Research Progress Report 3:
Distributed Reflective Architectures
Thesis Proposal

Catriona M. Kennedy

Supervisor: Prof. Aaron Sloman

Other members of thesis group:
Prof. John Barnden,
Dr. William Edmondson

June 1999

Abstract

The autonomy of a system can be defined as its capability to recover from unforeseen difficulties without any user intervention. This thesis proposal addresses a small part of this problem, namely the detection of anomalies within a system's own operation by the system itself. It is a response to a challenge presented by immune systems which can distinguish between "self" and "nonself", i.e. they can recognise a "foreign" pattern (due to a virus or bacterium) as *different* from those associated with the organism itself, even if the pattern was not previously encountered. The aim is to apply this requirement to an artificial system, where "nonself" may be any form of deliberate intrusion or random anomalous behaviour due to a fault.

When designing reflective architectures or self-diagnostic systems, it is simpler to rely on a single coordination mechanism to make the system work as intended. However, such a coordination mechanism cannot be inspected or repaired by the system itself, which means that there is a gap in its reflective coverage.

To try to overcome this limitation, this thesis proposal suggests a conceptual framework based on a network of agents where each agent monitors the whole network from a unique and independent perspective and where the perspectives are not globally "managed". Each agent monitors the fault-detection capability and control algorithms of other agents (a process called meta-observation). In this way, the agents can collectively achieve reflective coverage of failures.

Keywords: anomaly, meta-level closure, immune system, meta-observation, perspectives, reflection.

1 Introduction

This study is a response to a challenge presented by the emerging area of artificial immune systems (AIS). The goal of AIS is to detect computer viruses or other forms of intrusion which do not necessarily follow a known pattern (in contrast with typical virus scanning software). In other words, such systems are looking for *anomalies* which may be defined as any deviation from a "normal" pattern of activity. In the language of immunology, the system must distinguish between "self" and "nonself". For details, see e.g. Forrest et.al. [10].

It is the aim of this study to apply this requirement to autonomous agent architectures in order to make the agent less vulnerable in a hostile environment, i.e. the autonomous agent should distinguish between self and nonself. The virus detection problem is only one possible application domain.

1.1 Macro- and Micro-level

The requirement for self/nonself distinction is expressed on the "macro-level", i.e. it applies to the whole system viewed externally, without regard to its internal structure. This particular form of specifying the requirement is important because it draws attention to problems which would otherwise be ignored (see later). However, to make it useful for engineering design we must also ask the question: what internal micro-structure can be an underlying mechanism for the required description on the macro-level?

The term "agent" can be used on both the macro- and micro-levels. First, we are looking for an architecture for an *autonomous* agent which is defined on the macro-level and should distinguish between self and nonself. Secondly, there are entities on the micro-level of this architecture which may be called *non*-autonomous agents (because they are parts of a whole). As a convention, I will use the term "agent" for the micro-level entities (in the sense of participants in a "society" [25] while the macro-level system will be called the autonomous system.

1.2 Relationship to Cognitive Architectures

The macro-level requirement for self/nonself distinction should not be confused with human-like self-reflection as the latter is constrained by issues such as limited capacity and the complex, psychological understanding of "self", neither of which is relevant to the problem. Use of the term "self" in immune systems relates only to the identity of subject and object (i.e. the entity that is distinguishing/protecting etc. is the same as that being distinguished from other things and protected).

In spite of this, some methods for modelling human reflection (e.g. Sloman's work on meta-management [29]) may still be useful as design techniques if we ignore the specifically human-like constraints.

1.3 Meta-levels and "Aboutness"

In this report, the term "reflection" is used in the very broad sense used in Maes [20], namely a computational system is reflective if it is "about" parts of itself. First, this means that a computation is divided into a *meta-level* which does the reflection and an *object level* (sometimes called the "base" level) which does the actual work. There are many ways in which something can be a meta-level. For example, an agent architecture can have a meta-level that can reason about the agent's internal states and actions. This may or may not include control or modification of these states. Such a meta-level may be embedded in a reflective rulebase, where some meta-rules are "about" the interpretation of rules, e.g. choosing which rules to apply and recording their success and failure rate (EURISKO [19]). Secondly, the reflection must be causally connected in that any reasoning or report must be accurate and any modification must be effective in the intended way.

1.3.1 Implementation meta-levels

Meta-levels are not necessarily reflective in the above sense. The simplest example is a program being "about" its data, e.g. if it contains error-checking code of the form: "if the data is not in form X then there is probably an error". An operating system is also a meta-level because it is "about" the running of programs. Other examples include interpreters for languages and rulebases.

These "implementation meta-levels" are excluded from this study because the focus of the research is on agent architectures. Since the architecture must be simulated initially, there will be additional unintended meta-levels (incorporating the simulation mechanism) which do not belong to the object of study. I therefore use the term "meta-level" to mean a component that is embedded within the architecture of an agent as an explicit reflective mechanism, and where the kind of reflection is "conceptual" (as in e.g. Ferber [9]). However, if an explicit meta-level has a similar function to a rule interpreter or scheduler, (e.g. resource management), it will be considered as part of the architecture. We can summarise by listing three types of meta-level:

1. Implementation, e.g. rule interpreter;
2. Control: e.g. conflict resolution rules;
3. Declarative: e.g. self-diagnostic rules.

For the purposes of this study, the difference between type (1) and the others is that (1) is *implicit* and hidden (like the foundations of a house) while (2) and (3) are *explicit*. A particular meta-level can belong to both types (2) and (3), e.g. a meta-reasoning mechanism can also have "modify" access in order to correct specific problems.

1.4 The Autonomy Problem

Most existing literature on reflective architectures is concerned with language interpreters and other implementation meta-levels, e.g. a Lisp interpreter written in Lisp. Examples include 3Lisp [32] and object oriented reflection [31]. I will call these systems reflective programming architectures. Their purpose is typically to make the internal operation of components inspectable and modifiable *by a user*, to enable more flexibility in implementation, experimentation with language extensions etc.

In contrast, I am addressing the problem of autonomy, which primarily requires that *the agent* has these types of access to its own operation for the purpose of survival and (if possible self-repair) in a hostile environment. In other words, the agent must succeed *without* user intervention and this makes a significant difference to the kind of architecture required.

This is not to say that there is never any overlap between autonomy and reflective programming requirements. For example, it may sometimes be better for an agent to wait for user intervention instead of attempting to recover autonomously (deciding when to do this is the theme of a new research area called "adjustable autonomy"). Furthermore, some problems in reflective programming may be relevant to autonomous agent architectures (e.g. removing infinite regress).

It should also be noted that we only ignore implementation meta-levels during conceptual exploration of designs. If the architecture is to work in the real world, then these meta-levels must also be considered as part of its embodiment (i.e. hardware and any control software in which the agent software is embedded).

1.5 Reflection as Self-Observation

This research focuses on reflection in the context of *anomaly detection*, i.e. the autonomous system must detect any attack or intrusion involving critical parts of its own operation. An anomaly is defined as any significant discrepancy between an expected state and an actual state. To detect such irregularities, the system must have an expectancy about its next state and sensors to observe its actual state. Ideally, it should observe its own operation in the same way that it observes processes in the outside world. This means that the observing meta-level should be a concurrent thread which monitors the object level.

Examples of reflection as self-monitoring in a hostile environment include immunity-based approaches [3] and model-based diagnosis systems [27]. Work on more general execution monitoring and anomaly-detection has been done by De Giacomo et. al. [6], using a logic-based approach.

1.6 Sufficient Reflective Coverage

We can now define the problem which must be overcome: In existing self-monitoring architectures, certain kinds of anomaly can escape detection by the system itself, namely those which affect its anomaly-detection mechanisms. In other words, there is always a meta-level that is not simultaneously an object level. We call this the reflective blindness problem.

To address the deficit, I propose an architecture where the reflection is *distributed*. A distributed reflective architecture is a multi-agent system where each agent acts like a meta-level for the others. Thus everything that is a meta-level is simultaneously an object-level. This concept is an interpretation of Maturana and Varela's *organizational closure* of a network ([21], [34]) which is central to their theory of living systems. (I will subsequently use the term "meta-level closure"). The concept is also the foundation of "Second Order Cybernetics" (see e.g. von Foerster [35]). From the AI point of view, Minsky [24] has also discussed this issue in detail.

It should be noted that multi-agent distributed reflection may exist *without* meta-level closure. The aim is to investigate whether meta-level closure can satisfy the requirements more adequately than a non-closed architecture. Thus the effectiveness of different multi-agent architectures will be compared.

To explain what "effectiveness" means, the problem to be solved by a candidate architecture should now be stated more precisely. I am interpreting the requirement for self/nonself distinction as "sufficient" reflective coverage. This means that anomalies should be detected in any critical system operation. We can define a "critical" operation as one which is necessary to ensure that minimal user requirements are not violated (e.g. data inregrity). This is not the same as "complete" coverage which is clearly unattainable since it would require sensors that can detect all possible events (something which cannot be defined for the real world). Instead, self-monitoring resources should be focussed on the system's most critical management components.

A reasonable formulation of the the distributed reflection problem can now be given as follows: To what extent can an architecture detect anomalies in its own critical operations, given the following assumptions:

1. a physical event or a deliberate attack can only cause damage to one agent; i.e. more than one agent cannot fail simultaneously as a result of the one event, but more than one agent can fail sequentially as a result of separate events (e.g. hacker interference with one agent followed by a random hardware fault of another agent).

2. the time between successive failures or attacks is sufficient to allow detection and recovery.

3. the sensors can detect the type of change caused by the intrusion or failure.

These assumptions are reasonable because they also apply to biological systems. E.g. an immune system does not protect us from radiation or explosions. It does have the ability to distinguish between self and nonself but *with a certain response time* and *to a sufficient degree of accuracy*; it does not provide invulnerability. This means that the degree of solving the problem should be measured in terms of probabilities: i.e. what is the probability of detecting and recovering from a fault, given the above assumptions?

Furthermore, the degree of *discrimination* between a critical fault (e.g. in the controlling software) and a non-critical fault (e.g. in some rarely used utility software) is also important. In other words, the problem is solved well if the probability of detecting a fault in a critical management component is *higher* than that of detecting unimportant faults.

2 Limits of Existing Approaches

To show in detail the nature of the reflective blindness problem, and why existing methods do not solve it, I use artificial immune systems (AIS) as an example. A survey of immune-system inspired models can be found in [3]. Typically a database of "normal" patterns is used to define "self", which is collected in advance by running the system in isolation (in a "protected" environment). This database contains characteristic patterns produced by the execution of all legitimate programs and may be called the "signature" of the system being protected. Clearly, this has a statistical nature since there will be many isolated or rare patterns (e.g. a compiler with a non-typical set of directives or the processing of an unusual set of commands). There are many possible forms of recording such activity, e.g. file access patterns, system call sequences. In the "real" environment, the immune system continually compares actual patterns produced by currently active programs with the system's signature. If there is any significant deviation, a "nonself" has been detected. For details, including the definition of "significant", see [4].

To distinguish between self and nonself, an algorithm must ensure that the comparison between sets of patterns is carried out in the intended way. We call this algorithm the meta-level **R**. To my knowledge, existing AIS always have such a meta-algorithm. Even if the architecture is based on distributed pattern detectors e.g. [7]), a meta-algorithm must "manage" the detectors and ensure that recognition of nonself occurs as intended.

But if the underlying algorithm **R** of the anomaly-detection process behaves anomalously as a result of an intrusion, there is nothing that can sense this state in current AIS systems.

Simply including the pattern produced by the meta-algorithm **R** within the signature does not solve the problem, because this assumes that **R** will always be intact. If **R** becomes compromised then the whole immune system becomes unreliable. Indeed, **R** could be replaced by a "disinformation" algorithm which raises false alarms and covers up real anomalies, while a user may have the impression that the system works normally.

Clearly we cannot just add another meta-level to monitor **R** since this would lead to an infinite regress. It follows that there is always a significant part of the system that remains vulnerable to attack. This weakness has been called the "blind spot" or "reflective residue" by some in the historical cybernetics community e.g. Kaehr [14], although this has mostly been in a philosophical context. The notion of "distributed reflective architecture" is inspired by their work.

Before defining such an architecture in detail, it is first necessary to consider existing approaches which may address the same issues from different angles and show why they do not solve the problem.

2.1 Virtual Infinite Towers

Much work has been done in the elimination of infinite regress in the area of reflective programming, where the meta-levels are of the "implementation" kind. It is important to ask if they can be applied to a self-monitoring architecture where the meta-levels must detect anomalies in their object levels.

The most well-known programming architecture for avoiding an infinite regress is 3Lisp (Smith, [32]). The idea is to provide an environment which behaves *as if* there were an infinite tower of implementation meta-levels (in this case language interpreters). A typical application is the exploration of possible language extensions. If a developer is inspecting a program at a base level R_0, and it is being interpreted at meta-level R_1, it is possible to modify its interpreter, so that the program behaves differently or makes use of some new features. In this case a new meta-level R_2 is "spawned" in order to make the details of R_1 available for modification. The user may wish to repeat the process with the interpreter at level R_2 and so on. The term "interpreter" is actually very general and applies to any program which specifies how its object program is to be executed (e.g. what logical operators are available and what are their effects).

This means that any meta-level can be inspected on demand, giving the appearance that the reflective blindness problem is overcome. The most interesting language in this respect is RbCl (Reflective based Concurrent language) [13], which allows user modification of *all* aspects of every meta-level, up to the restrictions imposed by the operating system and hardware. What is normally a fixed kernel in systems such as 3Lisp is made into a user-modifiable structure, meaning that the language mechanisms such as memory management and reflective mechanisms can also be changed.

To apply this to autonomous agent architectures, the *user's* privilege of inspection and modification of meta-levels must be transferred to the agent. For example, the SOAR architecture may be described as a potential infinite tower of implementation meta-levels (Rosenbloom et. al. [28]). The first (or "base") level contains the initial description of the problem in terms of a state space and operators for changing from one state to another (e.g. move right). The first meta-level contains mechanisms which implement the operators (e.g. how to move right) using a production system and mechanisms for selecting operators (preference).

An "impasse" may occur either in the selection of an operator (e.g. two operators are equally preferred) or in an operator application (e.g. impossible to determine the state resulting from "move-right"). Then a subgoal is generated and a search takes places in a newly generated problem space on the first meta-level (e.g. find an alternative way of moving right which produces a desirable state).

During the search on the first meta-level, a second impasse can occur which results in a new subgoal being generated and a search through a problem space on the second meta-level. If there is not enough knowledge available to generate useful problem spaces, impasses may occur repeatedly, producing an infinite regress of meta-level activations (in the form of repeated recursive calls). To prevent this situation, SOAR has a mechanism for detecting when there is insufficient knowledge available to proceed further, which results in it reverting to a "default" behaviour.

Thus, the problem of giving an agent self-reflection can apparently be solved provided that there is some mechanism to prevent an infinite recursion of successive meta-level unfoldings (i.e. there must be some way of "bottoming out" the infinite regress).

2.1.1 Virtual Infinite Towers are Sequential

These virtual infinite tower systems do not satisfy our requirement because there is always a "current" meta-level which is not simultaneously an object level. Although any meta-level R_i in the infinite tower can be inspected on demand, another meta-level R_{i+1} must be activated to do this. But at that point, R_{i+1} is *not* being inspected by anything within the system. It only has the potential to be inspected. In other words, there is only a *sequential* role-switching between meta-level and object level.

While a SOAR type agent is modifying a particular meta-level (e.g. it is searching for a new implementation of an operator on the base level), the agent itself will *not* be able to detect any interference with the meta-level above it, i.e. the level that is doing the modification. Therefore we need a concurrent model, where meta- and object levels simultaneously coexist.

I do not know of any reflective programming systems which provide the required kind of concurrency. Where concurrency is present, it tends to be entirely *within* a single meta-level, An example is [22] where the object which comprises a meta-level is not a single thread but is instead a "group" of processes. At each level, the concurrency pattern in this group is preserved (i.e. each member of the meta-level group R_{i+1} is an implementation meta-level for each member of R_i).

However, the desired form of concurrency would probably have limited usefulness for typical reflective programming applications and may only introduce difficulty, e.g. modification access to an object level becomes a non-trivial problem (see later).

2.2 Self-Monitoring Meta-levels

An architecture which may allow a meta-level to run concurrently with its object level is Kornman's SADE [17]. This is not a virtual infinite tower but instead incorporates a meta-level (of the "declarative" type) for monitoring patterns of execution of an object level (in this case rule firing).

SADE is divided into three levels: a base level (M0), a meta-level (M1) and a meta-meta-level (M2). The task of M1 is to detect possible loops in M0 and to correct them. M1 detects a loop as a particular pattern of rule firings. It then interrupts M0 and selects a "loop repair remedy" (e.g. modify the situation that brings it about). However, if M1 is not successful in repairing M0's loop, it may go into a loop itself. E.g. it may repeatedly select the same unsuccessful remedy. But the possible ways in which looping can occur in M1 are known. Therefore the second meta-level M2 can then be designed so that it can always detect and repair M1's loops without going into a loop itself. Thus an infinite regress of monitoring levels is avoided.

This is much closer to the sort of architecture we are looking for, since it may detect intrusions or deliberate damage. Its two limitations are as follows: First, it only works for detection of known failure patterns; it cannot be used to detect anomalies because an anomaly occurs when something is *not* a known pattern.

Secondly, although the different levels can run concurrently, the architecture is "open" in that it not provide for the monitoring of M2.

2.3 Organizational Closure Model

A model which incorporates the required relationship between meta- and object levels is that of Kaehr and Mahler [15]. They attempt to formalise the autopoiesis concept of organizational closure using a somewhat obscure notation invented by the late German philosopher Gotthard Günther [11], [12]. The idea is to provide a framework for multiple parallel descriptions of the world, all of which are true in their own way, but which would result in contradictions if they were fused within a single description. (Although Günther's notation is sometimes called a "logic", it cannot be understood as a logic in the AI sense of fuzzy or modal logics).

As regards computational interpretation, a key idea is that something which is an operator may also simultaneously be an operand, i.e. a process P2 operates on an independently running process P1 which is simultaneously operating on another process P0, forming a kind of reflective tower, with the difference that each meta-level is a concurrent thread. "Operating" may include code modification (not just passive reflection). This would correspond to an interpretation of Kornman's architecture where the meta-levels run concurrently. Organizational closure is then implemented by making the code for process P2 the same physical thing as the code for P0.

As it stands, Kaehr and Mahler's model does not fit into the framework of autonomous agents. It is merely a "chaotic" model of computation written in ML. However, their overall idea is one of the main inspirations for the distributed reflection concept described here.

The closest practical concept to that of distributed reflection is that of a "society" of agents, where agents simultaneously observe each other (although they do not modify each other's code). We therefore look at current approaches to distributed control and diagnosis.

2.4 Decentralised Systems

Decentralising the control of an agent is normally not done for the purpose of improving reflective coverage. Advantages of decentralisation mentioned in the literature usually relate to issues such as agent specialisation and teamwork (e.g. [18]) or resource management and load balancing (e.g. [23]), although they also mention fault-tolerance in a more general sense.

The most interesting approach is called "social diagnosis" (Kaminka and Tambe [16]) based on agent tracking (Tambe and Rosenbloom [33]). The idea is that agents observe other agents' actions and infer their beliefs to compensate for deficiencies in their own sensors. E.g. if an agent is observed to swerve, it can be inferred that something exists which it wishes to avoid, such as a hole in the road. In particular, an agent may discover a fault in its own operation by observing the reaction of other agents.

However, for our problem, such models are unnecessarily restricted by the "society" metaphor (although it is useful up to a point). In a society, the internal operation of agents are not included in the observed world. It follows that such a multi-agent network can only be a distributed reflective network in a very weak sense (only insofar as an agent A_1 can detect faults in its own external behaviour by observing A_2's external reaction, hence the term "social diagnosis"). There is however the possibility of introducing multiple perspectives into such a model (i.e. different representations of the world), where one perspective may include features that are unknown in others.

Pell et. al. [27] address the issue of multi-perspective fault diagnosis of *internal* components of an autonomous spacecraft, but they include only the monitoring of hardware in their present model.

The following sections suggest a method for overcoming limitations of these models, while making use of some of their features.

3 Distributed Reflective Architectures

The aim is to use distributed reflection to provide coverage of those components which are the most critical to the operation of the whole system, thus compensating for the most serious reflective residues. In order to explore designs for distributed reflection, we require some working definitions.

We will say that a *current description* of an entity X is a combination of two things: the expected state of X according to a *model* of X's behaviour, along with X's *actual* state as shown by sensors. (We shorten this to "description" if there is no ambiguity). If X is the agent making the description, the description of X becomes a *current self-description.*

In a distributed reflective architecture, we can say that the self-description of a network is distributed over multiple "perspectives". To illustrate this, we consider two agents A_1 and A_2 which should be each other's meta-level. Then the network self-description from the perspective of A_1 is A_1's description (model predictions and sensor values) of A_2 and vice versa. In other words A_1 has a model of A_2 and sensor access to patterns caused by A_2's actual activity, thus enabling it to detect anomalies in its behaviour.

3.1 A Generic Agent Architecture: RML

We now define a "node" of the distributed network. This is a generic agent architecture and is inspired by the concept of "meta-management" (Beaudoin, [2], Sloman [29] and the anticipatory agent architecture of Ekdahl et. al. [8]. The agent may optimise some feature of the world or it may simply be homeostatic (ensuring that the world is maintained in an acceptable state). The architecture may be divided into four layers:

R: Reflective mechanism (meta-level)
M: Model-driven reasoning mechanism (in particular anticipation)
L: Low-level processing (e.g. low-level perception) and "instinctive" reactions.
S, E: Sensors and effectors.

L may be regarded as a reactive layer. I use the term "reactive" here to mean the kind of processing which is sensor-driven and where the result is immediately available (i.e. no search required). The result may be an external action or an update to memory. For example, evaluating the quality of something can be reactive (E.g. a sensory impression may produce a fearful anticipation but no external reaction).

M may be regarded as "deliberative" in that it makes predictions based on an internal model. The model may be a set of inference rules, possibly with certainty values (e.g. if a dark cloud appears then it will probably lead to rain). We will use the simplest form of this architecture where M is used for anticipation only, and not for planning or decision-making. This means that all actions are either determined within the reactive layer as "routine" actions or they are "emergency" actions taken by R in the event of an anomaly associated with danger (alarms - see Sloman [30]).

The reflective layer R compares sensor values with model-predicted states and takes action in the event of an anomaly (e.g. focus attention on the problem to get more information, and re-direct the operation of M and L and S/E).

Interaction between the three layers is shown very schematically in figure 1 (a). Sensors and effectors are omitted for space reasons. The large arrow labelled "1" indicates the relation "meta-level for ..", the small arrow labelled "2" indicates model-driven effects on the reactive level. E.g. the anticipation of being late causes an agent to move faster instinctively. The small arrow labelled "3" indicates "data-driven" processing when the *actual* current state is used to calculate the next state, since it may involve small refinements to the predicted current state (which are usually non-anomalous). If we suppress this arrow, we get "simulation" (i.e. we use the predicted next state as if it were the actual state).

3.1.1 Implementation Levels

There are actually three "micro-levels" within M. Level 1 is the implementation of the reasoning mechanism, e.g. a rule interpreter. Level 2 is the model itself, which is interpreted by the level 1 mechanism and level 3 is the actual *prediction* made by the model at any given instant. Thus if the model is "run" without taking account of sensors we have a sequence of predicted states. In accordance with 1.3 we ignore level 1 at this stage. I assume that it is reasonable do this for exploring conceptual designs (for an argument in support of this e.g. Sloman [30]). This means that M can be viewed as *being* the model instead of something that "applies" it. The agent's current description of an object is then a combination of the model-predicted next state and the actual sensor observations on the object.

3.1.2 Single-Agent Reflection

We can define the degree of single-agent reflection by considering what to include in its current description. This reflection is *minimal* if the agent's description includes the external world only. Then R simply compares model predictions about the world with sensor values. Reflection may take the form of "diagnosing" where the model of the world has failed. But since the agent has no explicit model of the "correct" behaviour of M, it cannot detect anomalies in its own reasoning mechanism, but only in the world (e.g. many dark clouds are seen which do not lead to rain). Note that an anomaly in the world will mean that any "repair" action will operate in reverse (i.e. the model should normally be changed and not the world), while normally a model of the "correct" behaviour of an internal component is the way we *want* it to behave, and any deviation means that *it* should be repaired and not the model.

Single-agent reflection is *non-minimal* if its description of the world also includes some aspects of M, L or S/E. In other words, the world for the agent includes not only the external world but also its own operation, including the status of its sensors and effectors. Then R can detect an anomaly in the operation of the other components by comparing their actual performance with their predicted behaviour. For example, it is possible that the knowledge encoded in the model was correct but the inference process was faulty; e.g. there may be an additional rule stating that dark clouds seen from a distance often do not lead to rain, but because of an unauthorised modification of a condition, this rule never fires (there may also be a software fault in the interpreter but we exclude this problem since it belongs to level 1). Similarly, if the sensors are faulty, a cloud may be perceived as dark when it is not. Ways of introducing non-minimal reflection will be introduced later.

3.2 Exploring Design Space

We now specify how to get from a single node to the desired multi-agent architecture as a series of incremental transformations. The "broad and shallow" approach is used to explore possible overall designs [29]. The starting point is the generic agent architecture which we call D_0 (for first design) followed by D_1, D_2, .. etc. The sequence may be called a trajectory in design space. From D_1 onwards it becomes a multi-agent system where each participating agent is a variant of D_0.

3.2.1 Minimal Solution: Mutually Observing Agents

The first stage D_1 is composed of two agents A_1 and A_2 and there is a task T to be achieved by A_2. For example, T may be file management. Then, if the file manager behaves anomalously (e.g. it may allow deletion of a file by someone other than the owner) the unusual activity can be detected by A_1. A_1 is then a meta-level for A_2. But additionally A_2 should independently monitor A_1, since A_1 itself may behave anomalously. D_1 may be specified as follows:

A_1 (Meta-agent): checks that the actual activity of A_2 does not deviate from its normal activity.

A_2 (Object-agent): carries out T **and checks that the actual activity of the anomaly-detection program A_1 does not deviate from its normal pattern**.

If we look closely, this architecture has the same form as D_0 with the addition of the second function of A_2 in bold face, which enables the object level to become simultaneously a meta-meta-level, i.e it "closes" the network. In D_0, this network effectively "collapses" to a single agent with meta-level A_1 and object level A_2 (where A_1 corresponds to R, A_2 corresponds to $M/L/S/E$ and R has a non-minimal reflective capability as defined above). In D_1, however, the terms "meta" and "object" are rough guidelines only and do not have the crisp definitions they have in D_0. Note that D_1 requires that both agents monitor each other's anomaly-checking activity. We will show later how this can be implemented.

3.2.2 Requirements-based Architecture

One serious problem with D_1 is that *any* anomalous pattern will be treated as hostile, resulting in "allergic" reactions if new software is introduced (using the immune system metaphor). The next stage (D_2) looks as follows:

A_1 (meta-agent): checks if the operation carried out by A_2 satisfies its requirements (i.e. does it continue to fulfil task T satisfactorily and does it check for anomalies satisfactorily?

A_2 (object-agent): carries out T **and checks if the anomaly-detection program A_1 satisfies its requirements**.

To be rejected as nonself, a pattern anomaly must also be associated with an undesirable or dangerous situation, which is defined here as a sudden non-fulfilment of the critical requirements. Determining whether requirements are met or not requires a utility function which will be defined later when we introduce a specific scenario.

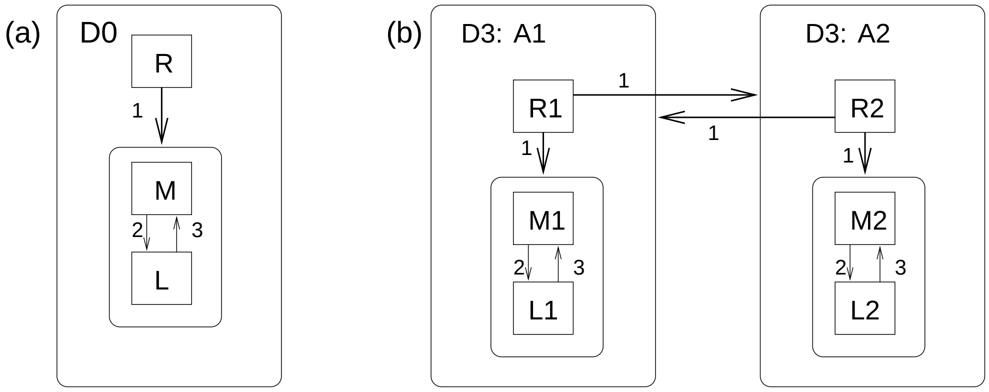

Figure 1: Transition from hierarchical to distributed reflection

3.2.3 Load-Balancing Architecture

If the load is balanced between the two agents, D_2 is transformed into the following specification (D_3):

A_1: Meta/object-agent: carries out T_1 and checks if the operation carried out by A_2 satisfies its requirements (i.e. does it continue to fulfil task T_2 satisfactorily and does it check for anomalies satisfactorily?)

A_2: Meta/object-agent: carries out T_2 and checks if if the operation carried out by A_1 satisfies its requirements (i.e. does it continue to fulfil task T_1 satisfactorily and does it check for anomalies satisfactorily?)

Now the architecture is symmetrical and T is subdivided into subtasks T_1 and T_2. A schematic diagram of the transition between an D_0 type architecture and an D_n architecture ($n > 0$) is shown in figure 1. (a) shows the generic agent architecture; (b) shows a symmetrical version of distributed reflection similar to D_3. Each agent in D_3 is a version of D_0. An agent's meta-level R now has the function of monitoring the other agent in addition to its monitoring of its own operation and of the external world. (Interfaces to the external world are not shown here for space reasons). The design space we have explored is summarised in figure 2, which also shows the design trajectory from D_1 to D_3. Another design (D_4) is shown which would correspond to a load-balanced architecture with pattern-based anomaly-detection only.

Note that D_n ($n > 0$) satisfies our definition of meta-level closure, but it becomes open if we remove one of the thick arrows between A_1 and A_2 (there is now a meta-level which is not an object level).

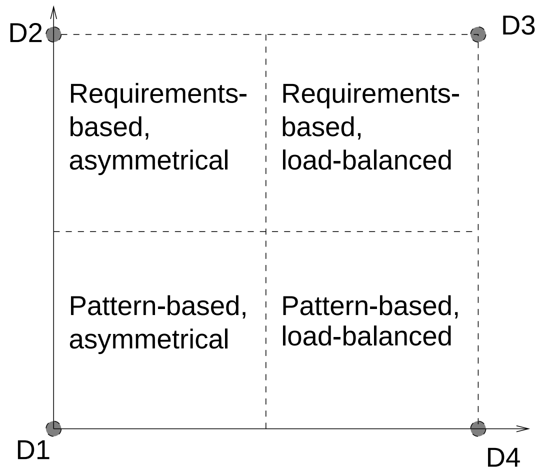

Figure 2: A simple design space

4 An Example Scenario

The aim is to approximate the design D_3 (in the upper right quadrant of figure 2). To define more precisely *how* agents can be each other's meta-level, we specify a scenario to simulate the kind of interaction we require. We do not completely eliminate global meta-levels at this stage, since we are relying on a simulator to ensure that the agents work as intended. But we use it as "scaffolding" to build the desired kind of system on the virtual level.

We use a modification of the "nursemaid" scenario [36] originally designed to simulate motivation and emotional states in a nursemaid which is taking care of a number of babies. The reason for selecting this scenario is that it is suitable for exploration of architectures where agents should represent the "values" of users; i.e. the agents' autonomy should be centred around those things that the user is most *concerned* about. One can say that the agents should be "concerned" *on behalf of* users. I will return to this later when considering design constraints for reflective coverage.

In contrast to the original minder scenario, we are simulating a collective "self-minder", where the babies also participate (albeit in an unusual way) in the "minding" process (there is another collective minder architecture which uses a different concept [5]). We call our scenario "minder3". We start with two agents only. Each agent is a specialisation of the above-defined generic architecture D_0 (i.e. A_1 becomes the nursemaid and A_2 becomes the baby). Their tasks (defined below) are homeostatic and do not involve any scheduling or planning at present.

We intend to simulate a distributed control system. Therefore the agents are components of a larger body which is moved around the virtual world as a single entity. For convenience, we call this the "vehicle" (although applications are not restricted to mobile vehicles or spatial worlds - see below). The agent software is embedded into this (simulated) hardware. For our purposes, the relevant hardware components are the sensors and effectors. The internal components of the vehicle (hardware and software) are called the "network". This includes the software of both agents, the separate processors on which they run and internal interconnections between them. (Alternatively we can think of the nursemaid always accompanying the baby).

It is assumed that there is a human user whom the agents are acting on behalf of (i.e. the tasks represent the user's goals) and that any *detection* of an anomaly is accompanied by a *report* to the user.

4.1 External World

The homeostatic aspect of each agent is simulated using a 2D virtual world which contains the following:

- A static energy supply (similar to the battery charging point of the original minder scenario),
- a ditch, which the vehicle might fall into
- various treasure stores near the ditch which the baby finds interesting.

The baby's task is to find as much treasure as possible in order to maintain its level of interest, which falls rapidly in the absence of anything new. The baby seeks out the treasure stores and collects treasure from them (although in practice this only means that it remains stationary until its interest level falls off). The nursemaid's task is to maintain the energy level and safety of the whole vehicle. Normally the baby has control of the vehicle but it may be taken over by the nursemaid at any time.

Depending on the amount of redundancy built into the design, an agent can take over another's task in a failure situation. However, the ability to do this is limited since the nursemaid is intended as a "specialist" in energy and safety while the baby is a "specialist" in treasure.

In addition to these primary tasks, the agents should detect anomalies in the external world and in each other's activity. In an enhanced version of this scenario, the treasure stores could appear spontaneously and decay after a certain time period. A snapshot of the external world is shown in figure 3.

4.1.1 Enemies

Enemies are agents which have destructive effects on the baby or nursemaid. The following are different forms of destructive interference:

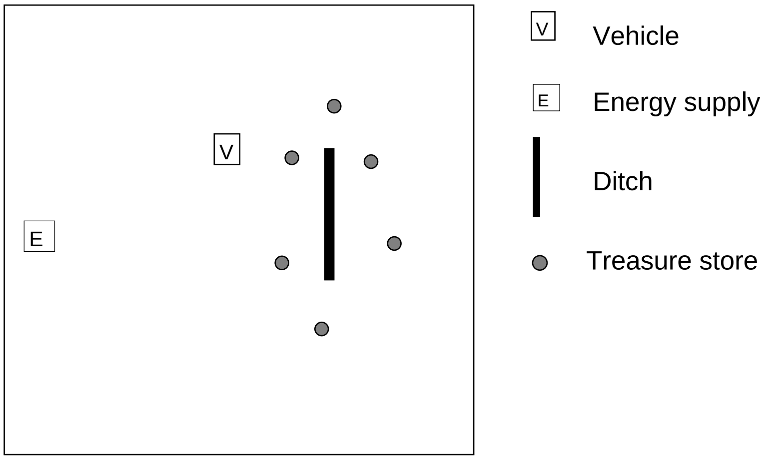

Figure 3: The modified nursemaid scenario.

1. **Direct damage**: damage to any of an agent's software components (controlling its sensing, acting, decision-making, perception etc.) by deleting, corrupting or otherwise modifying its code (e.g. by inserting a virus).

2. **Weakness exploitation**: present the agent with a situation that its software cannot cope with; i.e. give it an "unsolvable" problem that causes it to behave stupidly, crash or go into a loop. E.g. an enemy moves the treasure further away as soon as the agent gets near it.

3. **Resource blocking**: prevent the agent from achieving its goal by stealing, blocking or diverting its resources. E.g. continually distract the nursemaid with anomalies which are apparently associated with "dangerous" states and must be investigated immediately.

If the baby's control system is interfered with, the nursemaid should detect this and try to repair it.

An "enemy" can also be a random event in the environment: e.g. the ground around the ditch occasionally becomes slippery.

4.2 Meta-level Closure

In accordance with D_3, the nursemaid/baby relationship must be non-hierarchical and therefore we give the baby the ability to independently detect anomalies in the control software of the nursemaid. The baby is therefore "gifted" (actually "unevenly gifted" as we shall see later). If we refer back to the immune system metaphor, an enemy will leave an anomalous pattern which is collectively recognised as "nonself". In this scenario, "collective" means that either the nursemaid or baby does the recognition (or both simultaneously).

4.2.1 Observing an observation process

One of the most important requirements of D_3 is that both agents must detect anomalies in their respective anomaly-detection software. This may be called "observation of observation" or "meta-observation". Furthermore, the observation on both sides must be *independent*, in the sense that agents do not use the same method of observation. The preliminary solution adopted here is in two parts:

(1) monitor *patterns* associated with the anomaly-checking activity using signature techniques similar to that already outlined for immune-system models.

(2) couple this with *quality* of the agent's actions in the "external" world according to the utility function.

Thus a change in *pattern* does not necessarily indicate problems unless it is coupled with a change in external behaviour which looks "worrying". E.g. if the nursemaid's control software has been modified and produces an anomalous trace, it may still continue to maintain the energy level. It would look problematic only if the nursemaid ceased to act at all or acted in a destructive manner.

In the remainder of this report, I will use the following convention: sensors which monitor the state of the virtual world will be called "external", while those concerned with program signature patterns (dynamic or static) associated with the control system of an agent are called "internal". Each agent has a unique description of both the external world and the internal state of the whole network itself.

Meta-observation requires an internal world and internal sensors. An architecture of type D_3 with those components is shown schematically in figure 4. The external world is labelled "EW". The internal world ("IW") is divided into nursemaid (N) and baby (B) and includes all their hardware and software. Their different components S_E (external sensors), E_E (external effectors), S_I (internal sensors) and E_I (internal effectors, e.g. error-recovery) are assumed to be low-level software in which the high-level agent software C (for control) is embedded. The IW is actually embedded within the vehicle which is in turn embedded in the EW. The

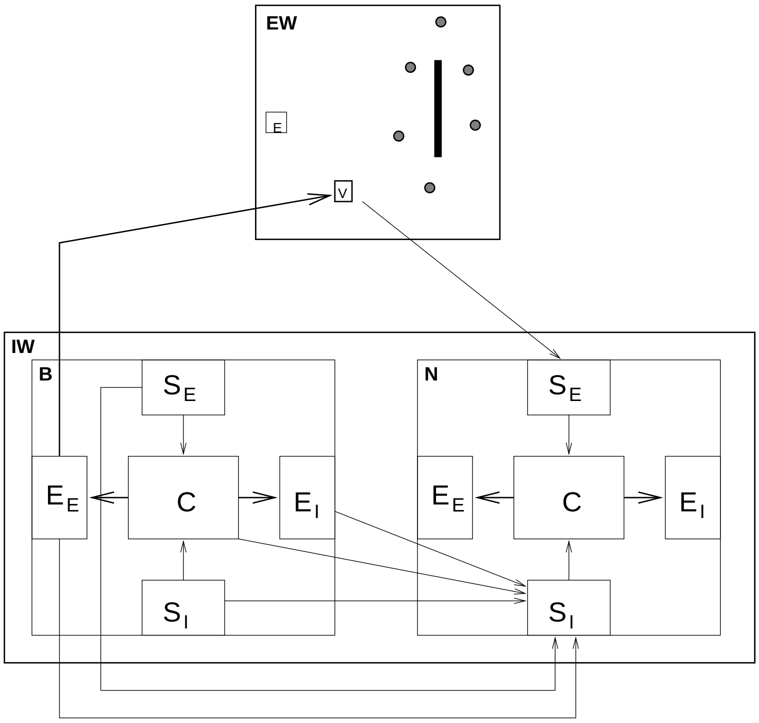

Figure 4: Internal and external interfaces

meanings of the term "sensor" or "effector" include not only the boxes in the diagram but also their input and output connections shown by arrows. Incoming arrows to a control system are sensor values while outgoing arrows are effector activations. The arrows to/from the control system typically contain more abstract data while those to/from the environment will have a more physical nature but we need not be concerned with those issues. For space reasons, only the set of connections to A_2's internal sensors are shown.

4.2.2 Concurrency

If agents are to act as each other's meta-levels they should monitor and evaluate each other *independently* and not according to any centrally coordinated method. The observing agent and the observed agent should run concurrently. This is similar to observation in nature, e.g. measuring room temperature does not involve "interrupting" the thing being measured. Nor does it involve waiting for it to reach a certain state (although one might wait for the measuring device to stabilise).

4.2.3 Notation

For any agent A_i, its world model may be written $M(i)$ and its sensors $S(i)$. Since we are ignoring implementation (such as a rule interpreter) $M(i)$ is identical to the layer M of the generic architecture for A_i. In general, we use the following convention: predictions made by the model about an object X are written $M(i, X)$ which could be read as "$M(i)$ applied to X". For example, $M(i, W)$ means "predicted next state of external world". In contrast $M(i)$ is the set of rules, declarations or other representation which is the *mechanism* of prediction. The subset of those rules that make predictions about X may be labelled $M(i).X$.

The same applies to sensors: if A_i uses a sensor S to observe some feature of an object X, (e.g. its temperature) then the *values* produced by the sensor is indicated by $S(i, X)$, while the *operation* of the sensor is written $S(i)$. The former is equivalent to asking "what's the temperature?" while the latter means "how does the thermometer work"? If A_i is a generic agent architecture, its reflective layer R detects discrepancies between $M(i, X)$ and $S(i, X)$.

We also use the following convention: in any mutually observing agent pair, the subscript i indicates the *observing* agent and the subscript j is used for the *observed* agent. This increases clarity later when discussing implementation.

A_i's model and sensor observations of A_j may be written as $M(i, j)$ and $S(i, j)$ respectively and each of these in turn may be further subdivided into internal and external partitions. The partition $M_I(i, j)$ is an expectancy about the internal operation of A_j while $M_E(i, j)$ is an expectancy about A_j's effects on the environment (e.g. how will it change position). A_i's prediction of the internal state of A_j is further partitioned into predictions about its different components. All partitions are summarised in the Tables 1-4 overleaf.

Table 1: External model partitions:

Partition of $M_E(i, W)$	object being modelled
$M_E(i, j)$	external actions of A_j
$M_E(i, U)$	inanimate objects

Table 2: Internal model of A_j:

Partition of $M_I(i, j)$	object being modelled
$M_I(i, C(j))$	control system of A_j
$M_I(i, S_I(j))$	internal sensors of A_j
$M_I(i, S_E(j))$	external sensors of A_j
$M_I(i, E_I(j))$	internal effectors of A_j
$M_I(i, E_E(j))$	external effectors of A_j

Table 3: External sensors:

Partition of $S_E(i, W)$	object being observed
$S_E(i, j)$	external actions of A_j
$S_E(i, U)$	inanimate objects

Table 4: Internal observations of A_j:

Partition of $S_I(i, j)$	object being observed
$S_I(i, C(j))$	control system of A_j
$S_I(i, S_I(j))$	internal sensors of A_j
$S_I(i, S_E(j))$	external sensors of A_j
$S_I(i, E_I(j))$	internal effectors of A_j
$S_I(i, E_E(j))$	external effectors of A_j

Notes:
(1) $C(j)$ is the control system of A_j. This includes the layers R, M and L of A_j.

(2) We assume that the *only* model of inanimate objects (e.g. rain) is external and this means that $M_E(i, U)$ is identical to $M(i, U)$ and $M_E(i, W)$ is identical to $M(i, W)$ where the latter includes external effects of agents. ("U" means "*un*interested" because these objects have no motives or interests.)

(3) When sensors are being modelled or observed, the *pattern of their operation* is being studied and *not* their values (see notational distinction above). Otherwise there would be a problem with $S_I(i, S_I(j))$ which corresponds to the sensor used by A_i for meta-observation. If the two agents interact according to D_3 and we "evaluate" the $S_I(j)$ component, we get an apparent infinite regress:

$$S_I(i, S_I(j, S_I(i, S_I(j, ...))))$$

However, this need not occur in an implementation. If we apply our definition of independence above then there are two concurrent threads monitoring each other: $S_I(i, S_I(j))$ which is part of the execution cycle of A_i and $S_I(j, S_I(i))$ which is part of A_j (if we imagine for simplicity that each agent's "execution" is a sense-decide-act cycle). A_i's thread does not wait to evaluate the content of $S_I(j, ...)$, as this would lead to a deadlock situation. Instead A_i monitors the *pattern of operation* of $S_I(j, ...)$. The result is A_i's specific way of describing $S_I(i)$. Later, I will outline how this can work in practice. For illustration we can imagine a thermometer T whose operation is monitored using a device D (e.g. by ensuring that there is mercury in it). But D can only work at a certain temperature, which in turn needs to be monitored using T.

(4) At this stage I have not explicitly included models and observations about A_j's *models*. This would be equivalent to asking: "does the mechanism for predicting the next state of X operate normally?". The mechanisms for those models are all included in $C(j)$. It is not necessary to partition this into different components at present, since they can all have the same form. Later we will see how the monitoring of R can be treated separately from that of M and L since R has a higher degree of "importance" (because it is a meta-level).

4.3 Non-minimal Single-Agent Reflection

It is important also to define single agent reflection so that it can be compared with distributed reflection. Single-agent reflection is non-minimal if the agent can detect anomalies in its own operation (see 3.3). It requires that the agent has a model of its own components and can observe them in operation, i.e. $M(i,i)$ and $S(i,i)$ exist and can be compared. They can be similarly partitioned into M_I and M_E, but some of them are redundant (see later).

$M_I(i,i)$ is encoded in the same ontology as $M_I(i,j)$ and together they form A_i's model of the "internal world". However, $M(i,W)$ is expected to be different because it is modelling different phenomena, such as the trajectory of an agent in the spatial world. We can say that an agent is *maximally* reflective if $M(i,i)$ and $S(i,i)$ contain all the information in $M(i,j)$ and $S(i,j)$ *except* that which would lead to infinite regress (example given later).

4.4 Assumptions and Special Constraints

For the scenario, the following are simplifying assumptions listed in approximate order of realism.

1. An anomaly in the internal world is assumed to be a failure, but an anomaly in the external world may be evaluated positively or negatively. E.g. if a ditch has some new interesting features.

2. Only one component can fail or be damaged within a specific time period. E.g. an enemy cannot attack the nursemaid and baby simultaneously. Components are those elements listed in the tables 1-4. E.g. an internal sensor is a component, and a control system is a component.

3. Either the nursemaid or the baby has "control" over the vehicle at any one time. In an emergency, one agent can disable the effectors of the other and take over control, e.g. the baby is in the danger area of the ditch and is still moving towards it. The problem of conflict resolution will not be addressed here; we simply assume that the disabled agent "submits". Due to the minimum time period between failures, the disabling agent cannot be damaged while it is taking over control, and there is therefore no reason to "question" its actions. (However, the kind of scenario that is ultimately desirable is a kind of cooperation or symbiosis between the two agents, e.g. cooperative repair).

4. The agents do not initiate any repair actions (except when this is trivial). Instead, a

suspected fault is reported to the user.

5. An enemy can only inflict *direct damage* on an agent's components (by deleting, corrupting or otherwise modifying its code), e.g. there is no deception or diversion of resources.

4.4.1 Minimal Scenario: Special Constraints

For the first stage of the implementation the following special constraints apply:

(1) *Inanimate objects are static*:
There is one ditch and a constant number of treasure objects. They are static and cannot be moved. They do not grow or decay spontaneously. The baby does not "collect" them but instead looks at them for several cycles, gets "bored" and proceeds to the next nearest object. This means that $M_E(i, U)$ is trivial.

(2) *All external objects are visible*:
The agents do not have a limited field of view; everything in the 2D grid is visible.

(3) *Damage to control system only*:
There can be no damage to sensors, effectors or internally stored data in the initial version.

(4) *Limited External Anomalies*:
The only external anomaly that can occur is that the ground around the ditch becomes more slippery than expected. This is also done by an enemy.

(5) *The effect of the ditch is predictable*:
If the vehicle falls into the ditch, exactly one part of it will be damaged. In the initial version this is restricted to the control system C of the baby or the nursemaid.

The first two constraints are not realistic for even a minimal simulation of a physically situated agent. However, the problem we are addressing here is independent of these considerations. Therefore these constraints may still apply in non-minimal versions.

The problem of meta-observing control systems is difficult and will be investigated on its own initially. However, anomaly-detection in sensors and internal data is fundamental to the problem of distributed reflection as a whole. Therefore constraint (3) should be lifted in non-minimal versions.

An enemy is simulated by a random number generator which determines when the baby or nursemaid should be attacked or when the ground around the ditch should be made slippery. These are the only dynamic things that happen in the environment. However, in later versions it is possible to design a third agent to act like an enemy and leave an "intruder trace".

4.5 Minimal Requirements for each Agent

We are now in a position to state the minimal requirements and the subjective quality of the world from the perspective of each agent. This is necessary for D_3 which is a requirements-based architecture.

The requirements for the minimal version are the homeostatic maintenance of energy level and level of interest.

A summary of each agent's requirements is in the following tables:

Table 5: Requirements of baby:

Condition	Action
NOT at treasure	move to nearest treasure
near ditch	move round it
at treasure AND NOT bored	collect treasure
at treasure AND bored	move to next treasure
in ditch	do nothing
damage to nursemaid	try to repair

Table 6: Requirements of nursemaid:

Condition	Action
energy level low	move to energy supply
too near ditch	move away from ditch
in ditch	pull out of ditch
damage to baby	repair
otherwise	do nothing

The nursemaid takes over control if the baby leads the vehicle into a ditch, or is dangerously close to the ditch. Both these situations can only happen in the event of an anomaly (the vehicle overshoots the expected position due to slippery ground). Pulling out of the ditch not only results in damage but also uses more energy. Although the nursemaid knows what to do in these situations, they are anomalous because they contradict its model of the environment.

4.5.1 Quality Evaluation

How does an agent A_i ensure that the other agent A_j satisfies the requirements for its task? To do this we can introduce the concept of *quality of the world from the perspective of* an agent. This is simply the task requirements of A_j as "seen" by the observing agent A_i. That is, A_i will conclude that A_j satisfies the requirements if the minimal quality of the world from A_i's perspective is not violated (although there may be some details of the task that do not affect the environment as seen from the observing agent's perspective).

If we assign the tasks T_1 and T_2 mentioned in D_3 to the baby (B) and nursemaid (N) respectively, then N evaluates B's performance of T_1 by its perception of "quality" of the world and conversely for B's evaluation of N.

While N is observing the way that B controls the vehicle, it concludes that the minimal requirements for B's task are violated if one of the following happens:

1. B remains stationary for more than t cycles (where t is much longer than the longest "observation" period). If this is accompanied by an internal pattern anomaly then N concludes that B's software has failed. Otherwise it reports a possible fault.

2. B is within the danger zone of the ditch. N takes over control of the vehicle (since it is a specialist in ditches) and reports a possible fault. If this is accompanied by an *internal*

pattern anomaly then N concludes that B's software has failed.

3. B is in the ditch and is now damaged (because of special constraint (4)).

In any of the situations where N concludes a definite failure, it takes over control of the vehicle and tries to repair the fault or damage. Taking over control in the event of certain failure means that it may also attempt to do some of B's work depending on the amount of redundancy we want to build into the system. Full redundancy would mean that N has the potential to do the job of B as well as B itself can and vice versa. The tasks would then be allocated according to the principle of load-balancing only. However, because we are assuming that the agents' software is developed independently according to different perspectives, the agents generally do not have the ability to do each other's tasks equally well (E.g. B is motivated by the need to maintain its "interest" level but N is not). However N can take over the treasure-collecting task if it finds that it cannot repair B (but in the present implementation, it can repair B - see later).

We now consider the situation from B's viewpoint. It detects that the minimal requirements are violated if one of the following happens:

1. At least u cycles have elapsed since N took over control for the energy supply task and the energy level still has not been recharged. If the energy level is becoming dangerously low or there is an internal pattern anomaly then N concludes that B's software has failed. Otherwise it just reports a possible failure.

2. N moves away from the energy supply before the vehicle is recharged. If this happens when the energy level is very low, or it is accompanied by an internal pattern anomaly then N concludes that B's software has failed. Otherwise it reports a possible failure.

3. The energy level has dropped to the danger level and N has not yet taken control. B takes over the task of N in this case and concludes there is a failure.

4. N moves the vehicle towards the ditch. If it moves into the danger zone or there is also an internal pattern anomaly, B takes over the task of N and reports a failure.

If B detects that N is damaged, it attempts to repair N. During its repair attempt, it is "cautious" i.e. it stops being interested in treasure, moves into the vicinity of the energy supply and stays there, replenishing its energy level as soon as it drops below the "high" level. This is because energy has the highest priority (both agents are dead without it), followed by safety and interest in that order. (In non-minimal versions, the degree of "caution" may be a variable parameter - for example, a more "adventurous" baby may still like to explore the treasure without the nursemaid). If B is successful in its repair of N it goes back to its normal state.

The occurrence of external anomalies may appear superfluous (why not just make "slippery ground" a known condition?). However, any dynamic change in the external world is anomalous in the initial scenario simply because the initial world model should be kept simple. This situation is very similar to that in real-world software where it may be better to design the system to deal with unforeseen situations rather than attempt to represent every possible failure in a model.

Moreover, the architecture should be "broad and shallow" and provide slots where future research can plug in "narrow and deep" components. One of those slots is the pre-programmed action "move out of ditch" which may be replaced with a search for a contingency plan (find a way to get out of the ditch). Another slot is the point where an anomaly is detected and reported; in addition to merely detecting the anomaly, the model of the environment could be revised using exploration-driven learning.

5 Implementation

I will now describe in detail how the minimal version is to be implemented. This is where only the control system of an agent can be damaged, and not its sensors and effectors.

5.1 The SIM-AGENT Environment

The SIM-AGENT environment provides generic object classes which may be specialised into agents or inanimate entities. Each agent class has its own database and rulebase (called a rulesystem) which is composed of modules known as rulesets (among other things). Agents and objects are represented graphically in a 2D world. A scheduler "runs" each agent by first updating the agent's sensors with the current state of the world, then using a rule interpreter to run its rulesystem, and finally updating the spatial world according to any external actions required by the rules. Normally each agent is run once in a single time-slice. during a single agent run, some of its rulesets may be run several times (ruleset cycles).

5.2 Implementation of an Agent Control System

The architecture of an agent A_i is its control system $C(i)$ along with external and internal interfaces as outlined in figure 4. The control system is implemented as a rulesystem. (Sensors and effectors will be discussed later). Each layer of the generic architecture is implemented by a partition of the rulesystem (one or more rulesets). The execution of the rulesystem can be divided into two phases A (initialisation) and B (normal). It is expected that rules for predicting the next external state will require a sequence of initial snapshots before a prediction can be made (exploratory implementations have shown this). During initialisation this history is collected, but otherwise the agents react as normal. They will simply not be able to detect an external anomaly during the initialisation phase. In each cycle of a phase, control is passed between the different architectural layers. The sequence of steps in a cycle looks as follows:

Stage(A) Initialisation: collect an initial history.
L: sense and interpret data;
R: check internal state against predictions;
M: predict next external state if enough information is accumulated;
L: act on environment as necessary;

Stage(B) Normal: next external state is predicted.
L: sense and interpret data;
R: check external and internal state against predictions;
M: predict next external state;
L: act on environment as necessary;

Both phases are effectively a sense-interpret-act cycle L with the addition of anomaly-detection R and anticipation M. "Emergency" actions initiated by R in the event of an anomaly are not shown for space reasons. The scheduler runs one cycle per agent. There is no concurrency *within* an agent rulesystem, but there is (coarse-grained) concurrency *between* agents which should be adequate for our purposes at present. One can think of each layer R, M, L as *distributed* over multiple perspectives and multiple concurrent threads. The predicted internal state is known in advance (defined later).

5.3 Implementation of Meta-Observation

The control system of an agent is its rulesystem. Meta-observation requires that the control system be monitored. This can be met using (at least) two mechanisms. The first (to be used by the nursemaid) is to have a copy of the static pattern (i.e. the code) of the rule system, as it should be (corresponding to $M_I(i, C(j))$) and regularly compare this with the actual state of the rulesystem $S_I(i, C(j))$. Note that in this case the model is "trivial" in that there are no prediction rules; the predicted next state is just "there" as a static value. Effectively the model mechanism $M_I(i)$ is an implicit rule which says: "the next state will be the same as the current state". However, the model will become more complex if we allow authorised modification of rules (e.g. during learning). A second method (to be used by the gifted baby) is to monitor the dynamic pattern associated with rule firing (to be defined later).

To determine which rules should be monitored (or whether they all should be), we have to subdivide the observed rulesystem $C(j)$ into subcomponents:

$R(j)$:
(a) $R_E(j)$: detect external anomalies: $R_E(j,j)$, $R_E(j,i)$, $R_E(j,U)$.
(b) $R_I(j)$: detect internal anomalies: $R_I(j,j)$, $R_I(j,i)$

$M(j)$:
(c) $M_E(j)$: make external predictions: $M_E(j,j)$, $M_E(j,i)$, $M_E(j,U)$.
(d) $M_I(j)$: make internal predictions: $M_I(j,j)$, $M_I(j,i)$.

(e) $L(j)$: all other processing in $C(j)$.

Many of these subcomponents are redundant as we will see later. Some of them are not really rules but segments of data which have the effect of implicit rules (such as the copy of the "correct" rulesystem, $M_I(j)$ as explained above. According to initial constraints the contents of an agent's database cannot be damaged. However, in later versions this should be taken into account, as it can be damaged by unauthorised modification, causing a non-existent anomaly to be detected (as the reality would be compared with an incorrect model). Such a problem can be detected by recording modification patterns of internal data, and should be simple in the case of something that should *not* be modified.

5.3.1 Selective Monitoring

In the case of static patterns, it is fairly simple to monitor the whole rulesystem. However, monitoring dynamic firing patterns for the whole rulesystem may be unnecessarily complex. We therefore require a design strategy.

To select the first design step, two principles are applied:
(1) identify the segment(s) of the rulesystem which are most *critical* to the operation of the system as a whole and include only the monitoring of this segment in the initial design. Additional monitoring is added only when necessary.
(2) the first design should be the simplest possible and should be expanded only when necessary, i.e. the simplest form of monitoring should be used initially.

Applying principle (1) leads to the choice of $R_I(j)$ as the most critical component, i.e. segment (b) of the rulesystem. This follows from the fact that $R_I(j)$ is the only component whose failure cannot be detected by $R_I(j)$ itself, even with maximal single-agent reflection. I.e. it is the reflective residue pointed out in section 1. (Unless $R_I(j)$ has a microstructure which is also a distributed reflective architecture, which is not he case in this implementation). The failure of any other component in the control system can in principle be detected by $R_I(j)$. For example, it may detect failures in its reactive layer L.

Thus a *minimal* solution will certainly require the monitoring of (b) and the initial design should only include this. Then it will become clear what other kinds of monitoring are also required.

5.3.2 Internal Meta-Observation

For the initial implementation, we start with the most critical problem, namely the internal monitoring of internal monitoring. This problem has the same form as the internal sensor problem mentioned earlier, except that we are modelling the control system (which uses the sensor). The partition of $C(j)$ we are interested in is its meta-level R_I. Using our notation, $R_I(i, X)$ means "R_I of A_i detects an anomaly in X" (using $M_I(i, X)$ and $S_I(i, X)$). Then internal meta-observation can be written as:

$R_I(i, R_I(j, R_I(i, ...)))$

which again appears to lead to an infinite regress, but as we have seen in section 4.2.3, the agents do not wait for the results of each other's anomaly-detection but run as concurrent threads, making independent observations of each other.

5.3.3 Rule Firing Patterns

We can now consider in detail how we can implement the monitoring of rule firing patterns, which was the second monitoring option. The meta-observing agent (in this case the gifted baby) will observe one of three classes of patterns. First, there are two patterns indicating "correct" activity: the recognition of a normal situation (which will happen most of the time) and the recognition of an anomaly (which will happen occasionally). We may label them as "no" and "yes" patterns respectively. The simplest way to do this is to observe the *decision* made at the end of the comparison procedure

In order to detect anomalies in the observed agent's anomaly-detection mechanism, it must be possible for the observing agent to detect a pattern which does *not* fall into the "no" or "yes" categories. This is possible by monitoring the firing pattern of rules. A list of data items is used, each one recording a rule firing (i.e. an item is added every time all conditions of a rule matched the database). Each item is a list of the following form:

[time-slice, ruleset-no, cycle-no, rule-no]

e.g. $[2, 5, 1, 3]$ means that in time-slice 2, rule 3 fired in cycle 1 of the execution of ruleset 5. In the nursemaid's rulesystem, there is one rule which should always fire. Its action is a call to a procedure "compare-rulesystems". The result of this comparison (true or false) is added to the nursemaid's database as shown in table 7.

Table 7: Possible "normal" actions of observed rulesystem:

Rule-no:	Conditions	Actions
1	true	compare-rulesystems; add result to database
2	rulesystems are different	anomaly procedure; EXIT ruleset
3	otherwise	normal procedure; EXIT ruleset

In any time slice i, ruleset r should be run once and the possible patterns are as follows:

1. Trace contains $[i, r, 1, 3]$: nursemaid is normal and has not detected an anomaly
2. Trace contains $[i, r, 1, 2]$: nursemaid is normal and has detected an anomaly
3. Trace does not contain $[i, r, .., ..]$: indicates an anomaly in the nursemaid's monitoring (which may be due to unauthorised removal or modification of a rule).

5.4 Sensors and Effectors

A summary of sensors and effectors is shown in the following tables:

Table 8: Baby's sensors:

Sensor	function
$S_E(B, B)$	vehicle position vehicle energy level own interest level
$S_E(B, N)$	whether N is in control vehicle position vehicle energy level
$S_E(B, U)$	nearest treasure other treasure positions ditch - position and dimensions
$S_I(B, B)$	current state of own rulesystem
$S_I(B, N)$	current state of N's rulesystem

Table 9: Nursemaid's sensors:

Sensor	function
$S_E(N, N)$	vehicle position vehicle energy level
$S_E(N, B)$	whether B is in control vehicle position vehicle energy level
$S_E(N, U)$	position of energy supply ditch - position and dimensions
$S_I(N, N)$	current state of own rulesystem
$S_I(N, B)$	current state of B's rulesystem

Table 10: Baby's effectors:

Effector	function
$E_E(B,W)$	move-to (new position) collect treasure
$E_E(B,N)$	take control from N
$E_I(B,B)$	none at present
$E_I(B,N)$	none at present

Table 11: Nursemaid's effectors:

Effector	function
$E_E(N,W)$	move-to (new position) recharge vehicle
$E_E(N,B)$	take control from B
$E_I(N,N)$	repair own rulesystem
$E_I(N,B)$	repair B's rulesystem

Notes:
(1) $S_E(i,j)$ are sensors of A_j's effects on the external environment. Note that energy level, interest etc. are defined as "external" environmental states because they are values being regulated by A_j as part of its homeostatic task being simulated in the virtual world.

(2) $S_E(N,N)$ has values only when N is in control, i.e. it records its own effects. If B is in control, this information is detected by $S_E(N,B)$. The same applies to B's sensors, meaning that there is some redundancy.

(3) $E_E(i,j)$ are the effectors for changing A_j's effects on the environment. The one possibility to do this is to stop it and take over control.

(4) $E_E(i,i)$ (i = N or B) is excluded unless there is a reason why an agent should correct its own (faulty) effects on the environment, e.g. by shutting down a move effector (e.g. engine) and starting a backup.

(5) Repairing of own rulesystem should probably be disallowed (included here only for completeness).

(6) The nursemaid knows nothing about treasure or level of interest; it regards the treasure objects as obstacles.

(7) B cannot repair rulesystems in the initial implementation; it can only report problems.

5.4.1 Simulation of Sensors and Effectors

Sensors and effectors are simulated as shown in the following table:

Interface	Implementation
External sensors	SIM-AGENT sense update procedure
External effectors	SIM-AGENT do-action procedure
Internal sensors	user-defined procedures
Internal effectors	user-defined procedures

External sensors and effectors are simulated by built-in procedures in the SIM-AGENT development environment. During each scheduler time-slice the rulesystem of an agent is "run" once. At the beginning of each run, the agent's sensors are updated and it is up to the rulesystem to fetch and interpret the sense data. At any time during the run (although normally at the end), the SIM-AGENT procedure "do-action" can be called, which simulates an effector action (although the scheduler does some additional management of actions to ensure equal treatment of agents). To monitor the operation of sensors and effectors, these procedures should leave observable traces corresponding to normal operation. This will probably involve the use of trace mechanisms within the procedures, but the exact method will be determined from the results of the minimal version.

5.5 Implementation of Models

The nursemaid and baby must have models of each other. In a non-trivial environment, this model may be learned by repeated observation.

5.5.1 Model of External World

Medium-level sensor view (low-level would be pixels). Snapshot of the world is a list of the positions of all objects at that time instant. A single data "unit" is a history of snapshots (effectively this is the trajectory of the vehicle while it is under the observed agent's control).

(1) $M_E(N, U)$ - Model of the external world with no other agent present. This contains two components:

(a) what the world normally does spontaneously, e.g. treasure always seems to appear near a ditch and decays within a certain time period.

(b) what the world does in response to various actions.

In the minimal version, both these components are trivial and do not need to be learned. For example in case (b), the agents have an effector move-to(x,y) and the "model" only states that the new position should be at (x,y). Slippery ground near a ditch would lead to an anomalous new position which overshoots the expected one. In the initial version, recharging energy or "collecting" treasure only increases the energy or interest level respectively.

(2) $M_E(N, B)$ - The nursemaid's model of the baby's external effects on the world. This model consists of rules for predicting the behaviour of the vehicle under the baby's control. For example, if there is treasure nearby it will move towards it; if it is too near a ditch it will move away etc. An example of an anomaly would be the baby suddenly moving away when treasure is nearby. Such a model m be more accurate if it is generated by a rule-induction

component and the resulting rules are statistical (i.e. have certainty factors). However, it is possible to handcraft a simple version.

(3) $M_E(B, N)$ - The baby's model of the nursemaid's external effects. This is the converse of the above, but is expected to be simpler because the trajectory of the vehicle under the nursemaid's control can only be a motion towards the energy source or out of the ditch.

(4) $M_E(B, B)$ and $M_E(N, N)$ are redundant. To show the redundancy, we can consider $M_E(B, B)$, which involve B observing its own behaviour externally and generating an additional set of rules to make predictions about its behaviour (similar to asking :"how will this part of me react when X happens?"). However, there is no fundamental difference between $M_E(B, B)$ and another model of B generated by an additional observing agent A, i.e. $M_E(A, B)$. The only difference is in the SIM-AGENT implementation in that $M_E(A, B)$ would be inserted into the rulesystem of A instead of B. If it were to be inserted into B's rulesystem, the new extended rulesystem would look as follows (where R1 and R2 are two collections of rules):

R1: monitor R2's performance externally: predict its actions and detect anomalies.

R2: normal rulesystem of B

If each collection of rules is developed independently (as they should be for independent monitoring), this is the same pattern that already exists *between* agents. In the same way it may be possible to include mutually observing rulebases within the same rulesystem, in which case the whole distributed reflective architecture would be embedded within a single SIM-AGENT agent. At present I do not see any advantage in including this kind of monitoring *within* the SIM-AGENT definition of an agent. It would only be important if we wished to have this kind of single-agent reflection *without* a concurrent, independent meta-level in order to investigate the significance of introducing concurrency (see later).

5.5.2 Model of Internal World

The model of the internal world (i.e. the network from the viewpoint of a participating agent A_i), is divided up into $M_I(i, i)$ and $M_I(i, j)$. In the initial version, $M(i, i)$ is excluded due to the infinite regress problem (see below).

$M_I(N, B)$ is an implicit expectation that the static rulesystem pattern of B will remain unchanged. $M_I(B, N)$ is an expectation that the dynamic rulesystem trace of B will include one of the two normal patterns defined in 5.3.2.

In the minimal version, $M_I(N, N)$ is specifically concerned with the static pattern of $R_I(N)$ which shows its comparison of patterns stored in its database with the actual rulesystem, leading to an infinite nesting of this part of the rule system. (In later versions, however an alternative method of representation may be considered which could overcome this problem). It is interesting to note that if both N and B were to use the same method of observation (and representation), then the distributed architecture would also not overcome the infinite regress problem (it would be an infinite recursion with an additional level of indirection).

$M_I(B, B)$ is excluded since it cannot be an independent thread observing its own execution, and we are not including concurrency within a rulesystem at present.

5.6 Limits of the Simulation

In a simulation like the one presented here, we cannot have true independent monitoring of activity patterns in the way that this is possible in the real world (e.g. observation of customers entering and leaving a bank). Instead the production of activity patterns must be simulated. The simplest way to do this is to enable tracing within the rule interpreter and in the procedures. Since this involves the insertion of code that would otherwise not exist, it is not really independent monitoring.

However, for the purposes of conceptual exploration, an artificial tracing mechanism does not present a serious problem. If an intruder were to disable it, an anomaly should be immediately detectable (because the trace output would suddenly stop). The intruder may replace the tracing mechanism with disinformation (simulating a "healthy" trace while modifying the actual code). However, this is difficult in a load-balancing architecture where there are also other sources of information, such as worsening of the external environment. For example, an apparently normal trace may be associated with negative effects on the environment of the sort defined in section 4.5.1.

In an effective immune system, the intruder itself should leave an anomalous trace. This is a problem of making the internal sensor data comprehensive enough. This may be attempted in an enhanced version of this implementation, but only if it turns out to be necessary for the solution of the problem.

6 Comparative Analysis

We now have to define more precisely what "the problem" is in terms of the scenario just introduced. We will say that an architecture has solved the distributed reflection problem if it can detect and report any failure or possible failure as defined in section 4.5.1. The whole network (macro-level) is being tested and not the individual agents, which are just "modules" of the distributed control system. In this way, there should be a collective detection of "nonself" (in immune systems terminology) albeit in a very rudimentary way in the initial implementation due to its unrealistic constraints.

Which architectures are to be compared? The research question focuses primarily on the problem of meta-level closure (agents being each other's meta-levels). We therefore compare closed architectures with non-closed (open) architectures.

Figure 5 shows four different configurations. C1 and C3 are open networks, while C2 and C4 are closed. The arrows correspond to different methods of observation (and repair if applicable). The initial implementation introduced in the last section is a minimal version of C2 where the arrow labelled "a" corresponds to the monitoring of the static rulesystem pattern along with the external effects of B. The arrow labelled "b" represents the monitoring of N's rule firing patterns together with its external effects. The design space introduced in section 3 (D_1 to D_4) contains versions of C2 only. C3 would involve the introduction of another baby (or nursemaid) which would specialise in a different subtask (e.g. certain kinds of treasure), but no new kind of monitoring would be necessary. C4 requires an additional form of monitoring represented by "c" (which may focus on a different type of rule firing pattern). For the primary question, the three most important comparisons are: C1/C2, C2/C3 and C3/C4. The difference involved in each of these involves only the addition or the redirection of a single arrow. For example, to get from C1 to C2 we simply add the monitoring capability that makes the baby "gifted".

A secondary question concerns the significance of independent and concurrent monitoring. This is the same as the difference between single-agent reflection (where there is no concurrency or independent perspective, as explained in section 5.5.2), and the open form of distributed reflection (C1). Since C1 can be "collapsed" into a single agent with a concurrent meta-level, this is effectively a comparison between two single-agent architectures. This question will only be investigated in detail if it turns out to be important for the primary question.

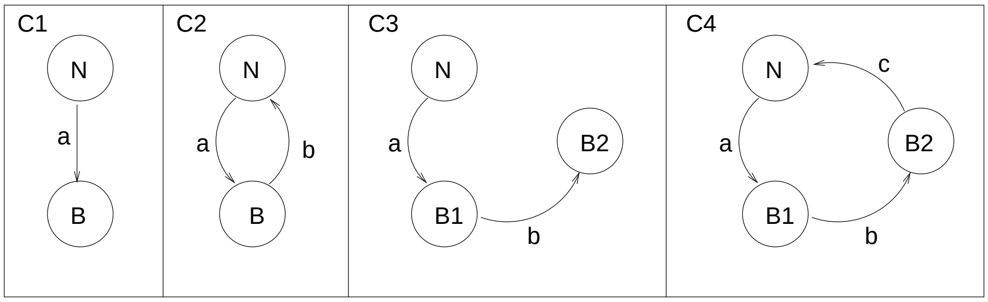

Figure 5: Open and closed network configurations

7 Evaluation

The main objective is to produce an architecture which can solve the distributed reflection problem as defined in 1.6, the hypothesis being that an architecture with meta-level closure can be developed and can overcome the inadequacies of the same kind of architecture without meta-level closure. If an architecture is found, an analysis of why it works will be given, along with its limitations. If the above hypothesis could not be confirmed, (e.g. because a closed architecture was no better than a non-closed one, or there were unexpected difficulties in the development of the architecture), the reasons for this should be given.

The second objective is conceptual clarification: The methods used in this work are largely informal and explorative, i.e. there are no very precise formal definitions to begin with; instead the concepts should become gradually more precise during the iterative process of design, implementation, testing, modified design, etc. The main concept to be explored and developed more precisely is meta-level closure. The original definition (organizational closure) relies on an imprecise notion of "meta-level". In the initial implementation described here, a "meta-level" for an object is defined as something that can detect anomalies in the object. Therefore the following question should be answered: can the first objective (producing an architecture which solves the problem) be achieved using this minimal definition of meta-level (and hence meta-level closure) or do we require a more complex definition?

I am also aiming to answer the following additional questions (for both open and closed architectures):

- What difference does the number of agents make: what are the advantages of more than two? It would appear that three agents would be more reliable. For example, if agents detect failures in each other's anomaly-detection, an additional agent must determine which one has actually failed. Is there a scaling up problem - and if so, how will it scale up?

- *Design* redundancy vs. *version* redundancy: what is the difference between agents based on the same design and those based on independently developed designs (as for example in [26]. It would seem that independent designs are more robust. However, this makes it more difficult for agents to repair each other's code. Similarly, the agents will generally not have the ability to do each other's tasks equally well (E.g. B is motivated by the need to maintain its "interest" level but N is not).

- Load distribution vs. specialist monitoring agents: should there be agents whose sole task is to monitor other agents? In the present implementation, both agents switch attention between their monitoring tasks and their "normal" tasks. Quality evaluation of the performance of a specialist monitoring agent is difficult (as it is mostly a passive observer).

The thesis is particularly successful if the new concepts gained can be applied to agent architectures in such a way that progress on the autonomy problem can be made which would otherwise not have been possible. This is the subject area which the work will focus on. However, it may also make an indirect contribution to cognitive science, in particular in the area of underlying architectures thought to give rise to "high-level" cognitive capabilities. E.g. what role does low-level (unconscious and massively parallel) anomaly-detection play in the high-level state of being surprised, anxious etc?

8 Provisional Timetable

The following timetable is a summary of work planned in the next two years.

Time period	Actions
Jun 1999 - Aug 1999	Implement minimal version with two agents Determine which changes are necessary
Sept 1999 - Oct 1999[1]	Attempt non-minimal two-agent version[2] Attempt comparison (1) write report 4
Nov 1999 - Jan 2000	Further work on non-minimal versions as necessary Design three-agent version[3]
Feb 2000 - May 2000	Implement three-agent version Attempt comparisons (2) and (3) Write report 5
Jun 2000 - Sep 2000	Further implementation work as necessary Attempt comparisons (1), (2) and (3)
Oct 2000 - Dec 2000	Write first thesis draft Write report 6
Jan 2001 - Mar 2001	Further work as necessary
Apr 2001 - Jun 2001	Write second draft of thesis

Notes:
[1]Normally this should be December, but an attempt will be made to synchronise with other thesis group meetings.

[2]Only possible if the required design changes are not too radical, in which case the actual implementation may not have begun by report 4.

[3]Only possible if there were no significant problems with the two-agent version. Otherwise the problem should be investigated to determine what can be learned from it. Further experiments will then be planned as necessary.

References

[1] J. Baxter, R. Hepplewhite, B. Logan, A. Sloman "SIM-AGENT: Two Years On". Technical Report CSRP-98-02, School of Computer Science, University of Birmingham, 1998.

[2] *Goal Processing in Autonomous Agents*, PhD Thesis, School of Computer Science, University of Birmingham, 1994.

[3] D. Dasgupta and N. Attoh-Okine. "Immunity-Based Systems: A Survey". *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Orlando, October 12-15, 1997.

[4] D. Dasgupta and S. Forrest, "Novelty-Detection in Time Series Data using Ideas from Immunology", *Proceedings of the International Conference on Intelligent Systems, 1997*

[5] D. Davis "Reactive and Motivational Agents: Towards a Collective Minder" *Proceedings of ATAL96 Workshop at ECAI96*, Budapest, Hungary, August 1996.

[6] G. De Giacomo, R. Reiter, M. Soutschanski "Execution Monitoring of High-Level Robot Programs" in Proceedings *Common Sense 98.*

[7] P. D'haeseleer, S. Forrest, P. Helman. "An Immunological Approach to Change Detection: Algorithms, Analysis and Implications" in *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, Oakland, California. IEEE Press Los Alamitos, CA, pages 110-119.

[8] B. Ekdahl, E. Astor and P. Davidsson "Towards Anticipatory Agents" in *Intelligent Agents - Theories, Architectures and Languages*, edited by M. Wooldridge, N. Jennings, 1995, Springer, pages 191-202.

[9] J. Ferber "Conceptual Reflection and Actor Languages" in *Meta-Level Architectures and Reflection*, edited by P. Maes and D. Nardi, North-Holland, 1988, pages 177-193.

[10] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukun "Self-Nonself Discrimination in a Computer" in *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*

[11] G. Günther, "Cognition and Volition: a Contribution to a Cybernetic Theory of Subjectivity" in: *Beiträge zur Grundlegung einer operationsfähigen Dialektik, zweiter Band*, Felix-Meiner Verlag, Hamburg, 1979.

[12] G. Günther, "Life as Polycontexturality" in: *Beiträge zur Grundlegung einer operationsfähigen Dialektik, dritter Band*, Felix-Meiner Verlag, Hamburg, 1980.

[13] Y. Ichisugi, S. Matsuoka, and A. Yonezawa, "RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel" in *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-level Architectures*, page 24-35. Tokyo, November 1992.

[14] R. Kaehr, "Zur Logik der 'Second Order Cybernetics'", *Kybernetik und Systemtheorie - Wissenschaftsgebiete der Zukunft?*, edited by ICS (Institute for Cybernetics and Systems Theory), IKS-Berichte, Dresden, 1991. (in German)

[15] R. Kaehr and T. Mahler, "Introducing and Modelling Polycontextural Logics", *Proceedings of the 13th European Meeting on Cybernetics and Systems Research*, Vienna, 1996.

[16] G. A. Kaminka and M. Tambe, "What is Wrong With Us? Improving Robustness Through Social Diagnosis", *Proceedings of the 15th National Conference on Artificial Intelligence* (AAAI-98)

[17] S. Kornman. "Infinite Regress with Self-Monitoring" in Proceedings *Reflection'96*, San Fransico, April 21-23, 1996.

[18] T. Laengle and U. Rembold "A Distributed Control Architecture for Intelligent Systems" *Proceedings of the International Conference on Advanced Sensor and Control Systems*, 1996.

[19] D. B. Lenat (1983). "EURISKO: a program that learns new heuristics and domain concepts. The nature of heuristics III: program design and results." *Artificial Intelligence, 21* (1 & 2), 61–98. CSRP-98-14, School of Computer Science, University of Birmingham, 1998.

[20] P. Maes. "Issues in Computational Reflection" in *Meta-Level Architectures and Reflection*, edited by P. Maes and D. Nardi, North-Holland, 1988, pages 21-35.

[21] H. Maturana and F. Varela *Autopoiesis and Cognition: The Realization of the Living.* D.Reidel Publishing Company, 1980.

[22] S. Matsuoka, T. Watanabe and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming" in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, Geneva, Switzerland, Springer-Verlag, 1991, pages 231-250.

[23] K. Meyer, M. Erlinger, J. Betser, C. Sunshine, "Decentralized Control and Intelligence in Network Management" *Proceedings of the 4th International Symposium on Integrated Network Management*, Santa Barbara, CA, May 1995.

[24] M. Minsky, "Matter, Mind and Models" in *Semantic Information Processing* MIT Press, 1968.

[25] M. Minsky *Society of Mind*, Simon and Schuster, 1986.

[26] N.H. Minsky "Independent On-Line Monitoring of Evolving Systems" *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[27] B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner and B.C.Williams "An Autonomous Spacecraft Agent Prototype" in *Autonomous Robotics*, **5**, 1-27, Kluwer Academic Publishers, Boston, 1998.

[28] P. Rosenbloom, J. Laird and A. Newell, "Meta-Levels in SOAR" in *Meta-Level Architectures and Reflection*, edited by P. Maes and D. Nardi, Elsevier Science Publishers B.V. (North-Holland), 1988, pages 227-239.

[29] A. Sloman "What sort of architecture is required for a human-like agent?" in *Foundations of Rational Agency*, edited by Wooldridge, M. and Rao, A., Kluwer Academic Publishers, 1997.

[30] A. Sloman "Supervenience and Implementation", Technical Report, Cognition and Affect Project, School of Computer Science, University of Birmingham, 1998.

[31] J. M. Sobel and D. P. Friedman, "An Introduction to Reflection-Oriented Programming", *Reflection 96*, San Francisco, April 1996.

[32] B. C. Smith, "Reflection and Semantics in a Procedural Language" MIT Technical Report, MIT-LCS-TR-272, 1982.

[33] M. Tambe and P.S. Rosenbloom "Architectures for Agents that Track Other Agents in Multi-agent Worlds" in *Intelligent Agents, Vol. II* Springer-Verlag, 1996 (LNAI 1037)

[34] F. Varela *Principles of Biological Autonomy*, North-Holland, 1979.

[35] H. von Foerster. *Observing Systems* Intersystems, Seaside, CA, 1981.

[36] I. Wright, A. Sloman. "MINDER1: An Implementation of a Proto-emotional Agent Architecture", Technical Report CSRP-97-1, School of Computer Science, University of Birmingham, 1997.