of emergence. We implemented a prototype, proving that it is possible to implement an adaptive agent architecture using a commercially available component architecture.

Emergence cannot be engineered by static software engineering processes, as it evolves only after implementation. We defined the BaBe methodology that extends current agent-oriented software engineering methodologies. The BaBe methodology uses similar analysis and design models as defined in the Gaia and coordination-oriented methodologies, but integrates an additional emergence model into the software engineering lifecycle. This model is a run-time software engineering model and it is also the internal model of the BaBe adaptive agent architecture, initialised by the software engineer during analysis and design. This model is then deployed and maintained by the Bayesian agencies. We described a general application that will enable the software engineer to maintain the emergence model. This application will enable the software engineer to visualize the dynamics of change and to adapt the behaviour of the BaBe agent architecture.

The J2EE component architecture does not support dynamic configuration of components and middleware services during runtime. For this reason, the J2EE platform will be unable to support our future research. A full implementation of the BaBe architecture will require a reflective component-based framework, which allows the dynamic adaptation of the configuration between components and middleware services, enabling BaBe to respond to its environment in a dynamic fashion.

Implementing the BaBe agent architecture using a reflective component-based framework, will be the first step towards achieving reflective, self-aware complex adaptive (software) systems. We plan to apply the reflective BaBe adaptive agent architecture and the BaBe methodology in order to engineer emergence in a complex adaptive enterprise. BaBe will assist such an enterprise to be self-aware so that it can use this self-understanding to adapt its business processes and strategies in response to the changing marketplace in order to sustain its competitive advantage.

Bibliography

April, K. (2002). Guidelines for developing a k-strategy. *Journal of Knowledge Management, 6(5)*, 445-456.

Baas, N. A. & Emmeche, C. (1997). On Emergence and Explanation. *Intellectica, 25*, 67-83. Retrieved March 22, 2001, http://www.nbi.dk/~emmeche/coPubl/97d.NABCE/ExplEmer.html

Bachman, F. Bass, L. Buhman, C. Comella-Dorda, S. Long, F. Robert, J. Seacord, R. & Wallnau, K. (2000, May). *Volume II: Technical Concepts of Component-Based Software Engineering* . Retrieved October 8, 2001, http://www.sei.cmu.edu/staff/rcs/CBSE-papers.html

Becker, A., Bar-Yehuda, R. & Geiger, D. (2000). Randomized Algorithms for the Loop Cutset Problem. *Journal of Artificial Intelligence Research, 12*, 219-234. Retrieved March 7, 2001, http://www.cs.washington.edu/research/jair/abstracts/becker00a.html

Becker, A. & Geiger, A. (1996). A sufficiently fast algorithm for finding close to optimal junction trees. *Proceedings of the Twelfth Conference on Artificial Intelligence,* 81-89. Retrieved March 8, 2001, http://www.cs.technion.ac.il/~dang/

Brooks, R. A. (1985). *A Robust Layered Control System for a Mobile Robot, MIT AI Memo 864.* Retrieved 18 July 2000, http://www.ai.mit.edu/people/brooks/papers/AIM-864.pdf

Brooks, R. A. (1990). *Elephants Don't Play Chess.* Retrieved 13 September 2002, http://www.ai.mit.edu/people/brooks/papers/elephants.pdf

Brooks, R. A. (1991). *Intelligence without Reason, MIT AI Memo 1293.* Retrieved July 18, 2000, http://www.ai.mit.edu/people/brooks/papers.html

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents, PhD Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.* Retrieved September 5, 2001, http://ftp.ai.mit.edu/pub/users/joanna/pdh.pdf

Cariani, P. (1991). *A Review of Emergence and Artificial Life.* Reviewed by R. Saunders, Retrieved September 17, 2001, http://www.arch.usyd.edu.au/~rob/study/EmergenceAndArtificialLife.html

Claus, C. & Boutilier, C. (1998). The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. *AAAI/IAAI*, 746-752. Retrieved May 29, 2000, http://citeseer.nj.nec.com/claus97dynamics.html

Dechter, R. (1996). Bucket Elimination: A Unifying Framework for Probabilistic Inference. *Uncertainty in Artificial Intelligence, UAI96*, 211-219. Retrieved October 8, 2000, http://www.ics.uci.edu/~dechter/publications/

Diez, F. J. (1996). Local Conditioning in Bayesian networks. *Artificial Intelligence, 87*, 1-20. Retrieved January 17, 2001, http://www.dia.uned.es/~fjdiez

Durfee, E. H. (2001). Scaling up Agent Coordination Strategies. *COMPUTER, IEEE Computer Society, 34(7),* 39-46.

Financial Systems Architects, MEGA International, Mercury ComputerSystems & TogetherSoft (2002, 3 June). OMG Unified Modeling Language Specification (revised submission). *Version 2.0.11, Revised InfraStructure Submission*, . Retrieved June 5, 2002, http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Infrastructure_RFP.html

Finin, T. Fritzson, R. McKay, D. & McEntire, R. (1994). KQML as an Agent Communication Language. *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM94) , ACP Press*, 456-463. Retrieved May 16, 2002, http://citeseer.nj.nec.com/finin94kqml.html

Foundation for Intelligent Physical Agents (2000, August). *FIPA ACL Message Structure Specification.* Retrieved May 16, 2002, http://www.fipa.org/specs/fipa00061/XC00061D.html

Gell-Mann, M. (1994). *The Quark and the Jaguar* (2nd ed.). London: Little, Brown and Company.

Gell-Mann, M. (1995). What is Complexity?. *Complexity, 1(1)*.

Griss, M. L. & Pour, G. (2001). Accelerating Development with Agent Components. *COMPUTER, IEEE Computer Society, 34(5),* 37-43.

Heylighen, F., Joslyn, C. & Turchin, V. (2001). *Principia Cybernetica Web.* Retrieved March 18, 2002, http://pcp.lanl.gov/

Holland, J. H. (1995). *Hidden Order: How Adaptation Builds Complexity*. Massachusetts :Addison-Wesley Publishing Company Inc.

Hopkins, J. (2000). Component Primer. *Communications of the ACM, 43*(10), 27-30.

Hume, D. (1888). *A Treatise of Human Nature* (17th ed.). L.A. Selby-Bigge(Eds.). London: Oxford University Press.

Jennings, N. R. (1993). Specification and Implementation of a Belief-Desire-Joint-Intention Architecture for Collaborative Problem Solving. *International Journal on International and Cooperative Information Systems, 2(3)*, 289-318. Retrieved May 9, 2002, http://citeseer.nj.nec.com/jennings93specification.html

Jennings, N. R. (2001). An Agent-based Approach for Building Complex Software Systems. *Communications of the ACM, 44(4),* 35-41.

Jennings, N. R. & Wooldridge, M. (2000). Agent-oriented software engineering. *Handbook of Agent Technology*. Retrieved January 23, 2001, http://www.elec.qmw.ac.uk/dai/pubs/ - 2000

Jennings, N. R., Sycara, S. & Wooldridge, M. (1998). A Roadmap of Agent Research and Development. *International Journal on Autonomous Agents and Multi-Agent Systems, 1(1)*, 7-38. Retrieved July 11, 2000, http://www.ecs.soton.ac.uk/~nrj/pubs.html - 1998

Jensen, F., Jensen, F. V. & Dittmer, S. L. (1994). *From Influence Diagrams to Junction Trees, Proceedings of the Tenth Conference of Uncertainty in Artificial Intelligence.* Retrieved February 13, 2001, http://www.cs.auc.dk/research/DSS/abstracts/jensen:jensen:dittmer:94.html

Kim, J. H. & Pearl, J. (1983). A Computational Model for Causal and Diagnostic Reasoning in Inference Systems. In A. Bundy (Ed.), *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (IJCAI-83), Karlsruhe, West Germany, (pp. 190-193).

Keeney, J. & Cahill, V. (2003). *Chisell: A Policy-Driven, Context-Aware, Dynamic Adaption Framework*. Retrieved February 22, 2004, http://citeseer.nj.nec.com/keeney03chisel.html

Kon, F., Costa, F., Blair, G. & Campbell, R. H. (2002). The Case for Reflective Middleware. *Communications of the ACM, 45*(6), 33-38.

Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C. & Campbell, R. H. (2000). Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*. New York. April 3-7, 2000. Retrieved February 22, 2004, http://choices.cs.uiuc.edu/2k/dynamicTAO/

Maes, P. (1989). How to do the Right Thing. *MIT AI Memo 1180.* Retrieved July 18, 2000, http://agents.www.media.mit.edu/groups/agents/publications/

Maes, P. (1990). A bottom-up mechanism for behavior selection in an artificial creature. *From animals to animats: International Conference on Simulation of Adaptive Behavior* (pp. 239-246). Cambridge, Mass: MIT Press.

Maes, P. (1994). *Modelling Adaptive Autonomous Agents.* Retrieved June 21, 2000, http://agents.www.media.mit.edu/groups/agents/publications/

Maes, P. & Brooks, R. A. (1990). Learning to Coordinate Behaviours. *National Conference on Artificial Intelligence,* 796-802. Retrieved May 4, 2002, http://citeseer.nj.nec.com/maes90learning.html

McGuren, Fl. & Conroy, D. (2002). X-Adapt: An Architecture for Dynamic Systems. *Seventh International Workshop on Component-Oriented Programming (WCOP 2002)*. Retrieved February 22, 2004, http://research.microsoft.com/~cszypers/events/WCOP2002/ -_Accepted_Position_Papers

Minsky, M. (1988). *The Society of Mind* (First Touchstone ed.). New York: Simon & Schuster.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis* (1st ed.). San Francisco, USA: Morgan Kaufmann.

Odell, J. (1998). Agents and Beyond: A Flock is not a Bird, *Distributed Computing*. Retrieved September 6, 2002, http://www.jeffsutherland.org/oopsla98/boca_cas.html

Odell, J., Van Dyke Parunak, H. & Bauer, B. (2001). *Extending UML for Agents, AOIS Workshop at AAAI2000.* Retrieved September 4, 2001, http://aot.ce.unipr.it/auml/working

Object Management Group (2000). Agent Technology Green Paper. *OMG Document ec/2000-08-01, Version 1.0*. Retrieved April 22, 2002, http://www.jamesodell.com/ec2000-08-01.pdf

Pearl, J. (1982). Reverend Bayes on Inference Engines: A distributed Hierarchical Approach. *Procceedings of the National Conference on Artificial Intelligence,* Pittsburgh, American Association for Artificial Intelligence, 133-136.

Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (2nd ed.). San Mateo, USA: Morgan Kaufmann Publishers.

Pearl, J. & Russell, S. (2000). *Bayesian networks, Technical Report R-277, UCLA Cognitive Systems Laboratory.* Retrieved May 5, 2001, http://bayes.cs.ucla.edu/csl_papers.html

Popescul, A. Ungar, L. H., Pennock, D. M., & Lawrence, S. (2001). *Probabilistic Models for Unified Collaborative and Content-Based Recommendation in Sparse-Data Environments.* Retrieved January 28, 2002, http://www.cis.upenn.edu/~popescul/publications.html

Potgieter, A. & Bishop, J. (2001). Bayesian agencies on the Internet. *Proceedings of the 2001 International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC '2001).*

Rao, A. S. & Georgeff, M. P. (1995). BDI Agents: From Theory to Practice. *Proceedings of the First International Conference on Multiagent Systems, Technical Note 56*. Retrieved April 21, 2002, http://citeseer.nj.nec.com/rao95bdi.html

Ronald, E. M. A. & Sipper, M. (2000). Engineering, emergent engineering, and artificial life: Unsurprise, unsurprising surprise, and surprising surprise. In M. A. Bedau, J. S. McCaskill, N. H. Packard, & S. Rasmussen (Eds.), *Artificial Life VII: Proceedings of the Seventh International Conference* (pp. 523-528). Cambridge, Massachusetts: The MIT Press.

Ronald, E. M. A., Sipper, M. & Capcarrĕre, M. S. (1999). Design, observation, surprise! A test of emergence. *Artificial Life, 5(3)*, pp. 225-239.

Rosenblatt, J. K. & Payton, D. W. (1989). A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control, *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, IEEE.* Retrieved September 11, 2002, (http://citeseer.nj.nec.com/cache/papers/cs/1141/http:zSzzSzwww.umiacs.umd.eduzSzuserszSzjuliozSzpaperszSzFine_Grained_Alternative.pdf/rosenblatt89finegrained.pdf

Rumbaugh, J. Blaha, M. Premerlani, W. Eddy, F. & Lorensen, W. (1991). *Object-Oriented modelling and design*. Prentice-Hall.

Russell, S. J., Binder, J. Koller, D. & Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence,* 1146-1152. Retrieved September 19, 2000, http://robotics/stanford/edu/~koller/papers/apnijcai.html

Russell, S. J. & Norvig, P. (2003). *Artificial Intelligence A Modern Approach* (2nd ed.). New Jersey, USA: Prentice Hall.

Sight Code Inc. (2001). Sight Code Initial Submission against the UML Infrastructure RFP. *UML 2.0 Infrastructure Proposal, OMG document number ad/2001-08-23*. Retrieved October 3,2001, http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Infrastructure_RFP.html

Skarmeas, N. & Clark, K. (1999). *Component Based Agent Construction.* Retrieved June 6, 2000, http://citeseer.nj.nec.com/skarmeas99component.html

Sommerville, I. (1995). *Software Engineering* (5th ed.). England: Addison-Wesley Publishing Company.

UML Revision Task Force (2001, September). OMG Unified Modelling Language Specification. *OMG Document formal/01-09-67*. Retrieved April 22, 2002, http://cgi.omg.org/cgi-bin/doc?formal/01-09-67

Wooldridge, M. (1997). Agent-based Software Engineering. IEE Proceedings of Software Engineering, 144(1), 26-37. Retrieved January 26, 2001, http://www.csc.liv.ac.uk/~mjw/pubs/

Wooldridge, M. & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review, 10(2)*, 115-152. Retrieved April 19, 2002, http://citeseer.nj.nec.com/article/wooldridge95intelligent.html

Wooldridge, M., Jennings, N. R. & Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems, 3(3)*, 15. Retrieved December 17, 2000, http://www.ecs.soton.ac.uk/~nrj/pubs.html - 1998

Zambonelli, F., Jennings, N. R., Omicini, A. & Wooldridge, M. (2000). *Agent-oriented software engineering for Internet Applications, Coordination of Internet Agents.* Retrieved January 23, 2001, http://www.csc.liv.ac.uk/~mjw/pubs/

APPENDIX A
SOFTWARE LISTINGS

1 Node Component

1.1 NodeEJB

```
////////////////////////////////////////////////////////////////////////////
/**
 * Name            NodeEJB
 * Description : The node component administers the conditional probability table,
 *               PI and LAMBDA of a Bayesian Network node
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////
// Imports
import javax.ejb.*;
import javax.naming.*;
import java.sql.*;
import javax.sql.*;
import java.util.*;

////////////////////////////////////////////////////////////////////
/**
 * Implementation class of the NodeEJB entity bean.
 * Author Anet Potgieter (anet@bayesbean.com)
 * Version 1.0
 *
 * EJB
 *   Type:       Entity
 *   Description: NodeEJB Bean
 *   Home :      NodeHome
 *   Remote:     Node
 *   Persistence: Bean-managed
 *
 */

public class NodeEJB implements EntityBean {

    private String node;
    private String nodeName;
    private EntityContext context;
    private Connection con;
    private String dbName = "java:comp/env/jdbc/BabeDB";
    private int nrParentStates = 0;
    private int nrStates = 0;
    private Matrix cpm;
    private Matrix occurrences;
    private Array lambda = new Array(0);
    private Array piX = new Array(0);
```

```
  private Array evidence = new Array(0);
  private String [] incomingLinks = new String[0];
  private String [] outgoingLinks = new String[0];
  private String [] stateNames = new String [0];
  private int nrparents;
  private int nrchildren;

  private boolean trace = false;
  private boolean trace2 = false;

  int test = 1;

////////////////////////////////////////////////////////////////////////
// Business Routines
// -----------------

/**********************************************************************
 *  Name : reset
 *  Description : Reset pi, lambda and external pi for this node
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

  public void reset() {

  if (trace) {System.out.println("\n Resetting PI and Lambda for Node" + node);}

  for (int cnt = 0; cnt < piX.length();cnt++){
   piX.set(cnt,1.0) ;}

  for (int cnt = 0; cnt < lambda.length();cnt++){
   lambda.set(cnt,1.0) ;}

  for (int cnt = 0; cnt < evidence.length();cnt++){
   evidence.set(cnt,1.0) ;}

  }

/**********************************************************************
 *  Name : setNoEvidence
 *  Description : Set the evidence of the node to "no evidence"
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void setNoEvidence() {

  if (trace) {System.out.println("\n Setting node to No Evidence" + node);}

  for (int cnt = 0; cnt < evidence.length();cnt++){
    evidence.set(cnt,1.0) ;}
  }
```

```
/**********************************************************************
 *  Name : getNrStates
 *  Description : Get the number of states for this node
 *  Input parameters : none
 *  Return : nr of states
 **********************************************************************/

public int getNrStates() {
  return nrStates;
}

  /**********************************************************************
 *  Name : getPI
 *  Description : Get the PI of this node
 *  Input parameters : none
 *  Return : Array
 **********************************************************************/

  public Array getPI() {

 if (trace) {System.out.println("\nNode" + node + " getPI PI :");

 for (int cnt = 0; cnt < piX.length();cnt++){
    System.out.println(piX.get(cnt)+ ",");}

 System.out.println("\nEnd Node" + node + " PI");}
     return piX;
  }

/**********************************************************************
 *  Name : setPI
 *  Description : Set the PI of this node
 *  Input parameters : new PI
 *  Return : none
 **********************************************************************/


public void setPI(Array pi) {

 if (pi.length() <= this.piX.length()) {
   for (int cnt = 0; cnt < pi.length();cnt++){
    this.piX.set(cnt,pi.get(cnt));
   }

   if (pi.length() != this.piX.length()) {
      System.out.println("\n NodeEJB : Invalid length for setting PI for Node" + node);
   }
 }
 if (trace) {System.out.println("\nNode" + node + " setPI PI :");

 for (int cnt = 0; cnt < pi.length();cnt++){
   System.out.println(pi.get(cnt)+ ",");}
   System.out.println("\nNode" + node + " END setPI PI :");
```

```
  }
}

/**********************************************************************
 *  Name : getEvidence
 *  Description : Get the evidence of this node
 *  Input parameters : none
 *  Return : Array
 **********************************************************************/


public Array getEvidence() {
  if (trace) {System.out.println("\nNode" + node + " getEvidence Evidence :");

  for (int cnt = 0; cnt < evidence.length();cnt++){
    System.out.println(evidence.get(cnt)+ ",");}
    System.out.println("\nEnd Node" + node + " Evidence");
  }
  return evidence;
}

/**********************************************************************
 *  Name : setEvidence
 *  Description : Set the evidence of this node
 *  Input parameters : new Evidence
 *  Return : none
 **********************************************************************/

public void setEvidence(int parentState,int state,boolean learn) {



  this.evidence = new Array(nrStates,0.0);
  this.evidence.set(state - 1,1.0);

  if (learn){
    if (trace2) {System.out.println("\nNode" + node + " setEvidence Evidence :");

      for (int cnt = 0; cnt < nrStates;cnt++){
        System.out.println("\nCount:" + cnt + "EV : " + this.evidence.get(cnt)+ ",");
        System.out.println("\nCOUNT:" + cnt + "PI : " + cpm.get(parentState - 1,cnt)+ ",");
        System.out.println("\nCOUNT:" + cnt + "Occurences : " + occurrences.get(parentState - 1,cnt));
      }
      System.out.println("Node" + node + " END setEvidence:");
    }

    Array newProbs = new Array(nrStates,1.0);
    double occ = this.occurrences.get(parentState - 1,state - 1) + 1.0;
    this.occurrences.set(parentState - 1,state - 1,occ);

    for (int cnt = 0; cnt < newProbs.length(); cnt++){
      newProbs.set(cnt, this.occurrences.get(parentState - 1,cnt));
    }
```

```
    newProbs = newProbs.normalize();
    for (int cnt = 0; cnt < newProbs.length(); cnt++){
         this.cpm.set(parentState - 1,cnt,newProbs.get(cnt));
    }
  }

  if (trace2) {System.out.println("\nNode" + node + " setEvidence Evidence :");

    for (int cnt = 0; cnt < this.evidence.length();cnt++){
      System.out.println(this.evidence.get(cnt)+ ",");
    }
    System.out.println("\nNode" + node + " END setEvidence:");
  }
}

/**********************************************************************
 *  Name : setLAMBDA
 *  Description : Set the LAMBDA of this node
 *  Input parameters : new lambda
 *  Return : none
 **********************************************************************/

public void setLAMBDA(Array lambda) {
  this.lambda = lambda;
}

/**********************************************************************
 *  Name : setCPM
 *  Description : Set the CPM of this node
 *  Input parameters : new CPM
 *  Return : none
 **********************************************************************/

public void setCPM(Matrix CPM) {
  this.cpm = CPM;
}

/**********************************************************************
 *  Name : getCPM
 *  Description : Get the CPM of this node
 *  Input parameters : new lambda
 *  Return : none
 **********************************************************************/

  public Matrix getCPM() {
    return this.cpm;
}

/**********************************************************************
 *  Name : getLAMBDA
 *  Description : Get the LAMBDA of this node
 *  Input parameters : none
 *  Return : Array
```

```
 ***********************************************************************/

public Array getLAMBDA() {
  Array prod = this.lambda.termProduct(evidence);
  return prod;
}

/***********************************************************************
 *  Name : getBelief
 *  Description : Get the belief of this node
 *  Input parameters : none
 *  Return : Array
 ***********************************************************************/

public Array getBelief() {
  Array prod = piX.termProduct(lambda);
  prod = prod.termProduct(evidence);
  Array normProd = prod.normalize();
  return normProd;
}

/***********************************************************************
 *  Name : getStates
 *  Description : Get the state names of this node
 *  Input parameters : none
 *  Return : List of strings
 ***********************************************************************/

public String [] getStates () {
  return this.stateNames;
}

/***********************************************************************
 *  Name : getNode
 *  Description : Get the node name of this node
 *  Input parameters : none
 *  Return : String
 ***********************************************************************/

  public String getNode () {
    return this.nodeName;
}
/***********************************************************************
 *  Name : getOutgoingLinks
 *  Description : Get the outgoing links of this node
 *  Input parameters : none
 *  Return : List of strings
 ***********************************************************************/

  public String [] getOutgoingLinks () {
    return outgoingLinks;
}
```

```
/*********************************************************************
 *  Name : getIncomingLinks
 *  Description : Get the incoming links of this node
 *  Input parameters : none
 *  Return : List of strings
 *********************************************************************/

public String [] getIncomingLinks () {
  return incomingLinks;
}

/*********************************************************************
 *  Name : multiplyTransposeCPM
 *  Description : Multiply the transpose of the CPM with an array
 *  Input parameters : input Array
 *  Return : Array
 *********************************************************************/

public Array multiplyTransposeCPM(Array inputArray) {
  Array arr = new Array(nrStates,1.0);
  Matrix transcpm = cpm.transpose();
  arr = transcpm.matMult(inputArray);
  return arr;
}

/*********************************************************************
 *  Name : multiplyCPM
 *  Description : Multiply the CPM with an array
 *  Input parameters : input Array
 *  Return : Array
 *********************************************************************/

public Array multiplyCPM(Array inputArray) {
  Array arr = new Array(nrStates,1.0);
  arr = cpm.matMult(inputArray);
  return arr;
}

//////////////////////////////////////////////////////////////////////
// The following are required EJB methods, called by the container

/*********************************************************************************
 *  Name : ejbCreate
 *  Description : ejbCreate is called by the EJB container after setEntityContext
 *  Input parameters : Bayesian Network Node Name
 *  Return : String
 *********************************************************************************/

public String ejbCreate(String node)
  throws CreateException {

  try {
    if (trace) {System.out.println("\nNode" + node + " ejbCreate");}
```

```
    this.node = node;
    loadCPMDimensions(node);
    lambda = new Array(nrStates,1.0);
    piX = new Array(nrStates,1.0);
    evidence = new Array(nrStates,1.0);
    stateNames = getStateNames(nrStates);
    nodeName = getNodeName();
    loadCPMByNode(node);

    loadPL("node_pi", "pi", piX);
    loadPL("node_lambda", "lambda", lambda);
    loadPL("node_evidence", "evidence", evidence);

    if (trace) {System.out.println("\nNode (EJBCreate) " + node + " PI :");

    for (int cnt = 0; cnt < piX.length();cnt++){
      System.out.println(piX.get(cnt)+ ",");}

    System.out.println("\nEnd Node" + node + " PI");}

    outgoingLinks = getLinks("parent");
    incomingLinks = getLinks("child");
  } catch (Exception ex) {
    throw new EJBException("ejbCreate: " +
    ex.getMessage());
  }

  if (trace) {System.out.println("\nEnd Node" + node + " ejbCreate");}
  return node;
}

/*********************************************************************
 *  Name : ejbFindByPrimaryKey
 *  Description : Find the NodeEJB instance that matches the given primary key.
 *  Input parameters : Bayesian Network Node Name
 *  Return : String
 *********************************************************************/


public String ejbFindByPrimaryKey(String primaryKey) throws FinderException {

  boolean result;
  try {
     result = selectByPrimaryKey(primaryKey);
  } catch (Exception ex) {
    throw new EJBException("ejbFindByPrimaryKey: " +
    ex.getMessage());
  }
  if (result) {
    return primaryKey;
  }
  else {
```

```
    throw new ObjectNotFoundException("Row for id " + primaryKey + " not found.");
  }
}

/**********************************************************************
 *  Name : ejbRemove
 *  Description : Called by container before data removed from database.
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void ejbRemove() {
  if (trace) {System.out.println("\nNode" + node + " ejbRemove");}
  try {
  }
  catch (Exception ex) {
    throw new EJBException("ejbRemove: " +
    ex.getMessage());
   }
  if (trace) {System.out.println("\nEnd Node" + node + " ejbRemove");}
}

/**********************************************************************
 *  Name : ejbLoad
 *  Description : Called by container to refresh entity Bean's state.
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void ejbLoad() {
  if (trace) {System.out.println("\nNode" + node + " ejbLoad");}
  try {
    //loadCPMDimensions(node);
    loadCPMByNode(node);

    loadPL("node_pi", "pi", piX);
    loadPL("node_lambda", "lambda", lambda);
    loadPL("node_evidence", "evidence", evidence);

    if (trace) {System.out.println("\nNode" + node + " ejbLoad PI :");

    for (int cnt = 0; cnt < piX.length();cnt++){
      System.out.println(piX.get(cnt)+ ",");}

    System.out.println("\nEnd Node" + node + " ejbloadPI");}

  } catch (Exception ex) {
    throw new EJBException("ejbLoad: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Node" + node + " ejbLoad");}
}
```

```
/*********************************************************************
 *  Name : ejbStore
 *  Description : save Bean's state to database.
 *  Input parameters : none
 *  Return : none
 *********************************************************************/

public void ejbStore() {
  if (trace) {System.out.println("\nNode" + node + " ejbStore");}
  try {
    storePL("node_pi", "pi", piX);
    storePL("node_lambda", "lambda", lambda);
    storePL("node_evidence", "evidence", evidence);
    updateCPMByNode();
  } catch (Exception ex) {
    throw new EJBException("ejbLoad: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Node" + node + " ejbStore");}
 }

 /*********************************************************************
 *  Name : ejbPostCreate
 *  Description : Called by container after ejbCreate.
 *  Input parameters : none
 *  Return : none
 *********************************************************************/
 public void ejbPostCreate(String node) {}

 ////////////////////////////////////////////////////////////////////
 // The following are callback methods, called by the container to
 // notify the Bean that some event is about to occur.

 /*********************************************************************
 *  Name : setEntityContext
 *  Description : Called by container to set Bean context.
 *  Input parameters : context
 *  Return : none
 *********************************************************************/

 public void setEntityContext(EntityContext context) {
   if (trace) {System.out.println("\nNode" + node + " setEntityContext");}
   this.context = context;
   try {
     makeConnection();
   } catch (Exception ex) {
     throw new EJBException("Unable to connect to database. " +
     ex.getMessage());
   }
   if (trace) {System.out.println("\nend Node" + node + " setEntityContext");}
 }

/*********************************************************************
```

```
 *  Name : unsetEntityContext
 *  Description : Called by container to unset Bean context.
 *  Input parameters : context
 *  Return : none
 ********************************************************************/

 public void unsetEntityContext() {
  if (trace) {System.out.println("\nNode" + node + " unsetEntityContext");}
  try {
     con.close();
  } catch (SQLException ex) {
    throw new EJBException("unsetEntityContext: " + ex.getMessage());
  }
  if (trace) {System.out.println("\nend Node" + node + " unsetEntityContext");}
 }

 /********************************************************************
 *  Name : ejbActivate
 *  Description : Called by container before Bean swapped into memory.
 *  Input parameters : none
 *  Return : none
 ********************************************************************/

 public void ejbActivate() {
  if (trace) {System.out.println("\nNode" + node + " ejbActivate");}
  node = (String)context.getPrimaryKey();
  if (trace) {System.out.println("\nend Node" + node + " ejbActivate");}
 }

 /********************************************************************
 *  Name : ejbPassivate
 *  Description : Called by container before Bean swapped into storage.
 *  Input parameters : none
 *  Return : none
 ********************************************************************/

 public void ejbPassivate() {
  if (trace) {System.out.println("\nNode" + node + " ejbPassivate");}
  node = null;
  if (trace) {System.out.println("\nend Node" + node + " ejbPassivate");}
 }


////////////////////////////////////////////////////////////////////////////
/********************** Database Routines ************************/

 /********************************************************************
 *  Name : makeConnection
 *  Description : Connect to the database
 *  Input parameters : none
 *  Return : none
 ********************************************************************/
```

```
 private void makeConnection() throws NamingException, SQLException {
  if (trace) {System.out.println("\nNode" + node + " makeConnection");}
  InitialContext ic = new InitialContext();
  DataSource ds = (DataSource) ic.lookup(dbName);
  con =  ds.getConnection();
  if (trace) {System.out.println("\nend Node" + node + " makeConnection");}
 }

 /**********************************************************************
  *  Name : loadCPMDimensions
  *  Description : Determine dimensions of CPM
  *  Input parameters : Node Name
  *  Return : none
  **********************************************************************/

 private void loadCPMDimensions(String primaryKey) throws SQLException {

  // Determine nr states for this node
  String selectStatement = "select count(distinct state) " + "from cpm where node = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, primaryKey);

  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()) {
    this.nrStates = rs.getInt(1);
    if (trace) {System.out.println("\nNodeBean " + node + " Nr of States = " + nrStates + "\n");}
    prepStmt.close();
  }
  else {
    prepStmt.close();
    throw new NoSuchEntityException("Could not determine nr of states for node " + primaryKey +
              " in database.");
  }
  // Determine Matrix indices - total nr of states for this node
  selectStatement = "select count(state) " + "from cpm where node = ? ";
  prepStmt = con.prepareStatement(selectStatement);
  prepStmt.setString(1, primaryKey);
  rs = prepStmt.executeQuery();

  if (rs.next()) {
    int totalNrStates = rs.getInt(1);
    this.nrParentStates = totalNrStates / nrStates;
    if (trace) {System.out.println("\nNodeBean " + node + "nrParentStates = " + nrParentStates +
"\n");}
    prepStmt.close();
  }
  else {
    prepStmt.close();
    throw new NoSuchEntityException("Could not determine total nr of states for node " + primaryKey
+
              " in database.");
```

```
  }

 }

 /*********************************************************************
  *  Name : loadCPMByNode
  *  Description : Retrieve CPM for this node
  *  Input parameters : Node Name
  *  Return : none
  *********************************************************************/

 private void loadCPMByNode(String primaryKey) throws SQLException {
  if (trace) {System.out.println("\nNode " + node + " loadCPMByNode");}
  String selectStatement = "select probability, state, parent_state, occurences " +
  "from cpm where node = ? order by parent_state,state";
  PreparedStatement prepStmt = con.prepareStatement(selectStatement);

  prepStmt.setString(1, primaryKey);
  ResultSet rs = prepStmt.executeQuery();

  double [][] vals = new double [nrParentStates][nrStates];
  double [][] history = new double [nrParentStates][nrStates];

  for (int i=0; i < nrParentStates; i++){
    for (int j = 0; j < nrStates; j++) {
      rs.next();
      double prob = rs.getDouble(1);

      if (trace) {System.out.println("\nload Node CPM" + node + " Prob = " + "State," +
      rs.getString(2) + "Parent_State," + rs.getString(3) + ", PROBABILITY = " + rs.getDouble(1)
+"\n");}
      vals[i][j] = prob;
      int nr = rs.getInt(4);
      history[i][j] = nr;
    }
  }

  prepStmt.close();
  cpm = new Matrix(vals);
  occurrences = new Matrix(history);

  if (trace) {System.out.println("\nLoad CPM Node" + node + " loadCPM PI :");

  for (int cnt = 0; cnt < piX.length();cnt++){
    System.out.println(piX.get(cnt)+ ",");}

  System.out.println("\nEnd Node" + node + " loadCPM PI ");}

  if (trace) {System.out.println("\nend Node loadCPMByNode");}
 }

 /*********************************************************************
  *  Name : updateCPMByNode
```

```
 *  Description : Update CPM with updatet probabilities
 *  Input parameters : New Probabilities
 *  Return : none
 ************************************************************************/

private void updateCPMByNode() throws SQLException {
  if (trace) {System.out.println("\nNode " + node + " updateCPMByNode");}
  String updateStatement = "update cpm set probability = ?,occurences = ? where " +
                           "node = ? and state = ? and parent_state = ?";

  if (trace) {System.out.println(updateStatement);}
  int rowCount;
  PreparedStatement prepStmt =
  con.prepareStatement(updateStatement);


  prepStmt.setString(3, this.node);
  if (trace) {System.out.println("\nAfter setstring Node = " + this.node);}

  for (int i=0; i < nrParentStates; i++){
    for (int j = 0; j < nrStates; j++) {

    if (trace) {System.out.println("\n Updating Node CPM " + node + " Parent State = " + (i + 1) +
    "," + " State = " + (j + 1) + ", PROBABILITY = " + cpm.get(i,j) +"\n");}
      prepStmt.setInt(4, j+1);
      if (trace) {System.out.println("\nAfter setInt state = " + j);}

      if (nrParentStates == 1) { // A Root
        prepStmt.setInt(5, i);
        if (trace) {System.out.println("\nAfter setInt state = " + i);}
      }
      else
      {
        prepStmt.setInt(5, i + 1);
        if (trace) {System.out.println("\nAfter setInt state = " + (i + 1));}
      }

      prepStmt.setDouble(1, cpm.get(i,j));
      prepStmt.setInt(2, (int)occurrences.get(i,j));
      if (trace) {System.out.println("\nold CPM [" + i + "][" + j + "] = " + cpm.get(i,j));}
      rowCount = prepStmt.executeUpdate();

      if (trace) {System.out.println("\n Node updated " + rowCount);}

    }
  }

  prepStmt.close();

  if (trace) {System.out.println("\nend Node updateCPMByNode");}
}

/**********************************************************************
```

```
 *  Name : getNodeName
 *  Description : Retrieve Node Name for Node
 *  Input parameters : none
 *  Return : String
 ************************************************************************/

private String getNodeName() throws SQLException {

  String selectStatement = "select description from nodes where node = ?";

  if (trace2){System.out.println("\ngetStateNames" + selectStatement);}

  PreparedStatement prepStmt = con.prepareStatement(selectStatement);

  prepStmt.setString(1, node);
  ResultSet rs = prepStmt.executeQuery();
  rs.next();
  String nodeName = rs.getString(1);
  prepStmt.close();

  return(nodeName);
}

/**********************************************************************
   *  Name : getStateNames
   *  Description : Retrieve State Names for Node
   *  Input parameters : none
   *  Return : State Names
   ************************************************************************/

  private String[] getStateNames(int stateCnt) throws SQLException {

    String [] states = new String [stateCnt];
    String selectStatement = "select state,description from states where node = ? order by state";

    if (trace){System.out.println("\ngetStateNames" + selectStatement);}

    PreparedStatement prepStmt = con.prepareStatement(selectStatement);

    prepStmt.setString(1, node);
    ResultSet rs = prepStmt.executeQuery();

    for (int i=0; i < stateCnt; i++){
      rs.next();
      String state = rs.getString(2);

      if (i < stateCnt) {
        if (trace) {System.out.println("\nNode " + node + "State " + i + " = " + state);}
        states[i] = state; }
      else {
      throw new NoSuchEntityException("GetStates ARRAY BOUND ERROR !!!!"); }
    }
```

```
    prepStmt.close();

    return(states);
  }

  /*********************************************************************
   *  Name : getLinks
   *  Description : Retrieve Incoming Links
   *  Input parameters : relation = "parent"/"child"
   *  Return : none
   *********************************************************************/

   private String[] getLinks(String relation) throws SQLException {

    int nr = 0;
    String [] links = new String [0];

    // Determine nr incoming links for this node
    String selectStatement = "select count(link) from links where " + relation + " = ?";
    PreparedStatement prepStmt =
    con.prepareStatement(selectStatement);
    prepStmt.setString(1, node);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
      nr = rs.getInt(1);
      prepStmt.close();
    }
    else {
      prepStmt.close();
      throw new NoSuchEntityException("Could not determine nr of links for " + relation + " for node "
+ node +
                " in database.");
    }

    if (nr == 0) {return links;}
    links = new String [nr];
    selectStatement = "select link from links where " + relation + " = ?";

    prepStmt = con.prepareStatement(selectStatement);

    prepStmt.setString(1, node);
    rs = prepStmt.executeQuery();

    for (int i=0; i < nr; i++){
      rs.next();
      String link = rs.getString(1);

      if (i < links.length) {
        links[i] = link; }
      else {
      throw new NoSuchEntityException("ARRAY BOUND ERROR !!!!"); }
```

```
    }

    prepStmt.close();

    return links;
}

/**********************************************************************
 *  Name : selectByPrimaryKey
 *  Description : Select node using primary key
 *  Input parameters : Node Name
 *  Return : boolean = status of retrieval
 **********************************************************************/

private boolean selectByPrimaryKey(String primaryKey) throws SQLException {

  String selectStatement =
      "select node " +
      "from nodes where node = ? ";
  PreparedStatement prepStmt =
      con.prepareStatement(selectStatement);
  prepStmt.setString(1, primaryKey);

  ResultSet rs = prepStmt.executeQuery();
  boolean result = rs.next();
  prepStmt.close();

  return result;
}

/**********************************************************************
 *  Name : insertPL
 *  Description : Update PI or LAMBDA or evidence
 *  Input parameters : Table Name, Values
 *  Return : none
 **********************************************************************/

private void insertPL (String TableName,Array PL) throws SQLException {

  String selectStatement =
      "select node " +
      "from " + TableName + " where node = ? ";
  PreparedStatement prepStmt =
      con.prepareStatement(selectStatement);

  prepStmt.setString(1, node);

  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()) {
    prepStmt.close();
    deletePL(TableName);
  }
```

```
  else {
    prepStmt.close();

  }

  String insertStatement = "insert into " + TableName + " values ( ? , ? , ? )";

  prepStmt = con.prepareStatement(insertStatement);
  prepStmt.setString(1, node);
  for (int cnt = 1;cnt <= PL.length();cnt++) {
    prepStmt.setInt(2, cnt);
    prepStmt.setDouble(3, PL.get(cnt-1));
    prepStmt.executeUpdate();
  }
  prepStmt.close();

}

/**********************************************************************
 *  Name : deletePL
 *  Description : Delete PI/LAMBDA/Evidence
 *  Input parameters : Table Name
 *  Return : none
 **********************************************************************/

private void deletePL(String TableName) throws SQLException {

  String deleteStatement =
  "delete from " + TableName + " where node = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(deleteStatement);

  prepStmt.setString(1, this.node);
  prepStmt.executeUpdate();
  prepStmt.close();

}

/**********************************************************************
 *  Name : loadPL
 *  Description : Load PI/LAMBDA/Evidence
 *  Input parameters : Table Name, Field Name, Array
 *  Return : none
 **********************************************************************/

private void loadPL(String TableName, String Field, Array arr) throws SQLException {

  String selectStatement =
  "select node, idx, " + Field +
  " from " + TableName + " where node = ? order by node,idx";

  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
```

```
    prepStmt.setString(1, this.node);

    ResultSet rs = prepStmt.executeQuery();
    int cnt = 1;
    while (rs.next()) {
      if (cnt <= arr.length()) {
        arr.set(cnt - 1, rs.getDouble(3));
        cnt++;}
    }
    prepStmt.close();

    if (trace) {System.out.println("\nNode" + node + " loadPL " + Field );

    for (cnt = 0; cnt < piX.length();cnt++){
      System.out.println(piX.get(cnt)+ ",");}

    System.out.println("\nEnd Node" + node + Field );}

  }


  /**********************************************************************
   *  Name : storePL
   *  Description : store PI/LAMBDA/Evidence
   *  Input parameters : Table Name, Field Name, Array
   *  Return : none
   **********************************************************************/

  private void storePL(String TableName, String Field, Array arr) throws SQLException {
    int rowCount = 0;
    String updateStatement = "update " + TableName + " set " + Field + " =  ? " +
    "where node = ? and idx = ?";
    PreparedStatement prepStmt =
    con.prepareStatement(updateStatement);
    prepStmt.setString(2, this.node);
    for (int cnt = 1;cnt <= arr.length(); cnt++) {
      prepStmt.setInt(3, cnt);
      prepStmt.setDouble(1, arr.get(cnt - 1));
      rowCount = prepStmt.executeUpdate();
    }
    prepStmt.close();

    if (rowCount == 0) {
      throw new EJBException("Storing pi or lambda for node " + this.node + " failed.");
    }
  }
} // NodeEJB
```

1.2 Node Home Interface

```
//////////////////////////////////////////////////////////////////////
/**
 * Name         NodeHome
 * Description : HOME INTERFACE FOR THE NODE ENTITY BEAN CLASS
 *
 *
 *
 *
 * Change Control
 *
 */

//////////////////////////////////////////////////////////////////////

import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.Collection;

public interface NodeHome extends EJBHome {
  public Node create(String node) throws CreateException, RemoteException;
  public Node findByPrimaryKey(String node) throws FinderException, RemoteException;
}
```

1.3 Node Remote Interface

```
////////////////////////////////////////////////////////////////////////
/**
 * Name         Node
 * Description : REMOTE INTERFACE FOR THE NODE ENTITY BEAN CLASS
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////////


import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Node extends EJBObject {
  public String getNode() throws RemoteException;
  public int getNrStates()throws RemoteException;
  public String [] getStates () throws RemoteException;
  public void reset () throws RemoteException;
  public String [] getOutgoingLinks ()throws RemoteException;
  public String [] getIncomingLinks ()throws RemoteException;
  public void setCPM(Matrix CPM) throws RemoteException;
  public Matrix getCPM() throws RemoteException;
  public Array multiplyTransposeCPM(Array productOfPis)throws RemoteException;
  public Array multiplyCPM(Array productOfPis)throws RemoteException;
  public Array getPI()throws RemoteException;
  public void setPI(Array pi)throws RemoteException;
  public Array getLAMBDA()throws RemoteException;
  public void setLAMBDA(Array lambda)throws RemoteException;
  public Array getBelief() throws RemoteException;
  public void setEvidence(int parentState,int state,boolean learn) throws RemoteException;
  public void setNoEvidence () throws RemoteException;
  public Array getEvidence()throws RemoteException;
}
```

2 Link Component

2.1 LinkEJB

```
////////////////////////////////////////////////////////////////////////
/**
 * Name            LinkEJB
 * Description : The link component administers the PI, LAMBDA and synchronization
 *               for a Bayesian Network Link
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////////
// Imports

import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
////////////////////////////////////////////////////////////////////////
/**
 * Implementation class of the LinkEJB entity bean.
 * Author Anet Potgieter (anet@bayesbean.com)
 * Version 1.0
 *
 * EJB
 *   Type:       Entity
 *   Description: LinkEJB Bean
 *   Home :      LinkHome
 *   Remote:     Link
 *   Persistence: Bean-managed
 *
 */

public class LinkEJB implements EntityBean {

  private String link;
  private String parent;
  private String child;
  private String [] otherParentLinks;
  private String [] otherChildrenLinks;

  private EntityContext context;
  private Connection con;
  private String dbName = "java:comp/env/jdbc/BabeDB";
  private int nrIndexes;
  private int [] cpmIndexes;
```

```
private Array lambda;
private Array pi;
private Array stretchPI;
private String PIFlag = "0";
private String LAMBDAFlag = "0";
private boolean allPIsCalculated = false;
private boolean allLAMBDAsCalculated = false;
private boolean setOutgoingLAMBDAsFlag = false;
private boolean setIncomingPIsFlag = false;
private int nrparentStates;
private int nrchildStates;
private int test = 1;
private boolean trace = false;

////////////////////////////////////////////////////////////////////////
// Business Routines
//

/**********************************************************************
 *  Name : reset
 *  Description : Reset pi, lambda and synchronization
 *                flags for this link
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void reset() {

 if (trace) {System.out.println("\n Resetting PI and Lambda for Link" + link);}

 for (int cnt = 0; cnt < pi.length();cnt++){
   pi.set(cnt,1.0) ;}
 for (int cnt = 0; cnt < lambda.length();cnt++){
   lambda.set(cnt,1.0) ;}

 stretchPI();

 PIFlag = "0";
 LAMBDAFlag = "0";
}

/**********************************************************************
 *  Name : setIncomingPIFlags
 *  Description : Set flags for incoming PIS
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

 public void setIncomingPIFlags() {
  setIncomingPIsFlag = true;
 }

 /**********************************************************************
```

```
 *  Name : setOutgoingLAMBDAFlags
 *  Description : Set flags for outgoing LAMBDAS
 *  Input parameters : none
 *  Return : none
 ***********************************************************************/

public void setOutgoingLAMBDAFlags() {
  setOutgoingLAMBDAsFlag = true;
}

/***********************************************************************
 *  Name : allIncomingPIsCalculated
 *  Description : Return flags for incoming PIS
 *  Input parameters : none
 *  Return : boolean
 ***********************************************************************/

public boolean allIncomingPIsCalculated() {
  return allPIsCalculated;
}

/***********************************************************************
 *  Name : allOutgoingLAMBDAsCalculated
 *  Description : Return synchronization flag for outgoing LAMBDAS
 *  Input parameters : none
 *  Return : boolean
 ***********************************************************************/

  public boolean allOutgoingLAMBDAsCalculated() {
    return allLAMBDAsCalculated;
  }

 /***********************************************************************
  *  Name : setPIFlag
  *  Description : Set synchronization flag PI
  *  Input parameters : none
  *  Return : none
  ***********************************************************************/

 public void setPIFlag() {
        PIFlag = "1";
 }

/***********************************************************************
 *  Name : setLAMBDAFlag
 *  Description : Set synchronization flag for LAMBDA
 *  Input parameters : none
 *  Return : none
 ***********************************************************************/

 public void setLAMBDAFlag() {
     LAMBDAFlag = "1";
 }
```

```
/*********************************************************************
 *  Name : getPIFlag
 *  Description : Get synchronization flag for PI
 *  Input parameters : none
 *  Return : String
 *********************************************************************/

 public String getPIFlag() {
   return PIFlag;
 }

/*********************************************************************
 *  Name : getLAMBDAFlag
 *  Description : Get synchronization flag for LAMBDA
 *  Input parameters : none
 *  Return : String
 *********************************************************************/

 public String getLAMBDAFlag () {
   return LAMBDAFlag;
 }

/*********************************************************************
 *  Name : getPI
 *  Description : Get PI for this Link
 *  Input parameters : none
 *  Return : Array
 *********************************************************************/

public Array getPI() {
   return pi;
}

/*********************************************************************
 *  Name : setPI
 *  Description : Set PI for this Link
 *  Input parameters : PI
 *  Return : none
 *********************************************************************/

public void setPI(Array pi) {


  if (pi.length() <= this.pi.length()) {
    for (int cnt = 0; cnt < pi.length();cnt++){
      this.pi.set(cnt,pi.get(cnt));
    }
    if (pi.length() != this.pi.length()) {
      if (trace) {System.out.println("\n LinkEJB : Invalid length for setting PI for Link" + link);}
    }
  }
  stretchPI();
```

```
  if (trace) {
    System.out.println("\n LinkEJB : Setting PI for Link" + link);
    for (int cnt = 0; cnt < pi.length();cnt++){
      System.out.println(pi.get(cnt) + ",");
    }
  }
}

/*****************************************************************
 *  Name : setLAMBDA
 *  Description : Set LAMBDA for this Link
 *  Input parameters : LAMBDA
 *  Return : none
 *****************************************************************/

public void setLAMBDA(Array lambda) {
  this.lambda = lambda;
}

/*****************************************************************
 *  Name : getLAMBDA
 *  Description : get LAMBDA for this Link
 *  Input parameters : none
 *  Return : LAMBDA
 *****************************************************************/

public Array getLAMBDA() {
  return lambda;
}

/*****************************************************************
 *  Name : getParent
 *  Description : get name of this Link's parent
 *  Input parameters : none
 *  Return : String
 *****************************************************************/

 public String getParent() {
  return parent;
 }

/*****************************************************************
 *  Name : getChild
 *  Description : get name of this Link's child
 *  Input parameters : none
 *  Return : String
 *****************************************************************/

 public String getChild() {
      return child;
 }
```

```
/*********************************************************************
 *  Name : getStretchedPi
 *  Description : Return Stretched PI
 *  Input parameters : none
 *  Return : Array
 *********************************************************************/

 public Array getStretchedPi (){
   return this.stretchPI;
 }

/*********************************************************************
 *  Name : getShrinkedLambda
 *  Description : Shrink a LAMBDA
 *  Input parameters : none
 *  Return : Array
 *********************************************************************/

public Array getShrinkedLambda (Array inArr,int nrStates){
  Array shrinkLAMBDA = new Array (nrStates,0.0);
  double sum;

  for (int cnt = 0; cnt < inArr.length(); cnt++){
    int idx = cpmIndexes[cnt];
    sum = shrinkLAMBDA.get(idx) + inArr.get(cnt);
    shrinkLAMBDA.set(idx,sum);
  }
  return shrinkLAMBDA;
}

/*********************************************************************
 *  Name : findOtherParentLinks
 *  Description : Return a list of other outgoing links of this link's
 *                parent node
 *  Input parameters : none
 *  Return : List of Strings
 *********************************************************************/

public String[] findOtherParentLinks(){
  return otherParentLinks;
}

/*********************************************************************
 *  Name : findOtherChildrenLinks
 *  Description : Return a list of other incoming links of this link's
 *                child node
 *  Input parameters : none
 *  Return : List of Strings
 *********************************************************************/

public String[] findOtherChildrenLinks(){
  return otherChildrenLinks;
}
```

```
////////////////////////////////////////////////////////////////////
// The following are required EJB methods, called by the container

/**************************************************************************************
 *  Name : ejbCreate
 *  Description : ejbCreate is called by the EJB container after setEntityContext
 *  Input parameters : Bayesian Network Link Name
 *  Return : String
 **************************************************************************************/

public String ejbCreate(String link)
  throws CreateException {

  try {
    this.link = link;
    selectParentAndChild();
    lambda = new Array(nrparentStates,1.0);
    //lambda = new Array(nrchildStates,1.0);
    pi = new Array(nrparentStates,1.0);
    loadPL("link_pi", "pi", pi);
    loadPL("link_lambda", "lambda", lambda);

    if (trace) {System.out.println("\n EJBCreate LinkEJB PI for Link" + link);
      for (int cnt = 0; cnt < pi.length();cnt++){
        System.out.println(pi.get(cnt) + ",");}}


    getNrIdxs(link);
    getCPMIndexes(link);
    stretchPI();

    loadFlags();
    allPIsCalculated = checkForIncomingPIs();
    allLAMBDAsCalculated = checkForOutgoingLAMBDAs();
    otherParentLinks = getOtherLinks("parent");
    otherChildrenLinks = getOtherLinks("child");

  } catch (Exception ex) {
    throw new EJBException("ejbCreate: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Link ejbCreate");}
  return link;
}

/*********************************************************************
 *  Name : ejbFindByPrimaryKey
 *  Description : Find the NodeEJB instance that matches the given primary key.
 *  Input parameters : Bayesian Network Node Name
 *  Return : String
 *********************************************************************/
```

```
public String ejbFindByPrimaryKey(String primaryKey)throws FinderException {

  boolean result;

  try {
    result = selectByPrimaryKey(primaryKey);
  } catch (Exception ex) {
    throw new EJBException("ejbFindByPrimaryKey: " +
    ex.getMessage());
  }

  if (result) {
    return primaryKey;
  }
  else {
    throw new ObjectNotFoundException
    ("Row for id " + primaryKey + " not found.");
  }

}

/*****************************************************************
 *  Name : ejbRemove
 *  Description : Called by container before data removed from database.
 *  Input parameters : none
 *  Return : none
 *****************************************************************/

public void ejbRemove() {
  if (trace) {System.out.println("\nLink " + link + "ejbRemove");}
  try {
  }
  catch (Exception ex) {
    throw new EJBException("ejbRemove: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Link " + link + " ejbRemove");}
}

/*****************************************************************
 *  Name : ejbLoad
 *  Description : Called by container to refresh entity Bean's state.
 *  Input parameters : none
 *  Return : none
 *****************************************************************/

public void ejbLoad() {
  if (trace) {System.out.println("\nLink " + link + " ejbLoad");}
  try {
    //lambda = new Array(nrparentStates,1.0);
    //lambda = new Array(nrchildStates,1.0);
    //pi = new Array(nrparentStates,1.0);
```

```
    loadPL("link_pi", "pi", pi);
    loadPL("link_lambda", "lambda", lambda);
    //getNrIdxs(link);
    //getCPMIndexes(link);
    stretchPI();
    loadFlags();
    allPIsCalculated = checkForIncomingPIs();
    allLAMBDAsCalculated = checkForOutgoingLAMBDAs();

  } catch (Exception ex) {
    throw new EJBException("ejbLoad: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Link " + link + " ejbLoad");}
}

/*********************************************************************
 *  Name : ejbStore
 *  Description : save Bean's state to database.
 *  Input parameters : none
 *  Return : none
 *********************************************************************/

public void ejbStore() {
  if (trace) {System.out.println("\nLink " + link + " ejbStore");}
  try {
    storePL("link_pi", "pi", pi);
    storePL("link_lambda", "lambda", lambda);
    setFlags(this.PIFlag, this.LAMBDAFlag);
    if (setIncomingPIsFlag  == true)
    {
      setPL("pi_flag", "child",this.child,"0");
      setIncomingPIsFlag  = false;
    }

    if (setOutgoingLAMBDAsFlag == true)
    {
      setPL("lambda_flag", "parent",this.parent,"0");
      setOutgoingLAMBDAsFlag = false;
    }
  }
  catch (Exception ex) {
    throw new EJBException("ejbLoad: " +
    ex.getMessage());
  }
  if (trace) {System.out.println("\nend Link " + link + " ejbStore");}
}

/*********************************************************************
 *  Name : ejbPostCreate
 *  Description : Called by container after ejbCreate.
 *  Input parameters : none
 *  Return : none
```

```
   ***********************************************************************/

   public void ejbPostCreate(String link) { }

////////////////////////////////////////////////////////////////////////
// The following are callback methods, called by the container to
// notify the Bean that some event is about to occur.

  /**********************************************************************
   *  Name : setEntityContext
   *  Description : Called by container to set Bean context.
   *  Input parameters : context
   *  Return : none
   **********************************************************************/

  public void setEntityContext(EntityContext context) {
    if (trace) {System.out.println("\nLink " + link + " setEntityContext");}
    this.context = context;
    try {
      makeConnection();
    }
    catch (Exception ex) {
      throw new EJBException("Unable to connect to database. " +
      ex.getMessage());
    }
    if (trace) {System.out.println("\nend Link " + link + " setEntityContext");}
  }

  /**********************************************************************
   *  Name : unsetEntityContext
   *  Description : Called by container to unset Bean context.
   *  Input parameters : context
   *  Return : none
   **********************************************************************/

  public void unsetEntityContext() {
    if (trace) {System.out.println("\nLink " + link + " unsetEntityContext");}
    try {
      con.close();
    }
    catch (SQLException ex) {
      throw new EJBException("unsetEntityContext: " + ex.getMessage());
    }
    if (trace) {System.out.println("\nend Link " + link + " unsetEntityContext");}
  }

  /**********************************************************************
   *  Name : ejbActivate
   *  Description : Called by container before Bean swapped into memory.
   *  Input parameters : none
   *  Return : none
   **********************************************************************/
```

```
  public void ejbActivate() {
    if (trace) {System.out.println("\nLink " + link + " ejbActivate");}
    link = (String)context.getPrimaryKey();
    if (trace) {System.out.println("\nend Link ejbActivate");}
  }

  /*********************************************************************
   *  Name : ejbPassivate
   *  Description : Called by container before Bean swapped into storage.
   *  Input parameters : none
   *  Return : none
   *********************************************************************/

  public void ejbPassivate() {
    if (trace) {System.out.println("\nLink " + link + " ejbPassivate");}
    link = null;
    if (trace) {System.out.println("\nend Link " + link + " ejbPassivate");}
  }

////////////////////////////////////////////////////////////////////////
/********************** Database Routines ************************/

  /*********************************************************************
   *  Name : makeConnection
   *  Description : Connect to the database
   *  Input parameters : none
   *  Return : none
   *********************************************************************/

  private void makeConnection() throws NamingException, SQLException {
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup(dbName);
    con =  ds.getConnection();
  }

  /*********************************************************************
   *  Name : getNrIdxs
   *  Description : Determine Number of Indexes for this link
   *  Input parameters : link
   *  Return : none
   *********************************************************************/

  private void getNrIdxs(String link) throws SQLException {
    if (trace) {System.out.println("\nLink " + link + "getNrIdxs");}
    //Determine nr indexes for this link
    String selectStatement = "select count(nr) from link_indexes where link = ?";
    PreparedStatement prepStmt =
    con.prepareStatement(selectStatement);
    prepStmt.setString(1, link);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
```

```
    this.nrIndexes = rs.getInt(1);
    if (trace) {System.out.println("\nLink " + link + " Nr of Indexes = " + nrIndexes + "\n");}
   }
   prepStmt.close();
}

/**********************************************************************
 *  Name : getCPMIndexes
 *  Description : Retrieve CPM Indexes
 *  Input parameters : link
 *  Return : none
 **********************************************************************/

private void getCPMIndexes(String link) throws SQLException {
  if (trace) {System.out.println("\nLink " + link + "getCPMIndexes");}
  cpmIndexes = new int[nrIndexes];

  String selectStatement = "select nr, index " +
    "from link_indexes where link = ? order by nr";
  PreparedStatement prepStmt = con.prepareStatement(selectStatement);

  prepStmt.setString(1, link);
  ResultSet rs = prepStmt.executeQuery();

  for (int i=0; i < nrIndexes; i++){
    rs.next();
    int idx = rs.getInt(2);
    cpmIndexes[i] = idx;
  }

  prepStmt.close();

  if (trace) {System.out.println("\nend Link getCPMIndexes");}
}

/**********************************************************************
 *  Name : selectByPrimaryKey
 *  Description : Select Link by Primary Key
 *  Input parameters : primaryKey
 *  Return : boolean
 **********************************************************************/

private boolean selectByPrimaryKey(String primaryKey)
  throws SQLException {
  String selectStatement =
    "select link " +
    "from links where link = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, primaryKey);

  ResultSet rs = prepStmt.executeQuery();
  boolean result = rs.next();
```

```
  prepStmt.close();
  return result;
}

/*********************************************************************
 *  Name : insertPL
 *  Description : Insert PI/LAMBDA for this link
 *  Input parameters : Table Name, PI/LAMBDA
 *  Return : none
 *********************************************************************/

private void insertPL (String TableName,Array PL) throws SQLException {
  String selectStatement =
    "select link " +
    "from " + TableName + " where link = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, link);

  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()) {
    prepStmt.close();
    deletePL(TableName);
  }
  else {
    prepStmt.close();
  }

  String insertStatement = "insert into " + TableName + " values ( ? , ? , ? )";
  prepStmt = con.prepareStatement(insertStatement);
  prepStmt.setString(1, link);
  for (int cnt = 1;cnt <= PL.length();cnt++) {
    prepStmt.setInt(2, cnt);
    prepStmt.setDouble(3, PL.get(cnt-1));
    prepStmt.executeUpdate();
  }
  prepStmt.close();
}

/*********************************************************************
 *  Name : deletePL
 *  Description : Delete PI/LAMBDA for this link
 *  Input parameters : Table Name
 *  Return : none
 *********************************************************************/

private void deletePL(String TableName) throws SQLException {
  String deleteStatement =
    "delete from " + TableName + " where link = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(deleteStatement);
```

```
  prepStmt.setString(1, this.link);
  prepStmt.executeUpdate();
  prepStmt.close();
}

/*********************************************************************
 *  Name : loadPL
 *  Description : Load PI or LAMBDA for this link
 *  Input parameters : Table Name, Field Name, PI/LAMBDA
 *  Return : none
 *********************************************************************/

private void loadPL(String TableName, String Field, Array arr) throws SQLException {
  String selectStatement =
    "select link, idx, " + Field +
    " from " + TableName + " where link = ? order by link,idx";

  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);

  prepStmt.setString(1, this.link);

  ResultSet rs = prepStmt.executeQuery();
  int cnt = 1;
  while (rs.next()) {
    if (cnt <= arr.length()) {
      arr.set(cnt - 1, rs.getDouble(3));
      cnt++;}
  }
  prepStmt.close();
}

/*********************************************************************
 *  Name : storePL
 *  Description : Store PI or LAMBDA for this link
 *  Input parameters : Table Name, Field Name, PI/LAMBDA
 *  Return : none
 *********************************************************************/

private void storePL(String TableName, String Field, Array arr) throws SQLException {
  int rowCount = 0;

  String updateStatement = "update " + TableName + " set " + Field + " =  ? " +
    "where link = ? and idx = ?";

  PreparedStatement prepStmt =
  con.prepareStatement(updateStatement);
  prepStmt.setString(2, this.link);
  for (int cnt = 1;cnt <= arr.length(); cnt++) {
    prepStmt.setInt(3, cnt);
    prepStmt.setDouble(1, arr.get(cnt - 1));
    rowCount = prepStmt.executeUpdate();
  }
```

```
  prepStmt.close();
}

/*********************************************************************
 *  Name : setPL
 *  Description : Update pi-flags/lambda_flag for link
 *  Input parameters : Field Name, Relation specification,relation,val
 *  Return : none
 *********************************************************************/

private void setPL(String Field, String relationSpec,String relation,String val) throws SQLException {
  int rowCount;

  String updateStatement = "update links set " + Field + " =  ? " +
    "where " + relationSpec + " = ?";

  PreparedStatement prepStmt =
  con.prepareStatement(updateStatement);
  prepStmt.setString(1, val);
  prepStmt.setString(2, relation);

  rowCount = prepStmt.executeUpdate();
  prepStmt.close();

  if (rowCount == 0) {
    throw new EJBException("Resetting incoming pi or lambda flags for child " + child + " failed.");
  }

}

/*********************************************************************
 *  Name : selectParentAndChild
 *  Description : Select the parent and child for this link
 *  Input parameters : none
 *  Return : none
 *********************************************************************/


private void selectParentAndChild()
  throws SQLException {

  String selectStatement =
    "select parent,child " +
    "from links where link = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, link);

  ResultSet rs = prepStmt.executeQuery();
  rs.next();
  parent = rs.getString(1);
  child = rs.getString(2);
  prepStmt.close();
```

```
    selectStatement =
      "select count(state)" +
      "from states where node = ? ";
    prepStmt =
    con.prepareStatement(selectStatement);
    prepStmt.setString(1, parent);

    rs = prepStmt.executeQuery();
    rs.next();
    nrparentStates = rs.getInt(1);
    if (trace) {System.out.println("\nLink " + link + "'s Parent " + parent + " has " + nrparentStates + "
States");}
    prepStmt.setString(1, child);
    rs = prepStmt.executeQuery();
    rs.next();
    nrchildStates = rs.getInt(1);
    prepStmt.close();
    if (trace) {System.out.println("\nLink " + link + "'s Child has " + nrchildStates + " States");}
  }

  /*****************************************************************************
   *  Name : getOtherLinks
   *  Description : Get other outgoing links of parent / other incoming links of child
   *  Input parameters : relation
   *  Return : List of Strings
   *****************************************************************************/


  private String[] getOtherLinks(String relation){
  String [] otherLinks;

  try {
    otherLinks = selectOtherLinks(relation);
  }
  catch (Exception ex) {
    throw new EJBException("getOtherLinks: " +
    ex.getMessage());
  }
  return otherLinks;
  }

  /*****************************************************************************
   *  Name : selectOtherLinks
   *  Description : Finds other sibling links for this link
   *  Input parameters : relation
   *  Return : List of Strings
   *****************************************************************************/

  private String[] selectOtherLinks(String relation) throws SQLException {
    // Determine nr other incoming/outgoing links for this link
    String [] a = new String[0];
    int nrOtherLinks = 0;
```

```
  String selectStatement = "select count(link) from links where " + relation + " = ? and not link = ?";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);

  if (relation.compareTo("parent") == 0) {
    prepStmt.setString(1, parent);}
  else {
    prepStmt.setString(1, child);}

  prepStmt.setString(2, link);
  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()) {
    nrOtherLinks = rs.getInt(1);
  }
  prepStmt.close();

  // Retrieve the siblings and return in an array

  if (nrOtherLinks == 0) {
    return a;
  }

  selectStatement = "select link from links where " + relation + " = ? and not link = ?";
  prepStmt = con.prepareStatement(selectStatement);

  if (relation.compareTo("parent") == 0) {
    prepStmt.setString(1, parent);}
  else {
    prepStmt.setString(1, child);}

  prepStmt.setString(2, link);
  rs = prepStmt.executeQuery();
  a = new String[nrOtherLinks];
  int cnt = 0;

  while (rs.next()) {
    String otherLink = rs.getString(1);
    a[cnt] = otherLink;
    cnt++;
  }
  prepStmt.close();
  return a;
}

/*****************************************************************************
 *  Name : selectOtherLinks
 *  Description : Store PI and LAMBDA flags for this link
 *  Input parameters : PIFlag, LambdaFlag
 *  Return : none
 *****************************************************************************/

private void setFlags(String PIFlg, String LAMBDAFlg) throws SQLException {
```

```
  int rowCount = 0;

  String updateStatement = "update links set pi_flag = ?, lambda_flag = ?" +
    " where link = ?";
  PreparedStatement prepStmt =
  con.prepareStatement(updateStatement);
  prepStmt.setString(1, PIFlg);
  prepStmt.setString(2, LAMBDAFlg);
  prepStmt.setString(3, this.link);
  rowCount = prepStmt.executeUpdate();
  prepStmt.close();

  if (rowCount == 0) {
    throw new EJBException("Storing pi or lambda for link " + this.link + " failed.");
  }
}

/*****************************************************************************
 *  Name : loadFlags
 *  Description : load Flags for this link
 *  Input parameters : none
 *  Return : none
 *****************************************************************************/

private void loadFlags() throws SQLException {

  String selectStatement =
    "select pi_flag, lambda_flag " +
    "from links where link = ? ";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, link);

  ResultSet rs = prepStmt.executeQuery();
  rs.next();
  this.PIFlag = rs.getString(1);
  this.LAMBDAFlag = rs.getString(2);
  prepStmt.close();

}

/*****************************************************************************
 *  Name : checkForIncomingPIs
 *  Description : See if all incoming PI's were calculated for child
 *  Input parameters : none
 *  Return : boolean
 *****************************************************************************/

private boolean checkForIncomingPIs() throws SQLException {

  boolean returnVal;
  String selectStatement =
```

```
  "select link " +
  "from links where not parent = ? and child = ? and pi_flag = ?";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, parent);
  prepStmt.setString(2, child);
  prepStmt.setString(3,"0");

  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()){
    returnVal = false; }
  else {
    returnVal = true ; }
  prepStmt.close();

  return returnVal;
}

/*****************************************************************************
 *  Name : checkForOutgoingLAMBDAs
 *  Description : See if all outgoing LAMBDA's were calculated for parent
 *  Input parameters : none
 *  Return : boolean
 *****************************************************************************/

private boolean checkForOutgoingLAMBDAs() throws SQLException {
  boolean returnVal;

  String selectStatement =
    "select link " +
    "from links where not child = ? and parent = ? and lambda_flag = ?";
  PreparedStatement prepStmt =
  con.prepareStatement(selectStatement);
  prepStmt.setString(1, child);
  prepStmt.setString(2, parent);
  prepStmt.setString(3,"0");

  ResultSet rs = prepStmt.executeQuery();

  if (rs.next()){
    returnVal = false; }
  else {
    returnVal = true ; }
  prepStmt.close();

  return returnVal;
}

/*****************************************************************************
 *  Name : stretchPI
 *  Description : Stretch PI for this parent so that it can by multiplied with PI's
 *                of siblings
```

```
   *  Input parameters : none
   *  Return : none
   ********************************************************************************/

  private void stretchPI() {
    if (trace) {System.out.println("\n LinkEJB stretching PI Link " + link + " NrIndexes: " + nrIndexes);}
    stretchPI = new Array (nrIndexes,1.0);

    if (trace) {System.out.println("\n LinkEJB: PI for Link " + link );

    for (int cnt = 0; cnt < pi.length();cnt++){
      System.out.println(pi.get(cnt) + ",");}

    System.out.println("\n LinkEJB: END PI for Link " + link );}

    for (int cnt = 0; cnt < nrIndexes; cnt++){
      int idx = cpmIndexes[cnt];

      if (idx < pi.length ())
        {stretchPI.set(cnt,pi.get(idx));}
      else
        {stretchPI.set(cnt,1.0);
      System.out.println("\n LinkEJB: WARNING  setting stretched PI[" + cnt + "] to 1.0 for Link " + link
);}
    }
    if (trace) {System.out.println("\nend LinkEJB stretchedPI");}
  }
} // LinkEJB
```

2.2 Link Home

```
////////////////////////////////////////////////////////////////////////
/**
 * Name         LinkHome
 * Description : HOME INTERFACE FOR THE LINK ENTITY BEAN CLASS
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////////

import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.Collection;

public interface LinkHome extends EJBHome {

public Link create(String link)
      throws CreateException, RemoteException;

public Link findByPrimaryKey(String link)
      throws FinderException, RemoteException;


}
```

2.3 Link Remote

```
////////////////////////////////////////////////////////////////////////
/**
 * Name         Link
 * Description : REMOTE INTERFACE FOR THE LINK ENTITY BEAN CLASS
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////////

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

  public interface Link extends EJBObject {
    public void reset () throws RemoteException;
    public String[] findOtherParentLinks()throws RemoteException;
    public String[] findOtherChildrenLinks()throws RemoteException;
    public Array getPI()throws RemoteException;
    public void setPI(Array pi) throws RemoteException;
    public void setLAMBDA(Array lambda) throws RemoteException;
    public Array getLAMBDA()throws RemoteException;
    public String getParent()throws RemoteException;
    public String getChild()throws RemoteException;
    public void setPIFlag() throws RemoteException;
    public void setLAMBDAFlag() throws RemoteException;
    public String getPIFlag() throws RemoteException;
    public String getLAMBDAFlag () throws RemoteException;
    public Array getStretchedPi ()throws RemoteException;
    public Array getShrinkedLambda (Array inArr,int nrStates) throws RemoteException;
    public void setIncomingPIFlags () throws RemoteException;
    public void setOutgoingLAMBDAFlags()throws RemoteException;
    public boolean allIncomingPIsCalculated() throws RemoteException;
    public boolean allOutgoingLAMBDAsCalculated() throws RemoteException;
}
```

3 Belief Propagation Component

```
////////////////////////////////////////////////////////////////////////////
/**
 * Name            MessageBean
 * Description : This belief propagation component implements the local belief
 *               propagation on a link between a parent and a child node
 *
 *
 *
 *
 * Change Control
 *
 */

////////////////////////////////////////////////////////////////////
// Imports


import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.CreateException;
import javax.naming.*;
import javax.jms.*;
import java.util.*;
import javax.rmi.PortableRemoteObject;
import javax.transaction.*;

////////////////////////////////////////////////////////////////////
/**
 * The MessageBean class is a message-driven bean.  It implements
 * the javax.ejb.MessageDrivenBean and javax.jms.MessageListener
 * interfaces. It is defined as public (but not final or
 * abstract).  It defines a constructor and the methods ejbCreate,
 * onMessage, setMessageDrivenContext, and ejbRemove.
 */

public class MessageBean implements MessageDrivenBean, MessageListener {
  QueueConnection     queueConnection = null;
  private Context newContext;

  private transient MessageDrivenContext mdc = null;
  private boolean trace = false;
  private boolean trace2 = true;
  private boolean propagateFlag = false;

  /**
  * Constructor, which is public and takes no arguments.
  */
```

```
public MessageBean() {
//System.out.println("In MessageBean.MessageBean()");
}

/**********************************************************************
 *  Name : setMessageDrivenContext method
 *  Description : declared as public (but not final or static)
 *  Input parameters : javax.ejb.MessageDrivenContext
 *  Return : none
 **********************************************************************/

public void setMessageDrivenContext(MessageDrivenContext mdc)
{
  this.mdc = mdc;
}

/**********************************************************************
 *  Name : ejbCreate method
 *  Description : declared as public (but not final or static)
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void ejbCreate() {

  try {
    newContext = new InitialContext();
    if (trace) {System.out.println("In MessageBean.ejbCreate()");}

    }
    catch (Throwable t) {

      // JMSException or NamingException could be thrown
      t.printStackTrace();
  }
}

/**********************************************************************
 *  Name : ejbCreate method
 *  Description : Closes the QueueConnection.
 *  Input parameters : none
 *  Return : none
 **********************************************************************/

public void ejbRemove() {
  Link link;

  try {

  if (trace) {System.out.println("In MessageBean.ejbRemove()");}

  if(queueConnection != null) {
    queueConnection.close();};
```

```
  }
  catch (Throwable t) {

    // JMSException or NamingException could be thrown
    t.printStackTrace();
  }

}

/*********************************************************************
 *  Name : onMessage method
 *  Description : Handles the incoming message
 *  Input parameters : inMessage the incoming message
 *  Return : none
 *********************************************************************/

public void onMessage(Message inMessage) {
  TextMessage msg = null;
  Queue queue = null;

  try {

    queue = (Queue)inMessage.getJMSDestination();
    String dest = queue.getQueueName();

    if (inMessage instanceof TextMessage) {
      msg = (TextMessage) inMessage;

      String tag = msg.getText();
      if (trace2) {System.out.println("MessageBean : " + dest + " received a Message" + tag);}

      if (tag.compareTo("PI") == 0) {
        handlePI(dest);
      }

      if (tag.compareTo("LAMBDA") == 0) {
        handleLAMBDA(dest);
      }

      if (tag.compareTo("PAUSE") == 0)
      {
       propagateFlag = false;
      }

      if (tag.compareTo("RESUME") == 0)
      {
       propagateFlag = true;
      }
    }
  }
  catch (Throwable t) {
```

```
    // JMSException could be thrown
    t.printStackTrace();

  }
}

/**********************************************************************
 *  Name : handlePI
 *  Description : Handles the incoming PI message
 *  Input parameters : Queue name
 *  Return : none
 **********************************************************************/

private void handlePI(String destination)
{

  int cnt = 0;
  Link link;
  Link otherLink;
  Array LinkPI = new Array(0);
  Array productOfLambdas = new Array(0);
  Array productOfPis = new Array(0);
  Array piX = new Array(0);
  String parent;
  String child;

  Object linkObjref;
  Object nodeObjref;
  NodeHome nodeHome;
  LinkHome linkHome;
  Node node;
  Node childNode;
  Node parentNode;
  String [] otherIncomingLinks = new String[0];
  String [] otherOutgoingLinks = new String[0];
  String [] ChildOutgoingLinkNames = new String[0];
  String [] ChildIncomingLinkNames = new String[0];

  try {

    linkObjref = newContext.lookup("java:comp/env/ejb/LinkBean");
    linkHome = (LinkHome)PortableRemoteObject.narrow(linkObjref, LinkHome.class);

    nodeObjref = newContext.lookup("java:comp/env/ejb/NodeBean");
    nodeHome = (NodeHome)PortableRemoteObject.narrow(nodeObjref, NodeHome.class);

    link = linkHome.create(destination);
    parent = link.getParent();
    child = link.getChild();

    // Get a list of other outgoing links
    // ----------------------------------
    otherOutgoingLinks = link.findOtherParentLinks();
```

```
// Get a list of other incoming links
// ----------------------------------
otherIncomingLinks = link.findOtherChildrenLinks();
link.remove();

// calculate product of lambdas for other outgoing links
// -----------------------------------------------------

for (cnt = 0; cnt <  otherOutgoingLinks.length; cnt++) {

  otherLink = linkHome.create(otherOutgoingLinks[cnt]);
  if (cnt == 0) {
    productOfLambdas = otherLink.getLAMBDA();
  }
  else {
    productOfLambdas = productOfLambdas.termProduct(otherLink.getLAMBDA());
  }
  otherLink.remove();
}

if (trace) {System.out.println(destination + "Product of Lambdas: [");
  for (cnt = 0; cnt < productOfLambdas.length();cnt++){
    System.out.println(productOfLambdas.get(cnt) + ",");}}


// Get parent's PI
// -----------------------------------

parentNode = nodeHome.create(parent);
//piX = parentNode.getPI();
piX = parentNode.getBelief();

String [] parentIncomingLinkNames = parentNode.getIncomingLinks ();

// calculate PI for this link
// --------------------------

if (otherOutgoingLinks.length > 0 ) {
  LinkPI = piX.termProduct(productOfLambdas);}
else {
  LinkPI = piX;
}

link = linkHome.create(destination);

// Set PI for this link
// --------------------
if (trace2) {System.out.println(destination + ": PI : [");
  for (cnt = 0; cnt < LinkPI.length();cnt++){
    System.out.println(LinkPI.get(cnt) + ",");}}

link.setPI(LinkPI);
```

```
// Set PI flag for this link
// ------------------------
link.setPIFlag();

// Test if other PI's were calculated for child's other incoming links
// -------------------------------------------------------------------

boolean testLinks = link.allIncomingPIsCalculated();

if (testLinks) {
  link.setIncomingPIFlags();
}

if (testLinks) {
  // Calculate PI for child node
  // ---------------------------


  if (trace2) {System.out.println("Can now calculate child " + child + "'s PI !");}
  productOfPis = link.getStretchedPi();
  link.remove();

  for (cnt = 0; cnt <  otherIncomingLinks.length; cnt++) {

    // For each other incoming link
    // ----------------------------
    otherLink = linkHome.create(otherIncomingLinks[cnt]);

    // calculate product of pis of other incoming links
    // ------------------------------------------------

    productOfPis = productOfPis.termProduct(otherLink.getStretchedPi());
    otherLink.remove();

  }

  childNode = nodeHome.create(child);
  ChildOutgoingLinkNames = childNode.getOutgoingLinks ();
  ChildIncomingLinkNames = parentNode.getIncomingLinks ();

  Array pi = childNode.multiplyTransposeCPM(productOfPis);

  if (trace2) {System.out.println(child + "'s PI : [");
    for (cnt = 0; cnt < pi.length();cnt++){
      System.out.println(pi.get(cnt) + ",");}}

  childNode.setPI(pi);

  // PROPAGATE TO CHILDREN
  // ---------------------

  // If not a leaf node, propagate PIs downwards
```

```
      if (ChildOutgoingLinkNames.length > 0 ) {
        for (int childCnt = 0; childCnt < ChildOutgoingLinkNames.length; childCnt++) {
          System.out.println("\n Propagating PI message to " + ChildOutgoingLinkNames[childCnt] +
          " from " + destination);
          sendMsg(ChildOutgoingLinkNames[childCnt],"PI",destination);
        }
      }
      else
      {
        if (propagateFlag) {
          for (int childCnt = 0; childCnt < ChildIncomingLinkNames.length; childCnt++) {
            System.out.println("\n Propagating LAMBDA message to " +
ChildIncomingLinkNames[childCnt] +
            " from " + destination);
            sendMsg(ChildIncomingLinkNames[childCnt],"LAMBDA",destination);
          }
        }
      }
    }
    else {
      link.remove();
      System.out.println(destination + ": Can not calculate " + child + "'s PI yet !");
    }

  } catch (Throwable t) {
    // JMSException could be thrown
    t.printStackTrace();
  }
 }

 /**********************************************************************
  *  Name : handleLAMBDA
  *  Description : Handles the incoming LAMBDA message
  *  Input parameters : Queue name
  *  Return : none
  **********************************************************************/

 private void handleLAMBDA(String destination)
 {
  int cnt = 0;
  Link link;
  Link otherLink;
  Array LinkLambda = new Array(0);
  Array productOfLambdas = new Array(0);
  Array productOfPis = new Array(0);
  Array lambdaY = new Array(0);
  Array longLambda = new Array(0);
  String parent;
  String child;

  Object linkObjref;
  Object nodeObjref;
```

```
NodeHome nodeHome;
LinkHome linkHome;
Node node;
Node childNode;
Node parentNode;
String [] otherIncomingLinks = new String[0];
String [] otherOutgoingLinks = new String[0];
String [] parentIncomingLinkNames = new String[0];
String [] parentOutgoingLinkNames = new String[0];

try {


  linkObjref = newContext.lookup("java:comp/env/ejb/LinkBean");
  linkHome = (LinkHome)PortableRemoteObject.narrow(linkObjref, LinkHome.class);

  nodeObjref = newContext.lookup("java:comp/env/ejb/NodeBean");
  nodeHome = (NodeHome)PortableRemoteObject.narrow(nodeObjref, NodeHome.class);


  link = linkHome.create(destination);

  parent = link.getParent();
  child = link.getChild();

  // Get a list of other outgoing links
  // ----------------------------------
  otherOutgoingLinks = link.findOtherParentLinks();

  // Get a list of other incoming links
  // ----------------------------------
  otherIncomingLinks = link.findOtherChildrenLinks();

  // calculate product of pis for other incoming links
  // -------------------------------------------------
  productOfPis = link.getStretchedPi();
  link.remove();

  for (cnt = 0; cnt <  otherIncomingLinks.length; cnt++) {

    // For each other incoming link
    // ----------------------------
    otherLink = linkHome.create(otherIncomingLinks[cnt]);

    // calculate product of pis of other incoming links
    // ------------------------------------------------


    if (cnt == 0) {
      productOfPis = otherLink.getStretchedPi();
    }
    else
    {
```

```
   productOfPis = productOfPis.termProduct(otherLink.getStretchedPi());
  }
  otherLink.remove();

}

if ((trace) & (otherIncomingLinks.length > 0)) {System.out.println(destination + ": productOfPis : [");
  for (cnt = 0; cnt < productOfPis.length();cnt++){
    System.out.println(productOfPis.get(cnt) + ",");}}

childNode = nodeHome.create(child);


// calculate LAMBDA for this link
// ------------------------------

lambdaY = childNode.getLAMBDA();

if (trace) {System.out.println(destination + "Got Child " + child + "'s LAMBDA : [");
  for (cnt = 0; cnt < lambdaY.length();cnt++){
  System.out.println(" Child " + child + lambdaY.get(cnt) + ",");}}


String [] childOutgoingLinkNames = childNode.getOutgoingLinks ();

Array lambdaProd = childNode.multiplyCPM(lambdaY);

if (otherIncomingLinks.length > 0 ) {
  longLambda = lambdaProd.termProduct(productOfPis);}
else {
  longLambda = lambdaProd; }

parentNode = nodeHome.create(parent);
int nrStates = parentNode.getNrStates ();
parentNode.remove();

link = linkHome.create(destination);
LinkLambda = link.getShrinkedLambda (longLambda,nrStates);

// Set LAMBDA for this link
// --------------------
if (trace2) {System.out.println(destination + ": LAMBDA : [");
  for (cnt = 0; cnt < LinkLambda.length();cnt++){
    System.out.println(LinkLambda.get(cnt) + ",");}}

link.setLAMBDA(LinkLambda);

// Set LAMBDA flag for this link
// -------------------------

link.setLAMBDAFlag();

// Test if other LAMBDA's were calculated for parent's other outgoing links
```

```
// ------------------------------------------------------------------

boolean testLinks = link.allOutgoingLAMBDAsCalculated();
if (testLinks) {
  link.setOutgoingLAMBDAFlags();
}

if (testLinks) {
  // Calculate LAMBDA for parent node
  // --------------------------------
  System.out.println(destination + ":Can now calculate " + parent + "'s LAMBDA !");
  productOfLambdas = link.getLAMBDA();
  link.remove();


  for (cnt = 0; cnt <  otherOutgoingLinks.length; cnt++) {

    // For each other outgoing link
    // ----------------------------
    otherLink = linkHome.create(otherOutgoingLinks[cnt]);

    // calculate product of lambdas of other outgoing links
    // ----------------------------------------------------

    productOfLambdas = productOfLambdas.termProduct(otherLink.getLAMBDA());
    otherLink.remove();

  }

  parentNode = nodeHome.create(parent);
  parentIncomingLinkNames = parentNode.getIncomingLinks ();
  parentOutgoingLinkNames = parentNode.getOutgoingLinks ();

  Array lambda = productOfLambdas;

  if (trace2) {System.out.println(parent + ": LAMBDA : [");
    for (cnt = 0; cnt < lambda.length();cnt++){
      System.out.println(lambda.get(cnt) + ",");}
  }

  parentNode.setLAMBDA(lambda);

  // If not a root node, propagate lambdas upwards

  if (parentIncomingLinkNames.length > 0 ) {

    for (int parentCnt = 0; parentCnt < parentIncomingLinkNames.length; parentCnt++) {

    if (trace2) {System.out.println("\n Propagating LAMBDA message to " +
parentIncomingLinkNames[parentCnt] + " from " + destination);}
      sendMsg(parentIncomingLinkNames[parentCnt],"LAMBDA",destination);

    }
```

```
      }
      else { // root node - propagate PI's downwards
        for (int childCnt = 0; childCnt < parentOutgoingLinkNames.length; childCnt++) {

        if (trace2) {System.out.println("\n Propagating PI message to " +
parentOutgoingLinkNames[childCnt] + " from " + destination);}
          sendMsg(parentOutgoingLinkNames[childCnt],"PI",destination);
        }
      }
    }
    else {
      link.remove();
      System.out.println(destination + ":Can not calculate " + parent + "'s LAMBDA yet !");}
  }
  catch (Throwable t) {

    // JMSException could be thrown
    t.printStackTrace();

  }

 }
 /**********************************************************************
  *  Name : sendMsg
  *  Description : Send Message to queue
  *  Input parameters : Link Name, Message,
  *  Return : none
  **********************************************************************/


 private void sendMsg(String linkName,String msg, String dest) {
  Context               jndiContext = null;
  QueueConnectionFactory  queueConnectionFactory = null;
  QueueConnection         queueConnection = null;
  QueueSession            queueSession = null;
  Queue                 queue = null;
  QueueSender             queueSender = null;
  TextMessage             message = null;
  String queueName = "java:comp/env/jms/" + linkName;
  String queueConnectionFactoryName = "java:comp/env/jms/" + linkName + "CF";
  /*
  * Create a JNDI InitialContext object if none exists yet.
  */
  jndiContext = newContext;
  /*
  * Look up connection factory and queue.  If either does
  * not exist, exit.
  */
  try {
    queue = (Queue)jndiContext.lookup(queueName);
    queueConnectionFactory =
(QueueConnectionFactory)jndiContext.lookup(queueConnectionFactoryName);
  }
```

```
    catch (NamingException e) {
      System.out.println("Could not create JNDI " + "context: " + e.toString() + "from " + dest);
      System.exit(1);
    }

    try {
      queueConnection = queueConnectionFactory.createQueueConnection();
      queueSession =
      queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
      queueSender = queueSession.createSender(queue);

      message = queueSession.createTextMessage();
      message.setJMSDestination(queue);
      message.setText(msg);
      queueSender.send(message);
      if (queueSession != null) {
        queueSession.close();
      }
      if (queueConnection != null) {
        queueConnection.close();
      }
    }
    catch (JMSException e)
    {

      System.out.println("Exception occurred: " +
      e.toString());

    }
  }

}
```