DISTRIBUTED REFLECTIVE ARCHITECTURES FOR ANOMALY DETECTION AND AUTONOMOUS RECOVERY

by

CATRIONA MAIRI KENNEDY

A thesis submitted to the Faculty of Science
of the University of Birmingham
for the Degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
Faculty of Science
University of Birmingham
June 2003

Abstract

In a hostile environment, an autonomous system requires a reflective capability to detect problems in its own operation and recover from them without external intervention. We approach this problem from the point of view of cognitive systems research. The simplest way to make such an autonomous system reflective is to include a layer in its architecture to monitor its components' behaviour patterns and detect anomalies. There are situations, however, where the reflective layer will not detect anomalies in itself. For example, it cannot detect that it has just been deleted, or completely replaced with hostile code.

Our solution to this problem is to distribute the reflection so that components mutually observe and protect each other. Multiple versions of the anomaly-detection system acquire models of each other's "normal" behaviour patterns by mutual observation in a protected environment; they can then compare these models against actual patterns in an environment allowing damage or intrusions, in order to detect anomalies in each other's behaviour. Diagnosis and recovery actions can then follow.

In this thesis we present some proof-of-concept implementations of distributed reflection based on multi-agent systems and show that such systems can survive in a hostile environment while their self-monitoring and self-repair components are repeatedly being attacked.

The thesis also compares the cognitive systems paradigm used in the implementations with the paradigm of distributed fault-tolerance and considers the contributions that one field can make to the other.

Acknowledgements

I am grateful to the members of the thesis committee Aaron Sloman, John Barnden and William Edmondson for their advice and challenging questions in the production of this thesis. The following people have also provided valuable feedback during the writing up stage: John Bullinaria, Robert Goldsmith, Jun He, Kay Hughes (QinetiQ, Malvern), Manfred Kerber, Tom McCutcheon (Defence Science and Technology Labs, Malvern), Gethin Norman and Manuella Viezzer.

Contents

List of Figures

List of Tables

Part I

CONCEPTUAL FRAMEWORK AND METHODOLOGY

Chapter 1

The Problem

1.1 Introduction

In the real world there are many situations where an autonomous system should continue operating in the presence of damage or intrusions without human intervention. Such a system requires a self-monitoring capability in order to detect and diagnose problems in its own components and to take recovery action to restore normal operation. Autonomous response and reconfiguration in the presence of unforeseen problems is already a fairly established area in remote vehicle control systems that have to be self-sufficient. See for example [Pell et al., 1997].

It is also becoming increasingly desirable for ordinary software systems (such as operating systems) to act autonomously to correct any negative effects of damage or intrusion and continue to provide a satisfactory service. There may not always be time for a system administrator to understand what is going on and take action. An example is the need to respond fast to prevent the negative effects of a Trojan Horse by identifying and suppressing hostile code execution and attempting to reverse or limit any damage that has already happened. Such systems are said to be "intrusion-tolerant" [Wu et al., 1999, Dacier, 2002] in a similar way to systems that are fault-tolerant.

This thesis addresses the above problem using the paradigm of *cognitive systems*. We define a minimal "cognitive system" to be the same as an "agent" of the type defined in [Russell and Norvig, 1995]. That is: anything that perceives its environment through sensors and acts upon that environment through effectors. (For reasons given later, our use of the term "agent" will be more specific than this).
The following additional features are *desirable* for a cognitive system:

1. learning and adaptation
2. reasoning and explaining (including anticipating and planning)
3. reflection (reasoning about and modifying aspects of own behaviour)
4. self/nonself distinction (to be elaborated later)

The thesis proposes a novel self-monitoring architecture for a cognitive system satisfying the above requirements. Therefore, the thesis is particularly concerned with (3) and (4) and they will be explained in detail later in the next sections.

We focus on the problem of survival in a hostile environment (of the type outlined above) as a special case of a more general class of problems requiring self-reflection in cognitive systems (we will give examples later). We define a "hostile" environment as one in which *any* part of the system may be damaged or modified, including the self-monitoring components.

1.2 Reflection

When discussing reflection, there are two aspects to be considered: first, there is the *structure* of the reflection: what processes and components are involved in it, and how to they interact? Secondly, the *content* of the reflection is important: what kind of thing should the system know about itself to enable it to achieve its goals?

1.2.1 Self-monitoring and computational reflection

The broad notion of *computational* reflection is defined by [Maes, 1988] as follows:

> A reflective system is a system which incorporates structures representing (aspects of) itself. We call the sum of these structures the *self-representation* of the system. This self-representation makes it possible for the system to answer questions about itself and to support actions on itself.

In particular this means the following:

- A running program has a self-representation that it can inspect and modify, or (as is currently more common) it can “open up” its self-representation for inspection and modification by a user.
- The self-representation must be *causally connected* in that any inspection must provide accurate information about the program’s internal workings and any modification must change the behaviour of the program in the intended way. The term used is *causally connected self-representation* (CCSR).

Self-monitoring is a restricted form of computational reflection and allows a program to monitor what it is currently doing. By contrast, the most widely used forms of computational reflection have the purpose of giving a program access to the *implementation* of its different kinds of behaviour. We call the latter *reflective programming* mechanisms because they concern a program’s “knowledge” of how its different types of behaviour are programmed on a lower level. For example, if a program can send messages or call applications remotely, it may require knowledge about its current implementation of these features and the alternatives available [Hof, 2000].

If we take an everyday human example, such as walking down stairs, the contrast between reflective programming and self-monitoring can be illustrated as follows:

- Reflective programming:
  - **Inspection:** ask someone (or they ask themselves) how they normally walk down stairs.
  - **Modification:** they change the general method if they are in a building with spiral staircases.
- Self-monitoring:
  - **Inspection:** ask someone what they are doing while they are walking down a particular staircase - is there anything unexpected or dangerous in this particular situation?
  - **Modification:** they have to walk more slowly because these particular stairs are slippery

Note that reflection in both cases is not just about passive self-knowledge, but also about active self-modification (we use the term self-monitoring as a shorthand for both monitoring and modifying).

Both types of reflection require a self-representation. In the case of self-monitoring, however, the representation also says things about actual behaviour that may be different from expected or desired behaviour. Thus the required self-representation contains the following components:

- a causally connected map of the system’s components (which can be inspected and modified as required)

- representation of expected or desired states (internal model)
- representation of actual states (sensor readings)

1.2.2 Anomaly-detection

A cognitive system should make a distinction between its desired and actual states. We call a significant difference an *anomaly*. Computational reflection, by contrast, is normally only concerned with what the system *should* do (how it is programmed).

An anomaly may be merely unusual (in that measurements are wildly different from expected) or "impossible" (in that measurements seem to contradict the theory on which expectancy is based). In this thesis we are mostly concerned with the former, although we will occasionally refer to the latter possibility. An example of "impossibility" is given in [Kennedy, 1998a].

1.2.3 Self/nonself distinction

In nature, immune systems detect foreign invaders by distinguishing between "self" and "nonself". (A general overview can be found in [Hofmeyr, 2000]). The recognition of self happens when actual patterns (sometimes called "pattern instances" in AI terminology) match "expected" patterns already known by the immune system (the immune system's "knowledge" being equivalent to a model). Unknown micro-organisms are regarded as "nonself" and may be attacked.

The new research field of *artificial* immune systems is surveyed in [Dasgupta and Attoh-Okine, 1997] and uses this concept of self/nonself distinction. Their limitations will be discussed in Chapter 2. In our investigation of autonomous reflective systems, we make this immune system capability a requirement.

A self/nonself distinction requires a model of "self". The model is a representation of the normal patterns of behaviour (or other states) of the system and can be associated with a map of the system's components. Any anomalous pattern may then be categorised as "nonself". In software this may be associated with a new component (such as a new version of a software package) or a different kind of user. The appearance of a "nonself" pattern may be associated with a change in the static component map.

It is advantageous for the system to evaluate whether an unknown pattern is "good" or not. For example, a foreign but harmless pattern may be "accepted" into the model of "self" after a period of observation. Thus, we can identify two kinds of capability:

- *Pattern anomaly-detection*: the system should detect unusual features when observing its own behaviour, or the absence of expected features. In immune systems terminology, it detects "nonself" (patterns that it classes as "foreign"). In the case of absence, a "nonself" is not directly observed but may be inferred as a cause of the absence.

- *Quality evaluation*: the system should evaluate the current state of the world (or its own performance) as good, bad, satisfactory, etc. This evaluation is only useful if a foreign pattern has also been detected. If the system is very "trusting", it can learn to accept as "self" those foreign patterns that have not been associated with a quality deterioration.

The simplest (minimal) situation is a "pessimistic" policy where anything unknown is assumed to be "bad" and everything familiar is "good". A more detailed discussion of the distinction between the two is given in [Kennedy, 1998b].

These two capabilities can also be used non-reflectively, for example when encountering new features in the environment.

1.2.4 Self-diagnosis and repair

The recognition of “nonself” should trigger a self-protective response. Existing AI techniques may be used to enable autonomous reconfiguration in the event of an anomaly. Ideally the system should identify the component or program that produced the anomalous pattern and try to prevent it from causing damage. This is a difficult diagnosis problem. Moreover, there may not always be a single “component”. For example, a user may enter an anomalous sequence of commands. In this case, the anomalous sequence itself may be classed as “hostile code” and any autonomous response should prevent its execution.

Another possibility is an unusual interaction between “trusted” components due to an unforeseen environmental situation that the system was not originally designed for. An example could be a change of working practice, causing an unintended negative effect such as extremely slow response or confidential data becoming accessible by unauthorised parties. In such situations, it may be possible to use planning techniques to avoid activating the problematic interaction.

Autonomous response is beginning to be addressed in intrusion detection systems. See for example [Somayaji, 2000] where a response is in the form of suppression or delay of system calls initiated by an executing program, any delay being proportional to the number of recent anomalies observed in the program’s system call patterns. Although they do not use a measure of “quality”, their approach is relevant to the subproblem of diagnosis and response, and we will return to it later.

We are assuming here that the “quality” of performance can be precisely specified and a deterioration detected. In Part 2 we demonstrate a simple prototype that uses these techniques successfully, along with autonomous response such as pinpointing the component and suppressing it.

1.2.5 Autonomous acquisition of a self-model

In the language of self/nonself distinction, the expected behaviour patterns together with acceptable quality-states can be said to constitute the system’s “self-model”. If new software components are correlated with unacceptable or undesirable states then the system does not accept them as part of “self” and if possible their execution is suppressed or prevented.

One may argue that it is easier conceptually to talk about “expected or desired state” (of self) and “actual state” (of self). However this is not strictly accurate in the biological sense, since the actual state may include elements that the system regards as being “foreign” even though an external observer may regard them as part of the system itself. In other words, the boundary between self and nonself should be autonomously generated by the system during its development and not defined by an external observer or designer. This is a major foundation of autopoiesis theory [Maturana and Varela, 1980], which we will outline in Chapter 3.

To be consistent with this biological interpretation, we are aiming for an autonomous system whose self-model is gradually “bootstrapped” by self-observation during its interaction with the world.

1.2.6 Summary of requirements

The following is a summary of requirements of the reflective capabilities of a cognitive system:

1. explicit representation of “self” with ability to reason about it and explain its status
2. ability to detect “nonself” as an anomaly or a threat” to correct operation
3. autonomous acquisition of a self-model
4. self-diagnosis and repair: ability to restore correct operation in the event of attack without external intervention

“Requirements” are intended to mean “desirable features”.

1.3 Main Thesis Contributions

To explain the thesis contribution we look at Marvin Minsky's "Emotion Machine" [Minsky, 2002]. Section 1-10 introduces the idea of an "A-Brain" being monitored and modified by a "B-Brain". The A-Brain plays the same role for the B-Brain as the environment plays for the A-Brain:

> Your A-Brain gets signals that stream along nerves that come from devices like ears, nose, and eyes. Then using those signals, your A-Brain discerns some events that occur in the outer world. It also has ways to react to these, by sending out signals that make muscles move - which in turn can affect the state of the world. Your A-Brain may also record some descriptions of its recent perceptions and subsequent actions.
>
> Your B-Brain does not connect to the world, or even to your eyes or ears; it only can interact with A. So B cannot 'see' real things, but only A's descriptions of them. Therefore, so far as B is concerned, the A-Brain is its 'outer world'-a world that contains no physical "things", but only their second-hand descriptions. Nor can B directly move things in the world; it can only send signals that change A's behaviour.

A third "C-Brain" might in turn monitor the B-Brain. Minsky then suggests "closing the loop" so that the C-Brain monitors the A-Brain. Alternatively the A- and B-brains may mutually monitor and modify each other. Such a "closed" architecture may provide advantages over an "open" system:

> *Student: Wouldn't that expose us to the risk of dangerous circular processes? What if one tried to turn all the others off, or deduced a logical paradox? All sorts of terrible things might happen. A machine that could keep changing itself might lead to inconsistencies?*
>
> Yes indeed, but it's worth that risk because every attempt to prevent "self-reflection" would cripple the systems in other ways. I suspect that what our brains did, instead, was to evolve ways to detect such bugs - and develop techniques for suppressing them...

The main claim of the thesis is that such "circular" processes are useful and can be implemented to solve real problems, while satisfying the requirements of a cognitive system above. An additional claim is that hostile environments exist in which such an architecture would provide advantages over an architecture without "circularity" (effectively supporting Minsky's reply to the student above). To show this we have implemented several prototypes as a proof of concept.

1.3.1 Proof of concept implementations

We have used the term "closed reflection" for Minsky's hypothetical architecture above (in which all processes are monitored) and "distributed reflection" for the more general situation in which there is mutual observation between processes.

The general aim is to allow any part of the system to be questioned or "criticised" so that no component has "absolute" authority. An example is the recognition that a particular approach to a problem is no longer suitable because a situation has changed, where the "unsuitability" may be inferred due to anticipated or actual quality degradation. The detection that a component has been attacked or damaged by a hostile environment is just one example of this kind of "criticism" of a component. We are specialising in "hostile" environments because it is easy to see when the system fails due to lack of self-monitoring (it does not continue to "survive"). The main work of the thesis has the following objectives:

1. Explore the different variants of distributed and closed reflective architectures. Identify key concepts and problems.

2. Develop prototype implementations of a closed architecture to show that it can satisfy the following requirements:

    (a) Feasibility: The architecture is feasible in practice and can behave consistently and coherently (i.e. it does not fail because of destructive interference between agents as predicted by the hypothetical "student" above)

    (b) Usefulness: It can enable survival in hostile environments.

    (c) Novelty: It can satisfy the requirements in Section 1.2.6 while comparable distributed paradigms only satisfy some of them.

1.3.2 Additional objectives

The thesis will also investigate the interface between cognitive systems and traditional fault-tolerance and identify the contributions that each field can make to the other. In particular, the novel aspects of our implementations with respect to existing fault-tolerant systems may also be a contribution to fault-tolerance research (in addition to being a contribution to cognitive systems research). Conversely, some existing techniques in fault-tolerance may contribute to cognitive science.

The methodology used to develop the implementations is informal and effectively translates cognitive science concepts into software engineering concepts (the "design-based" approach [Sloman, 1993b]). It has the following features:

- specification of "whole" cognitive architectures with emphasis on the interrelationships between components instead of specific algorithms or representations

- discovering new concepts and acquiring a more precise understanding by testing architectures (designs) in different types of environment (niches). Design is driven by *scenarios*, which are example situations that can occur in the given environment.

The thesis will evaluate the methodology and the new insights gained as a result.

1.4 The Reflective Blindness Problem

It is important to relate Minsky's A- and B-Brain architecture to real world systems and ask why we might need such a system of "closed" reflection, when possibly a simpler system might be sufficient.

The simplest way to make an autonomous system self-observing is to include a layer in its architecture to monitor its components' behaviour patterns and detect anomalies. There are situations, however, where the monitoring layer will not detect anomalies in itself. For example, it cannot detect that it has just been deleted, or completely replaced with hostile code. If we just add another layer to monitor the first one this would lead to an infinite regress. In a simple multi-layer architecture there is always a significant part of the system that remains vulnerable to attack because it cannot be observed or "criticised" by any component within the system. This weakness has been called the "blind spot" or "reflective residue" by some in the historical cybernetics community e.g. [Kaehr, 1991], although this has mostly been in a philosophical context. In previous papers [Kennedy, 1999a, Kennedy, 1999b, Kennedy, 2000] we called this problem "reflective blindness".

If we return to artificial immune systems, we see that the definition of "self" also has to include all monitoring levels. In other words, all components that are actually *making* the distinction between self and nonself must be included in the self-description. We will discuss in the next section whether existing artificial immune systems are actually able to do this in the required way. (We will not discuss how natural immune systems do this, since this would go beyond the scope of the thesis).

1.5 Are we Addressing an Unsolvable Problem?

One may argue that it is not worth addressing this problem because it corresponds to something that is unsolvable and nature does not solve it. We consider various forms of this argument.

1.5.1 Limits of formal systems

Some may argue that reflective blindness is just a fundamental limitation of formal systems in general and that this limitation will always exist for any artificial system that can be defined as a formal system. Therefore, it is pointless to try to solve it. In particular Gödel's Theorem [Podnieks, 1992] states that any introspective logical system will be "incomplete" in that there will always be some statement that cannot be proved within the system itself.

Even if we assume that all artificial cognitive systems can be defined as formal systems (an assumption that may not hold), the incompleteness argument is not relevant to this thesis. We are not aiming to *remove* any fundamental limits, but simply to "work around" them so that they do not cause serious problems for an autonomous system. We wish to explore architectures that can reduce the effects of these limitations.

1.5.2 Limits of conscious reflection

It may be argued that even human consciousness does not solve this problem, because we are "blind" about many of our own internal states but we still survive. This is a flawed argument because it refers to the limits of subjective experience (the "content" of consciousness) and not to the underlying architecture that produces the experience. Survival may be possible because the architecture causes the "right" contents to appear in our minds due to a self/nonself distinction at a lower (unconscious) level.

Although a detailed discussion of this issue goes beyond the scope of the thesis, it is mentioned here because it shows a common confusion between subjective content and objective (externally described) architecture. See also [Kennedy, 1999b].

1.5.3 Limits of natural immune systems

One may argue that nature does not solve this problem. For example, natural immune systems do not perfectly distinguish between self and nonself, otherwise they would not be vulnerable to AIDs or auto-immune diseases. Furthermore the immune system does not protect us from radiation or explosions. Therefore it seems to suffer from reflective blindness.

However, an immune system does distinguish between self and nonself *within a certain response time* and *to a sufficient degree of accuracy*. "Sufficient" can be understood in the context of evolutionary fitness. It does not mean invulnerability.

Similarly, in the case of autonomous systems, we can talk about "sufficient" reflective coverage, which means that the coverage enables it to survive in the hostile environments it is likely to find itself in. (See also [Kornman, 1996] for the concept of "reflective self-sufficiency").

"Sufficient" is not the same as "complete" coverage, which is unattainable since it would require sensors that can detect all possible events (something that cannot be defined for the real world). Instead, self-monitoring resources should be focused on the system's most critical management components. We discuss this further in Chapter 3. Scenarios for "hostile" environments will be presented in Part 2.

1.6 Why is the Problem Important?

Some may argue that the problem can be minimised by having a hierarchy of monitoring levels where the top one is “minimal” so that the amount of processing that is not being observed is minimised. They would argue that we cannot hope to do more than this because it is not possible to completely remove the problem.

To show that “minimising” a top-level is unsatisfactory, we return to the example of intrusion-detection. The simplest way to protect the monitoring layer would be the addition of new “layers” to trap the intrusions that lower ones have missed, and to detect any problems in the layers themselves. The very top layer (that is not subject to monitoring) could be made “minimal”. For example, it may only contain a few lines of code that regularly test other levels to see if they are “alive”. Even if it is disabled the bulk of the intrusion detection would still be available.

However, if it is a pure hierarchy, the top level can be disabled without any other part of the system detecting this. The attacker then has an arbitrary amount of time to disable the next level down the hierarchy (which again will not be detected) and so on, until all levels are disabled. Even worse, an attacker could *modify* the code of the topmost level so that it disables or misleads all other levels. For example, the few lines of code could be replaced so that they report non-existent problems with other levels. Hence they will not be believed if they detect a real intrusion. In this case the enemy only needs to attack the top-level component.

1.7 Structure of the Thesis

1.7.1 Chapter Breakdown

Part One introduces the concepts of distributed reflection and reviews related work.

Chapter 2 reviews existing architectures that involve different types of reflection without distribution. We argue that they do not address the reflective blindness problem although they may appear to have solved it. However, some techniques used in the non-distributed approaches are useful and can be integrated into a cognitive architecture with the required features.

Chapters 3 introduces concepts for specifying cognitive architectures and gives examples of distributed reflection, including the special case of “closed” reflection using these concepts. A major problem is ensuring that components do not interfere with each other in an arbitrary way. The chapter then goes on to examine related work in biology and philosophy (in particular Maturana and Varela’s autopoiesis theory), which emphasise the need to “maintain constant” the system’s ability to survive. The main conclusion is that the control systems paradigm can help to operationalise these philosophical concepts so that components only interfere with each other in a way that ensures the survival of the system. The chapter also contains a discussion on the relationship between “closed” reflection and self-nonself distinction.

Chapter 4 applies the control system paradigm to the concepts introduced in Chapter 3 to develop a minimal architecture that can be implemented. The methodology for software design is also introduced and examples of “design space” regions are given.

Chapter 5 compares the architecture developed in Chapters 3 and 4 with existing distributed architectures and asks whether we are producing something really new. The chapter also reviews fundamental concepts in distributed fault-tolerance. We show why a distributed fault-tolerant architecture can go some way to solving the problems of survival in a hostile environment. In addition, some problems of inconsistency and illogical behaviour mentioned with regard to Minsky’s model can be solved by fault-tolerant architectures. However, these architectures are very limited in their ability to reason and learn by experience. Therefore they should be combined with AI and cognitive systems techniques.

Part Two of the thesis presents some proof-of-concept implementations in different kinds of environment

Chapter 6 presents a simple virtual world in which the system has to survive, and the kinds of damage and attacks that it can undergo. This leads to “environmental restrictions” similar to fault assumptions in fault-tolerance. A minimal implementation with two agents is presented in a world in which random deletion of components can occur.

Chapter 7 presents the algorithm that was used to acquire a model of “self”. The most important findings in this chapter are the need for context dependence and the importance of autonomous generation of contexts.

Chapter 8 extends the minimal implementation of Chapter 6 to resist insertion of hostile code that causes some components to “lie”. The anomaly detection is extended so that degradations in the “quality” of the state of the world and the system behaviour are also detected.

Chapter 9 presents a second extension to the prototype, which involves an additional agent and hostile code being inserted into the anomaly detection components of an agent. A scenario in which agents mutually “accuse” each other of being wrong is addressed and prevented by adding a voting mechanism. Since this mechanism is similar to some methods used in fault-tolerance, the chapter concludes by looking at possible ways in which existing fault-tolerance mechanisms can be incorporated into a cognitive system.

Part 3 of the thesis looks at issues raised during implementation of the prototypes and suggests ways of overcoming their current limits.

Chapter 10 relates the abstract cognitive architecture to real world computing and shows how the different components of the architecture have relevance to intrusion detection.

Chapter 11 argues that the architecture can be extended so that it can be mapped onto real world concerns.

Chapter 12 evaluates what has been achieved and makes some proposals for future work.

1.7.2 Thesis limits

The thesis excludes a detailed exploration of the following topics:

- Formal methods: we are exploring informal architectures only (see Chapter 4, Section 4.3.3). The thesis complements formal methods.

- Low-level learning and anomaly-detection algorithms (such as rule-induction, neural networks or statistical pattern analysis). The architectures we are investigating are concerned with high-level anomaly-detection only (e.g. when an event is anomalously absent or an unexpected event is encountered). Reasons for this are given in 4.3.2.

- Languages for computational reflection. We are exploring cognitive architectures only, although the general concept of reflective languages is described in order to contrast it with our approach.

Examples from these categories are described briefly in the text when required, but they are not investigated in detail.

Chapter 2

Limits of Existing Approaches

2.1 Introduction

In this chapter we examine related fields of research and identify some useful concepts that they provide. The chapter covers the following diverse research fields:

- self-monitoring intelligent agent architectures
- computational reflection
- artificial immune systems

Some may appear to have solved the reflective blindness problem already without any specific need for distribution. We will argue that they do not in fact do so but are instead addressing different (but related) problems.

In Chapter 5, we will evaluate existing distributed architectures, as well as distributed fault-tolerance.

2.2 Self-Monitoring Agents

A working prototype of a "complete" agent architecture with self-monitoring capability was developed by [Beaudoin, 1994]. A "meta-management" component is implemented as a layer for overseeing the planning and scheduling of multiple goals. For example, if a particular plan appears not to be making progress towards a goal, it may be interrupted.

Apart from complete agent architectures, examples of implemented self-monitoring systems (which could be part of a meta-level in an agent) include the following:

1. Reflective rule-based systems: For example EURISKO [Lenat, 1983]. There are two types of rule: domain rules that operate on mathematical concepts and rules that operate on other rules (meta-rules). Both types of rule have a strong similarity to our object-level and meta-level respectively. The meta-rules are a kind of meta-management. They control the search for new mathematical concepts by modifying rules. In contrast to our system, Eurisko meta-rules can modify all rules (including themselves) to enable the system to improve itself. Our architecture does not allow a meta-level to modify rules arbitrarily or delete itself (as would be possible with Eurisko). It can only repair and suppress rules in order to maintain its survival.

2. Meta-cognition for plan adaptation: [Oehlmann et al., 1993, Oehlmann, 1995]. The system monitors its reasoning during its attempts to adapt a plan to a new situation using its previous experience. This meta-cognition enables more efficient plan adaptation than would otherwise be achieved. The meta-level here involves complex high-level reasoning about similarities

between situations. Meta-cognition can also be interleaved with the plan adaptation process. In contrast to our work, the purpose of the meta-level is to improve the plan adaptation process and make it more efficient. It does not monitor the integrity of the system in a hostile environment.

3. Introspective reasoning to refine memory search and retrieval:
   for example [Fox and Leake, 1995, Leake, 1995]. This approach is similar to ours in that the system has a model of *expected behaviour* and *ideal behaviour* of its reasoning. The models are not acquired by learning, however. Instead discrepancies with models are used to guide learning with the aim of improving efficiency of memory search and retrieval.

All of the above mechanisms (including Beaudoin's) have the following limitations:

- They are all built on the basic assumption that the *architecture* of reflection is hierarchical with a single layer representing the meta-level. We need a non-hierarchical system.
- They do not have a concept of a "self" that has to be preserved intact in a hostile environment. One can say that in each case the software is not "situated" in an environment in which can be attacked and damaged.
- In general, the definition of "success" and "failure" (or good and bad "quality") is externally specified. We are aiming for some degree of discovery.

2.3 Computational Reflection

Computational reflection was introduced by Brian Cantwell Smith [Smith, 1982] and was first applied to object-oriented programming by [Maes, 1987]. Reflective programming architectures do not provide anomaly-detection. We consider these architectures here, however, because they can be designed to allow programs to access and modify their implementation without any restriction (apart from that imposed by the operating system and hardware). Hence, it may look as if such programs have no reflective blindness. We show that these architectures actually solve a different problem from the one that we require.

2.3.1 Meta-levels

The term "meta-level" is used widely in the reflective programming literature and normally means a level that specifies the semantics of another component, usually called its object-level. We will say that such a meta-level *implements* its object-level in that it *creates an instance* of it by "executing" or sometimes "interpreting" it. Such meta-levels exist in a normal computer architecture, although they are only called meta-levels in the reflective programming literature. For example, a processor creates an instance of an operating system. An operating system creates an instance of an application program; a simulating environment such as SIM_AGENT [Sloman and Poli, 1995] creates instances of agents by "executing" their specifications and enabling them to interact with a simulated world; a rule interpreter creates instances of a rule etc. If such a meta-level is not part of any reflective programming architecture, we simply call it an "implementation level" (since it is not normally regarded as a "meta-level" in such contexts). We use the term "implementation meta-level" for an interpreting meta-level in a reflective programming architecture. In contrast to an implementation meta-level, a *self-monitoring* meta-level is a component which actively monitors its object-level and may intervene to change its behaviour if necessary.

2.3.2 Reflective operators

To explain how reflection works, we imagine a hypothetical reflective programming system that allows application programs to enquire about their implementations and modify them. We will assume that the programming language is interpreted and that the same component (the interpreter) plays the roles of implementation and declarative meta-level. An application program is said to exist on the *base level* to distinguish it from the meta-level which interprets it. (Note that an object-level need not be the base-level. There may be a meta-meta-level, in the form of an interpreter which specifies the semantics of another interpreter).

For example, the base-level program may enquire about its normal method of evaluating logical expressions. To do this, it may call a function `inspect`. (This kind of function is often called "reify" in the literature). The program could modify its implementation by calling another function `modify`. (sometimes confusingly called "reflect" in the literature).

The functions `inspect` and `modify` may be understood as *reflective operators* which are available to a base-level program. Both functions give a program access to the way its linguistic expressions are *being interpreted*. This means that the interpreter can be treated as a modifiable program in the same way as the base-level program. To do this, a *second* implementation meta-level is required to interpret the changes applied to the first interpreter. This second level is activated whenever reflection is activated.

A common application of reflection is when a programmer wishes to extend a language, such as adding a new kind of assignment statement or changing the effect of list-processing operators. The second interpreter specifies the effects of the requested modifications and whether they are "legal" in exactly the same way as the first interpreter specifies the effects of base-level program expressions. The "self-representation" is the interpreter currently being operated on.

2.3.3 Virtual reflective towers

Some reflective programming architectures can be designed so that they allow any arbitrary meta-level to become an object-level on demand. Hence, anything that is created as an interpreter can be made available for inspection or modification. This appears to overcome the "blindness" problem, since in theory any meta-level can be made "visible".

The first programming architecture to make available any arbitrary meta-level was 3Lisp [Smith, 1982]. The idea is to provide an environment which behaves *as if* an infinite tower of meta-levels existed. The tower is "virtual".

A particularly interesting example is RbCl (Reflective based Concurrent language) [Ichisugi et al., 1992] which allows user modification of *all* aspects of the language, 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 language mechanisms such as memory-management can also be changed.

2.3.4 The SOAR architecture

Such an infinite tower could also be used by an autonomous system, provided there is a mechanism to prevent it getting trapped in an infinite recursion activating new meta-levels. The SOAR architecture may be described as a potential infinite tower of implementation meta-levels [Rosenbloom et al., 1988] which is available to a problem-solving agent. 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 arbitrary capability to inspect and modify its components 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).

However, if we return to the requirements for anomaly-detection and the concepts of "self" in Chapter 1, it is clear that SOAR does not provide these features. All such virtual infinite tower systems have the property that the meta-levels are not independently monitoring their object-level (that is, the components that they are inspecting). To analyse this problem, we compare self-monitoring with implementation meta-levels in more detail.

2.3.5 Self-monitoring meta-levels

A self-monitoring layer is also called a meta-level, since it "talks about" its object-level, although in a different way. To show this, we return to the hypothetical language extension scenario above which uses `inspect` and `modify` and show where a self-monitoring meta-level might fit in.

If we think of a program as a cognitive agent (as we wish to do), the level that actually does the reflection in the language extension scenario is the *second* level, because this is the meta-level that is *activated* when a reflective operation is carried out. Therefore, our concept of *self-monitoring* meta-level is a special case of an actively reflecting meta-level of this kind. In general, if we let M be a meta-level and P its object-level, the contrast between the two can be shown as follows:

- Implementation: M is a program that interprets P: it takes P as an argument.

- Reflective: P calls M to inspect/modify/monitor P

A meta-level that implements P may also be reflective with respect to a lower-level program whose meta-level is P. The first category cannot monitor the execution of P because it is taking P as a passive argument (an expression). In the second category a reflective meta-level may also be self-monitoring and able to intervene to correct or improve P's behaviour if required. However, it may not be very useful for a program P to call a procedure to monitor (and possibly change) P's behaviour. Two things are normally desirable for monitoring: namely *independence* and *concurrency*.

2.3.6 Meta-levels should be independent

For the purposes of intrusion detection, independence is particularly important for monitoring. An independent meta-level is not controlled and possibly not known about by its object-level and can be either reflective or non-reflective. Thus we can list two further types of meta-level relationship:

- Independent reflective: P contains a component M which is active independently of other components in P and inspects, modifies or monitors all components in P (including itself)

- Independent non-reflective: an independently active component M inspects, modifies or monitors a component P which does not contain M

The first category is our starting point when considering the reflective blindness problem. This is the situation where M cannot reliably monitor its own status. It *can* however, inspect and modify its own implementation by activating an arbitrary number of meta-levels using a reflective tower. It follows that a reflective tower does not remove the reflective blindness of self-monitoring systems.

Note that the difference between reflective and non-reflective may be vague if the boundaries of the system being observed are not always clearly defined. This may not be important for many software systems but for an autonomous self-defending system, a well-defined boundary is essential. We will come back to this issue in Chapter 9.

2.3.7 Concurrency is desirable

Concurrency is also desirable for independent self-monitoring and intervention. If we return to the example in Chapter 1 of walking downstairs, self-monitoring is concerned with the *currently running instance* of the program for walking downstairs, and not with the program code. This is why we are able to distinguish between expected behaviour (based on knowledge of what the program *should* do) and the actual behaviour which may be anomalous.

An independent reflective meta-level may run concurrently with its object-level or the same process may oscillate between object-level and meta-level code. For example, a rulebase may contain reflective rules, or a meta-level may be activated without the knowledge of a component in Y whenever the component calls a procedure (interception [Hof, 2000]). If there is a sequential oscillation, it must have a high enough frequency to detect problems before they can do serious damage.

Some reflective programming architectures do include concurrency, but entirely *within* a single meta-level. An example is [Matsuoka et al., 1991] where the component 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 L_{i+1} is an implementation meta-level for each member of L_i). A more recent example is [Mashuhara and Yonezawa, 2000].

2.3.8 Kornman's self-monitoring meta-levels

An architecture that might allow a meta-level to run concurrently with its object-level is Kornman's SADE [Kornman, 1996]. This is not a virtual infinite tower but instead incorporates a meta-level (of the "self-monitoring" 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 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. However, it only detects known failure patterns, and not anomalies as we have defined them (nonself).

2.4 Artificial Immune Systems

Artificial immune systems are designed to distinguish between self and nonself. A survey of immune-system inspired models can be found in [Dasgupta and Attoh-Okine, 1997]. A typical artificial immune system is composed of two main components: an algorithm for use in a protected training phase to develop a model of "self"; an algorithm for use in the unprotected "operational phase" which compares actual patterns with the model of "self".

For our investigation, two basic approaches are relevant:

- signature-based intrusion detection [Forrest et al., 1994]

- negative selection [Dasgupta and Forrest, 1996]

In *signature-based* approaches, a database of "normal" patterns is constructed during the training phase, which may be obtained by repeatedly observing the system while it runs under normal conditions, protected from intruders. The resulting database is the model of "self" called the "signature" of the system being protected. It contains characteristic patterns produced by the execution of all legitimate programs. There are many possible forms of recording such activity, e.g. file access patterns, user command sequences. The patterns are typically in the form of binary encoded strings which hold a record of a particular action or sequence of actions, e.g. Unix system call sequences [Forrest et al., 1996]. 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. Typically the comparison 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) and they should be regarded as insignificant fluctuations.

In *negative selection*, a random population of unique "detectors" is first generated. Each detector is a string representing a fragment of a possible execution trace (or other activity being observed). During the training phase, all detectors which match "normal" patterns are eliminated (hence the term negative selection). Thus if a detector matches some activity during the operational phase, the activity is anomalous. Negative selection has many advantages (e.g. it is more efficient than signature-based methods because anomaly-detection only involves the matching of a single detector, while a signature requires a more global comparison).

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, while the whole system is the object-level. If the object-level "contains" the meta-level the patterns produced by the algorithm itself are included within the signature for the whole system (the object-level). It may detect anomalies in its own operation in some cases. However, if the algorithm has only one instance, there is no remaining immune-system component that can reliably detect anomalies in the algorithm's execution patterns.

If the algorithm has more than one instance, it is necessary to consider the way in which the instances interact and their interrelationships in a software engineering sense. This is what we call the "architecture level" and will be defined more precisely in Chapter 3. A detailed analysis of possible interrationships between instances is not done in the existing AIS literature at the time of writing. For example, although negative selection is suitable for distribution over multiple sites, where each site can detect a subset of "nonself", they do not specify any requirements for the interaction between them, although it is possible that distributed forms of an AIS may implicitly reduce vulnerability by providing more general fault-tolerance and redundancy. Such a distributed AIS is presented by [Hofmeyr and Forrest, 2000]. We will return to this issue in Chapter 5.

Most AIS work is focused on the properties of the algorithms, and not on the whole architecture in which the algorithms are embedded as components. Regardless of how powerful the algorithm is that carries out the functionality of a component, it will not help if the component is not protected by the architecture.

2.5 Vulnerability analysis and defensive programming

How do we know that existing fault- and intrusion-tolerant systems have not already solved the problem in an implicit way without mentioning "reflection" or "autonomy"? The problem of giving an intrusion detection system self-monitoring capabilities to resist attacks against itself has been recognised but not yet solved satsfactorily (see for example [Cachin et al., 2000a], page 18). The problem is usually addressed by analysis of vulnerabilities and using defensive programming to make it difficult to exploit these vulnerabilities. For example, in the intrusion detection system *Bro* [Paxson, 1999], the following measures are used to defend against particular attacks on a monitoring system:

- overload attacks: if volume of data stream is suddenly increasing, reduce any "irrelevant" data being monitored.
- deliberate crashing: a logging system detects an abnormal exit of the monitoring system
- deliberate misleading of the monitoring system ("subterfuge"): analyse assumptions made by the system and use defensive programming to prevent them being exploited.

These measures do not involve detection of anomalies in the monitoring system, but instead rely on the designer knowing in advance about vulnerabilities and possible kinds of attack. Defensive programming in intrusion detection is an example of a more general class of fault-tolerance measures that are normally added as new features to an existing architecture without questioning the limitations of the architecture itself. Distributed fault-tolerance is reviewed in Chapter 5.

2.6 Summary and conclusions

This chapter has reviewed some related work on different kinds of "reflection" but without distribution. The categories do not satisfy our requirements for the reasons given:

- Self-monitoring agents: they assume that the architecture is hierarchical, with normally only one meta-level (Section 2.2). The agent does not question or modify the rules of its meta-level. Therefore it is reflectively "blind" as defined in Section 1.4.
- Computational reflection: although some languages (such as 3Lisp) can allow the system to inspect and modify its internal details arbitrarily, the reflective operations cannot be used for anomaly-detection because they do not independently monitor the behaviour of the system and compare it with expectation (Section 2.3.5). Computational reflection does not use a model of expected (or desired) behaviour and therefore cannot detect "nonself".
- Artificial immune systems: they do have a model of "normal" behaviour and can detect "nonself". However, their primary focus is on the anomaly-detection *algorithm* without regard for the architecture in which it is embedded (Section 2.4). There is no requirement for the algorithm to be a component within the system being monitored or for its behaviour to be included in the definition of "self" (as required in Section 1.4). "Self/nonself" distinction is only regarded as a distinction between any two classes.
- Vulnerability analysis and defensive programming: this does not enable the system to detect anomalies, although it is useful for fault-tolerance.

A more detailed discussion on fault-tolerance is in Chapter 5.

Chapter 3

Proposed Solution

3.1 Overview

This chapter introduces the conceptual framework for "working around" the problem of reflective blindness. The first part introduces distributed reflection and the special case of "closed" reflection. The second part looks at some related work in philosophy and biology.

3.2 Distributed Reflection: Conceptual Foundations

To compensate for reflective blindness, we 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. Since multiple agents run concurrently and independently of each other, the desirable features for monitoring and intervention stated in chapter 2 are satisfied.

3.2.1 Terminology

In this thesis we use the term "agent" to mean the highest level of component in a design that is both *sequentially controlled* and *hierarchically organised*. In practice this means that there is a sequential process that has "authority" over all other aspects of the agent. We also assume that an agent has a representation of the world (and itself), and that it has reasoning capability.

We use the term "architecture" to mean any software design involving interacting components. We use the term "component" for any software module that does not contain concurrently executable subcomponents. For example, a component may be an agent, a subroutine or a collection of related rules in a rulebase. We assume that anything that is a meta-level or object-level is a component, and that all components play one or both of these roles.

The following taxonomy of terms have been developed by [Sloman, 2001] at Birmingham:

- *Architecture Schema (AS)*: constraints on components and their interrelationships; what kind of algorithms and other components are required? What are the constraints on their interaction? Are there any constraints on how they should start themselves up? Is there any learning or adaptation involved? An example of a schema the Birmingham *CogAff* schema.

- *Architecture (A)*: a specific application of an architectural schema to a set of components. How should the components be mapped onto instances (executable entities)? How do the instances interact? An architecture can be translated directly into an implementation (but clearly there may be several implementations of the one architecture e.g. in different programming languages). An example of an architecture is the Birmingham *H-CogAff* architecture, which is a special case of *CogAff*.

- *Architecture Implementation (AI)*: set of executable components written in one or more languages and allocated to one or more processors.

- *Instances of an architecture implementation (AII)*. Different instances of the same implementation may exist; for example, they may run on different networks.

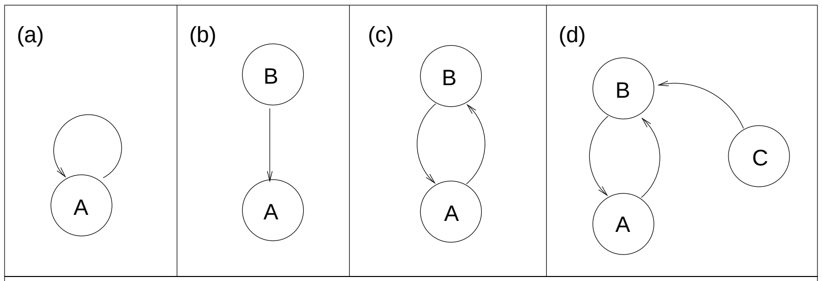
An architecture (before it becomes an implementation instance) already contains component instances (processes) because the architecture must specify the relationships between processes during execution. However, for abbreviation we use the term "component" for an (active) architecture component as well as for a passive code component.

Any bootstrapping constraints specified in the architecture schema must be applied to the architecture. For example, the whole architecture may be bootstrapped by activation of random components at random times, or the components may activate each other or gradually increase each other's effectiveness. Some of these constraints may apply to a particular implementation only, but some may be specified on the architecture level because they may be essential for the required function of the system. (We will be particularly concerned with mutual bootstrapping in later chapters).

Constraints on relationships between components are informally specified. They may become formalised at a later stage, when there are many working instances of the architecture being applied to real-world problems.

3.2.2 Reflective networks

Possible configurations (architecture schemas) of agents are shown in figure 3.1. Each circle in figure



(a) A's meta−level monitors itself; (b) "open" hierarchical reflection: a second agent B monitors A's meta−level; (c) closed distributed reflection: A and B monitor each other's meta−levels; (d) open distributed reflection with an additional agent C.

Figure 3.1: Configurations for reflective networks

3.1 is an agent. An arrow indicates the monitoring relationship (and also includes capability to modify). Implicit in each agent is a meta-level that actively does the monitoring. The arrow also means "equally distributed" monitoring; that is, for any agent that is monitored, *all* of its components are monitored in the same way (including its meta-level).

Figure 3.1(a) shows the "centralised" configuration where a single agent detects anomalies in its own execution (which we have argued is not reliable). (b) and (d) are examples of "open" reflection. A configuration is "open" if at least one agent exists that is a source of an arrow but not a destination. Configuration (b) is *hierarchical* because A is unambiguously *subordinate* to B, but we will not discuss hierarchies in detail until the next chapter. (c) is "closed" meaning that all meta-levels are also object-levels (they are observable and modifiable from within the system itself). Later, more precise definitions of these schemas will be given. For the moment, we summarise features that exclude each other in Table 3.1. For example, according to our definition, an architecture is either "hierarchical" or "distributed". "Hierarchical" means that there is always one agent that has final "authority" in determining the system's actions, because its model of the whole system is not questioned. "Distributed" means that there are at least two agents whose models of the system have equal "value", thus giving both agents the same "authority" (We will consider in Chapter 10 what "authority" and "equality" mean in practice).

centralised	distributed
hierarchical	distributed
open	closed

Table 3.1: Mutually exclusive properties

The following combinations are unreliable:

1. Centralised and closed (one agent monitors all of its components)
2. Centralised and open (one agent monitors some of its components)
3. Hierarchical and open (for two agents, one agent monitors another)

There is no fundamental difference between (2) and (3) above, since A and B in (3) can play the roles of meta- and object-levels in (2).

Hierarchical reflection is *not* distributed because the same agent is always in control. In other words, its viewpoint of the whole system is the only one that determines the system's actions.

In this thesis, we are interested in architectures that are both "distributed" and "closed". Note that in a two-agent system a distributed system is always closed. For convenience, we call such a configuration a *closed reflective network* or CRN. We now define closure more precisely as a design constraint.

3.2.3 Closed reflection as a design constraint

We can specify the constraints (architecture schema) informally for closed reflection as follows:

Constraint 1: There is no component in the system that is not subject to monitoring and modification by the system itself. In particular, the meta-level components are subject to *at least as much* monitoring as all other components.

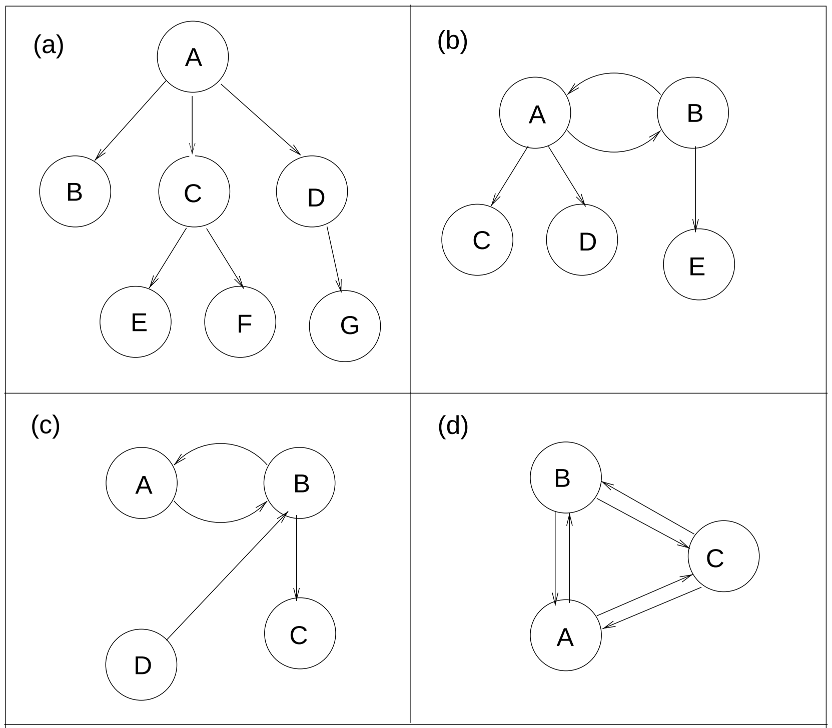
Constraint 2: Monitoring and modification should include a minimal capability to continually observe a component's current state, to detect deviations from normal behaviour (anomalies) and to modify or influence the operation of the component in response to an anomaly.

Constraint 3: a model of normal behaviour should be acquired by mutual observation (bootstrapping requirement)

We emphasise that the constraints are informal in that they are guidelines for the design process. The highest level component is an agent (which is always a sequential process in our work). A component can be defined on arbitrarily low levels depending on the monitoring accuracy required by the particular environment.

The definition of "minimal" capability depends on the type of environment the system must survive in. For example, if the environment demands precise timing of actions and contains enemies that can modify the timing in subtle ways then a minimal capability would involve precise measurement of "normal" timing patterns.

An open reflective architecture is one for which constraint 1 does not apply; there is at least one executive or control component that is not subject to monitoring or control by the system itself. In other words, there is *no* design objective to apply self-monitoring resources to *all* components that are themselves involved in self-monitoring and recovery, as there is in closed reflection. Examples involving more than 2 agents are shown in figure 3.2. Figure 3.2(c) is an example of an open distributed



(a) A simple hierarchy; (b) closed distributed reflection with two agents, each with subordinate agents; (c) open distributed reflection: D is not monitored, but it monitors B; (d) closed distributed reflection with three agents and full connectivity.

Figure 3.2: More complex reflective configurations

architecture. A, B and D have different views of the whole system, none of which is subordinate to any other (therefore it is distributed). However D is not itself subject to monitoring, although it monitors another agent. In general, if there is any *mutual* reflection, the architecture is distributed.

3.2.4 Why is this called "reflection"?

Since we are discussing a network of agents, one may ask why it is still called "reflective". Why not just call it mutual monitoring and intervention? A network of mutually monitoring agents may be "closed" but not be reflective. It depends on the nature of the "model" mentioned in Constraint 3. If the model is part of a causally connected self-representation (which we intend) then it can be called reflective (Section 1.2.1).

Furthermore, the aim is to allow the same relationships *between* agents that we have *within* a single agent. If we call an agent "reflective" when it reasons about its own rule firing patterns then the *network* of agents is reflective if one of its agents can reason about the rule firing patterns of another agent. The only difference is that one kind of reflection is "horizontal" and across agent boundaries while the other is "vertical" and within an agent. If the reflection is "closed" then the network is "collectively" reasoning about its reasoning. We will come back to this issue in later chapters.

3.3 Related Work in Biology and Philosophy

One problem with the above architecture schema is that it does not specify constraints on interference between mutually monitoring agents. A system of this kind may allow agents to interfere with each other arbitrarily, which seems to support the counter-argument of Minsky's "student" in Section 1.3. To answer this question we look in more detail at related concepts developed independently by biologists Maturana and Varela and argue that they support the "control system" paradigm.

3.3.1 Maturana and Varela's Autopoiesis Theory

The concept of a closed reflective network in 3.2.2 is an interpretation of Varela's *organisational closure* as it relates to biological autonomous systems [Varela, 1979]). Varela's concept is based on the theory of *autopoiesis* [Maturana and Varela, 1980], which means "self-production" ("auto" = self, "poiesis" = make). The theory provides a new set of concepts (ontology) that emphasises the *circular* nature of the architecture of living systems (which we have above). However it also emphasises the ability to *maintain their integrity* in a hostile environment (which they call "perturbations").

Varela's concept of *autonomous* systems is more general than autopoiesis and can be interpreted as "self-maintaining". According to Varela such systems *require* organisational closure:

> **Closure Thesis:** *Every autonomous system is organizationally closed.*

(The opposite may not apply: not every organisationally closed systems needs to be autonomous).

Autopoiesis describes a living system as a structure similar to figure 3.1(a) with the difference that the internal workings of the "agent" is not a single thread but contains multiple processes that "bootstrap" each other. On page 55 of [Varela, 1979] these requirements about the internal structure of a closed system are stated as follows:

> We shall say that autonomous systems are organizationally closed. That is, their organization is characterized by processes such that (1) the processes are related as a network, so that *they recursively depend on each other in the generation and realization of the processes themselves*, and (2) they constitute the system as a unity recognizable in the space (domain) in which the processes exist. (Emphasis added)

In other words, organisationally closed systems are generated and maintained by the mutual interdependence (bootstrapping) of processes that constitute them. This implies that *arbitrary* interactions between processes does not happen, but instead the interactions *always* have the purpose of maintaining the integrity of the system. To support this, [Maturana and Varela, 1980] page 48) have defined a living system as follows:

> "... a living system is a homeostatic system whose homeostatic organization has its own organization as the variable that it maintains constant through the production and functioning of the *components* that specify it.... It follows that living systems are a subclass of circular and homeostatic systems."

Therefore the components should take action in a "homeostatic" sense in order to correct negative effects of the environment (known as "perturbations"). In a biological system, this means constant "production" of components that are being continually damaged by the environment. In an artificial system, we will have to define this in terms of required function of the system.

3.3.2 Rosen's "Closure to Efficient Causation"

A similar model of closure was independently developed by Rosen [Rosen, 1991] called "closure to efficient cause". On page 244 he defines a living thing as follows:

> *...a material system is an organism if, and only if, it is closed to efficient causation*. That is, if f is any component of such a system, the question "why f" has an answer within the system, which corresponds to the category of efficient cause of f. (Original emphasis)

In Rosen's terminology, the category of "efficient cause" corresponds approximately to "production". Component f exists because something within the system itself is actively producing f. Therefore the system is producing all its own components. Hence Rosen's closure concept appears to be just another way of defining autopoiesis (self-production). The main problem is that it depends critically on the definition of "efficient cause" and this is not defined precisely; it is simply one category out of four types of Aristotelian causality. The other three are "formal", "material" and "final" causality respectively.

Rosen seems to be implying that "efficient cause" is similar to execution of a program by a processor. In this case, the "component" is the instance of the program. The program behaviour (function) is "efficiently caused" by the processor. The program code itself is a "formal" cause and the data is a "material" cause for its behaviour (the final cause would be the requirements). Rosen does not seem to consider whether these categories continue to apply on the different *virtual* levels of a computer system. There is no reason why they should not, however, since the notion of efficient causation is very similar to that of an implementation meta-level.

For a computer system, closure to efficient causation would mean that every component (including the hardware) is efficiently caused by something within the computer system itself. The hardware would be "instantiated" ("produced") and maintained by one of its programs (efficient cause), which is itself executing something similar to a "genetic code" (formal cause). A less radical version would involve the program being able to modify the hardware (as in neural plasticity).

One of the most important issues here is that a single component plays different roles simultaneously. The hardware is an efficient cause for the program (an implementation meta-level) but the program is also an efficient cause in "producing" the hardware (it is acting as a different implementation meta-level).

Since our architecture has to be simulated, we have to ignore implementation meta-levels, but we use a similar architecture with self-monitoring meta-levels instead. This only means that our architecture cannot be a strict "autopoietic" system (if we assume that an implementation meta-level corresponds to efficient cause or "production"). Instead it should correspond to Varela's more general notion of an *autonomous* system. That means "self-maintaining", not necessarily "self-producing".

3.3.3 Kaehr and Mahler's distributed meta-levels

[Kaehr and Mahler, 1996] have produced an architecture that is very similar to Rosen's closure concept although developed independently. They attempted to formalise Varela's organisational closure using a notation invented by Günther [Günther, 1971]. We will not discuss this notation in detail, but we will summarise their model from the point of view of concurrency and interactions between processes.

Kaehr and Mahler's general 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. (Günther originally called his notation a "logic", but not in the AI sense of fuzzy or modal logics). It essentially allows a number of formal logical systems to exist in parallel. (This also puts into question the applicability of Gödel's theorem to any artificial computational system, but we do not discuss this here).

In the computational interpretation, a key idea is that something that 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 process is a concurrent thread. Each process takes the one below it as an *operand* (that is a parameter) and "evaluates" it, meaning that the nature of the evaluation itself is changing while the state of the operand is changing. (A process does not wait for its operand to complete executing and return a result).

This can be easily expressed in the language of Kornman's meta-levels: we have meta-level L_2 operating on L_1, which in turn operates on L_0. In Kornman's architecture, the meaning of "operating" is specific: the meta-level monitors the state of rule-firing of its object-level and could modify its code in response (to produce a "desired" rule firing pattern).

Organisational closure is implemented by making the code for L_2 the same instance as the code for L_0. We can say that L_0 and L_2 are identical. Therefore we only need two meta-levels L_1 and L_2. Their actions can be described as follows:

"L_1 monitors L_2, which monitors L_1, which monitors L_2, which ..."

They have to monitor (and possibly modify) each other *concurrently* without having to wait for each other's execution to complete; otherwise nothing happens because of deadlock. In the ideal situation, their monitoring should also be *independent* in that they are *not* both regulated by a central scheduler or management program of any kind. As this is difficult to achieve in practice (on a single processor at least), our distributed architecture has to be simulated in a virtual environment.

The main observation about this model is that it also supports the idea of the same component playing different "roles" simultaneously, and is thus similar to Rosen's concept of closure to efficient causation.

Kaehr and Mahler's model is impractical in its current form. It is a "chaotic" model of computation written in ML. In the next chapter we consider how to apply it to the framework of autonomous systems that we require.

3.4 Closed Reflection and self/nonself distinction

A closed reflective network has the potential to distinguish *collectively* between self and nonself because a deviation from correct operation (self) by *any* part of the system can be recognised by some other part of the system without external intervention. This seems simple. However, if we look more closely at Varela and Rosen's work above, it becomes clear that there are two kinds of boundary between "self" and "nonself". Autopoiesis theory implies that the system treats its own components differently from other objects (because it maintains their integrity but it does not maintain the "integrity" of environmental objects). Therefore it implies that a boundary is generated between its own components and the environment. There is a second distinction, however, in that these components must also distinguish between "integrity" and "non-integrity".

The first distinction is represented as the "component map" introduced in Section 1.2.3 and required by constraint 2 in Section 3.2.3 in order to monitor the current status of each component. The second distinction is defined by the model of normal behaviour of each component. Anything that is not represented in the component map will not be monitored and will simply be ignored (unless it is represented as a part of the "external world".)

In living systems, the two kinds of boundary may coincide in some cases, where the "maintainance of integrity" is approximately the same as preserving the continued existence of the system's components (because they would disintegrate physically if they were not functioning correctly).

In contrast, an artificial autonomous system is (usually) not continually regenerating its physical components in the presence of a damaging environment. This means that the boundary between the physical components of the system and the environment may be something very different from the boundary between “correct” behaviour of components and incorrect behaviour (since correct behaviour is not identical to the continued physical existence of the compoments).

Thus, we set the following requirement for components distinguishing between “self” and “nonself”:

1. they must be within the system and represented in the component map;
2. their own correct behaviour must be included within the set of correct behaviours; if their own behaviour is not “correct” they must also be rejected as “nonself”
3. they must trigger a self-protective response when “nonself” is detected (i.e. diagnosis and repair)

Note that condition 1 is just a reformulation of constraint 1 for closed reflection (Section 3.2.3). If condition 1 is not true we have a situation where an external component is making a distinction between one class and another within the system. This would correspond to an “open” system in which there is one agent that monitors the system but is not itself monitored by anything within the system. Note also that condition 1 is not satisfied by artificial immune algorithms, since they do not have the equivalent of a “component map”. If condition 2 is not satisfied, the components’ decision may be incorrect. Condition 3 may be a reactive nonspecific response or it may involve reasoning about specific components.

In this thesis, we are primarily concerned with learning the boundary between “correct” and “incorrect” behaviour (condition 2). We will return to the relationship between the two boundaries in Chapter 10.

3.5 Summary and Conclusions

In this chapter we have introduced the concepts for distributed and closed reflection. We have also identified the problem of arbitrary interference if there is no clear hierarchy of “authority”.

The main conclusion of this chapter is that the control systems paradigm is a useful way forward in defining an implementable architecture. Maturana and Varela’s autopoiesis theory emphasises that circularity is acceptable and important in biological systems, as does Minsky (Section 1.3). However, both autopoiesis and Rosen’s theory tell us something additional: the circularity serves the “maintaining constant” of the system by its own components (in that it is continually being “produced” while undergoing damage (perturbations). Therefore the “interference” between components is not arbitrary if we specify the conditions in which they can “interfere” (namely, when they have to return the system to a satisfactory state). In the next chapter we will make this more precise.

The non-arbitrary nature of “interference” also has consequences for understanding self/nonself distinction (Section 3.4 above). Namely it has the form of re-asserting “self” when “nonself” is detected.

One feature of some of the work reviewed above (Rosen and Kaehr) is that they require the same component to play different roles simultaneously (rather than increasing the number of components). We will see later that this feature is an advantage in the actual implementation.

3.5.1 What remains to be done?

In the next chapter we will move towards implementation. In particular, we have to address the following:

- apply the control system paradigm - what exactly can “maintaining of system integrity” mean?

- ensure that the resulting architecture is closed (and hence can distinguish between self and nonself)
- ensure that it is reflective in the way that is required (with a component map and model)

Chapter 4

Towards an Architecture

4.1 Overview

In this chapter, we describe the first stages of developing the schema of closed reflection into an architecture using the paradigm of control systems. While the last chapter was mainly about the overall *structure* of reflection, this chapter is concerned more with the *content* of the reflection: what should each agent be "concerned" about?

The chapter also outlines the methodology and assumptions on which the architecture development and later implementations of it are based. (Implementations are described in Parts 2 and 3). The notion of a "design space" is used to show a selected subset of architectures that satisfy the closed reflection schema.

4.2 Control Systems

To further develop the architecture, and in accordance with the conclusion of Chapter 3, we use the control system paradigm for cognitive systems as outlined in [Sloman, 1993a].

We define the *primary task* of an autonomous system as the set of user-specified requirements it must satisfy (e.g. observe something and take measurements). The primary task corresponds to the aspects of the functionality that a user is concerned about. We expect the specification to be encoded in an ontology that fits in with the user's work (and may even involve multiple ontologies for multiple user groups). For example, a primary task in the medical domain would be specified using terminology and concepts from that domain, and the system's actions should be explained using those concepts.

Note that a primary task is only defined for autonomous systems that act (at least partly) on behalf of human concerns (for example autonomous vehicles). There are situations where such a task may not be defined (for example an Alife scenario such as artificial insects) but we are not considering those systems.

For a system that does act on behalf of human concerns, we can define its *survival* as the capability to satisfy the primary task in the presence of faults and intrusions without external help. This is the equivalent of the "integrity" of the system discussed in Chapter 3.

We define a *hostile environment* as one in which the system's executive and control components (*including* any anomaly-detection and self-repair mechanisms) can be disrupted or directly attacked. Examples of simulated hostile environments are presented in Part 2 of the thesis.

4.2.1 Concern centred approach

An autonomous system that acts on behalf of human concerns may be regarded as a control system in that it maintains the world in a human-desired state. The system itself can be said to have "concerns" in the sense implied by Frijda [Frijda, 1986], page 335 as *dispositions to desire the occurrence or non-occurrence of a given kind of situation.*

These concerns can be defined as the system's mechanisms for ensuring that a user's critical requirements are not violated. If all concerns are externally specified, the system is like a homeostatic system (in the same way that a thermostat does not allow temperature "requirements" to be violated).

As an example, we can consider a hypothetical network management system that ensures that faults and intrusions do not degrade network response-time, or do not violate data access constraints such as privacy. We may represent user rights as required states of the world that should be preserved, even if a privileged user requests something that contradicts them (since passwords may be stolen). This could mean, for example, that a Unix user with the "root" password is refused permission to delete the files of other users, on the grounds that it violates data access requirements (or put another way, it would degrade the quality of the world to an unacceptable level).

Such a simple "homeostatic" system is very brittle, however, since an attacker may first disable the part of the software that ensures that user rights are not violated. Although the system may *detect* that the quality of the world deteriorates, it cannot diagnose the problem or take any autonomous action to correct it. To overcome this, we need an additional level of desirable state relating to the system's capability to continue operating. We now have two levels of concern:

- desirable states relating to primary task
- desirable states relating to performance and integrity of the system (relating to software and hardware), which may be called self-protective concerns.

This two-layered concern structure is also implied by autopoiesis theory [Maturana and Varela, 1980] (page 13) as well as the philosophy of Second Order Cybernetics in general [von Foerster, 1981], where "second order" refers to the second level that "regulates the control system itself".

The precise content of the desirable states for self-protection is not expected to be externally specified because this content refers to internal states of the software and it is unlikely that these states could be known in advance by a user, unless the software is trivial. An acquired tendency to preserve a state that was not externally specified is often called an "emergent" concern. An architecture for emergent concerns is presented in [Allen, 2000].

As mentioned previously, the *acquisition* of the model of normal behaviour is a central part of the problem, and will be discussed in Part 2. This acquisition has similarities to the acquisition of a database of "self" strings in artificial immune systems, although we may look for higher level regularities using a method similar to data mining.

A self-protective system should defend its software and internal data from unauthorised changes and ensure that it has sufficient computational resources to do its task. Similarly, it should monitor how it uses these resources: does it spend most time and resources on the most relevant aspects of its problem domain? E.g. it should recognise when it is being swamped with meaningless data and false alarms that "distract" it from doing productive work. This is related to the concept of "meta-management", see e.g. [Sloman, 1998a].

For convenience, we call the internal state of the system its *internal world*, in contrast to its external world.

4.2.2 A single agent control system

We can now make the CRN architecture introduced in the previous chapter more precise by specifying how the *content* of the reflection relates to the distributed configuration. Figure 4.1(a) shows a single agent as a two-layer structure containing an object-level and a meta-level respectively. The object-level is a control system C_E for an external world as shown schematically in figure 4.1(a).

We assume that the agent has a model M_E of the external world to maintain the environment within acceptable values. We assume that M_E is represented in a symbolic declarative form (e.g.

rules) and enables the agent to predict the outcome of different actions. The model is part of its *representation* of the world, which also includes the actual states.

The agent selects its next action according to the quality of the predicted state. Thus M_E allows the "simulation" of hypothetical states ("what-if" reasoning), which is often called "deliberative" processing (see e.g. [Sloman, 1997]) in contrast to reactive processing, which involves immediate reactions to the external world without the use of a predictive model.

4.2.3 Internal control system

The simplest way to introduce reflection (and hence self-protection) is to add a meta-level labelled C_I as shown in figure 4.1(a). The meta-level is like a second control system, which is applied to the agent's internal world (i.e. aspects of its own execution) to maintain its required states (hence the label C_I). In the simplest case, the model M_I only predicts that the internal world will remain "normal" and its "quality" is its similarity to the expected state. M_I is autonomously acquired by C_I during a protected training phase. If a component can be identified that is behaving in a different way from expected, C_I takes action to "repair" it so that its behaviour is again "normal". (Methods of doing this will be introduced in Part 2).

Note that sensors and effectors are also used on the meta-level (S_I and E_I). We call the two layered structure a *reflective agent*. Its structure is similar to the agent concept supported by the sim_agent toolkit [Sloman and Poli, 1995], which we use for exploring architectures. Note also that this is similar to Minsky's A-Brain being monitored and modified by a B-Brain (as introduced in Section 1.3).

In a more complex version of C_I, an unexpected state of a component can be tolerated so long as it does not interfere with the quality of the environment or the quality of the software performance. (We will present an implementation of this in Part 3). In a still more complex version, C_I may be continually experimenting with alternative components to find one that is "optimal" in a particular situation. This version of C_I would always be "striving towards" an optimal state (self-improving software) rather than simply being "satisfied" with an acceptable state.

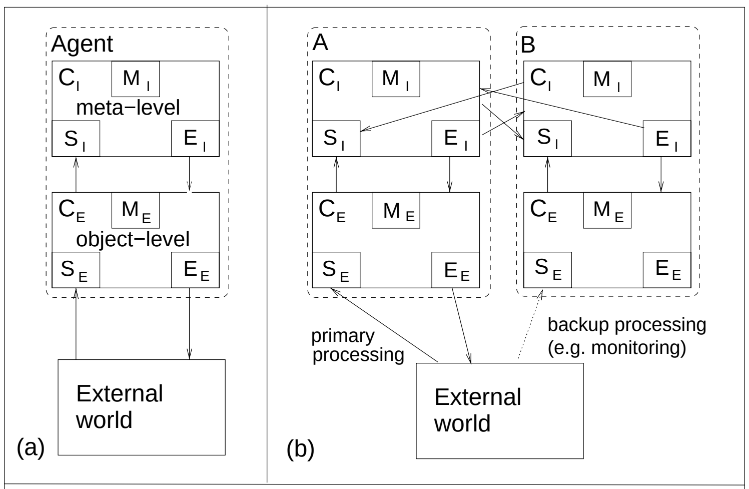
The reflective blindness problem becomes apparent in that C_I cannot reliably detect anomalies in its own operation (for example, if C_I is deleted or prevented from executing). The internal sensor readings in figure 4.1(a) would normally include the execution patterns of C_I itself. However, any anomaly in C_I's execution patterns may mean that the anomaly-detection itself is defective. Similarly if the internal sensors are defective, and they leave an anomalous execution trace, these same defective sensors would be used to read the trace.

4.2.4 Distributed reflection and control

Figure 4.1(b) shows one way in which the reflection can be "distributed" to produce a closed network. The reflection is closed because the meta-level of agent A is the object-level of agent B and vice versa (although it is difficult to use these labels in the diagram). Each agent uses its internal sensors and model of the expected state to monitor the meta-level of the other. It also has the capability to intervene using internal effectors if necessary.

The monitoring is independent because an agent does not control the way it is being monitored. It should also be concurrent; the agents can run on different processors or the same processor. Hence the requirement of all meta-levels being simultaneously object-levels of other agents within the system is satisfied.

Although all meta-levels are object-levels the converse is not true. The box labelled "object-level" in figure 4.1(a) is the component responsible for the *primary task* and acts as the "absolute" object-level or "application-level". It acts entirely within a user-visible external world. The meta-level is responsible for the secondary task of maintaining the object-level's ability to perform the primary task.



C_I	control of internal world	C_E	control of external world
M_I	model of internal world	M_E	model of external world
S_I	internal sensors	S_E	external sensors
E_I	internal effectors	E_E	external effectors

(a) Single−agent reflection; (b) Distributed reflection (simplest case): agent is divided into primary and backup copies A and B. Only the primary changes the external world; both have read/write access to each other's meta−levels and their own object−levels

Figure 4.1: Reflective agent architectures

4.2.5 Causal connection and control

At this point we return to the ongoing comparison with reflective programming. As mentioned in Chapter 1, reflection does not just mean passive self-knowledge, but also the ability to modify aspects of the system's own behaviour. This is also the case for reflective programming architectures.

The concept of a *causally connected self-representation* (CCSR) briefly referred to in section 1.2.1 is important in reflective programming. We now ask: does the self-representation of a reflective *control* system have the same causal connectivity as a self-representation that is accessed by a program using reflective operators of the `inspect` and `modify` type?

The control system equivalent of `inspect` is provided by the internal sensors, which if accurate, should give a representation of the actual state of the system. (Monitoring of sensor operation is also included in the monitoring of a meta-level).

To ensure that the system is self-modifying as well as self-monitoring we require the equivalent of a `modify` operator. For this purpose the self-representation should include a map of the system's components so that modifying the map will modify the behaviour of the system. For example, if the system removes a component from the map, the execution of the component should be suppressed in reality. (See also Section 3.4). A self-representation that enables anomaly-detection *and* causal connection has the following partitions:

- Map of components that allows the system to suppress, activate or modify their actual execution as required.
- Representation of normal behaviour of components (model)
- Representation of actual behaviour of components (internal sensor readings)

As described briefly in 1.2.1, the system's internal component map can also be *inspected*. However, this will not give the same information as the internal sensors. It only gives an accurate picture of the system's *implementation*, not its current behaviour with respect to the environment. It is not the "sensory" causal connection of a control system.

With a control system of the above type, there is also a third causal connection between the desired state and the system's actual behaviour (in the above architecture the desired state is the expected state). However, in contrast to a reflective programming operator (such as `modify`), this type of causal connection is not guaranteed to work perfectly. The difference is in the existence of an environment. For example, the suppression of a component may be subverted by a hostile intrusion. The causal connection is in the form of a *concern* - a tendency or pressure towards certain corrective actions.

4.2.6 Symmetrical control relationships

We now return to the issue of hierarchies touched on in the last chapter when introducing closed reflection. We use the term "hierarchy" in the sense of control. Roughly, if A can control B but B cannot control A then A and B have a hierarchical relationship. In other words the relationship is *asymmetric* with one being clearly subordinate to the other. "Capability to control X" means "able to maintain X in a satisfactory state", as defined for control systems above. For simplicity we use a discrete boolean understanding of "capability" where either the capability is there or not.

Since our control system is a two-agent network, closed reflection requires an approximately symmetric (that is balanced) control relationship between the two agents. They are *non*-hierarchical. This follows because closure requires that all agents should be observable and modifiable from within the network. The situation is more complex for networks with more than two agents and will be considered in part 3.

"Capability" is only one of several forms of control relationship. For example, we might specify that A has authority (permission) to control B or A has an obligation to control B as part of a requirement specification. These kinds of relationship may appear on different levels of the software development process. They may also emerge during the bootstrapping process as defined in Section 3.2.1. In the CRN architecture we assume that there is an *obligation* for agents to protect each other (given the capability).

The capability is gradually bootstrapped during execution of an implementation instance. The *method* of bootstrapping is specified at the architecture level with possible additions at the implementation level (e.g. depending on language, operating system etc.). In Part 2 we present a real implementation based on the closed reflective architecture, which demonstrates this principle.

An implementation instance of an architecture with symmetrical control relationships (a non-hierarchy) could evolve into a system that is in practice a hierarchy, because an agent may become unable to satisfy its obligation effectively. This situation should be prevented by the architecture because it would violate the requirement that agents are obliged to be mutually protective (and corrective).

In this thesis we assume that each agent has authority to override the actions of the other and is obliged to do this when necessary. This is a special case of a non-hierarchical system. In Part 2 we will consider what this means in practice, when describing a real implementation.

4.3 Methodology and Assumptions

4.3.1 Design-based approach

Our methodology is *design-based* [Sloman, 1993b]. The objective of a design-based strategy is to improve understanding of a phenomenon by attempting to build it (using software engineering). It involves the exploring of “design-space” and “niche-space”. A “niche” is a set of requirements a design has to satisfy. In our case, the requirement is to survive in a particular kind of environment. More detailed explanations are in [Sloman, 1998b] and [Sloman, 1995].

4.3.2 Broad-and-shallow architectures

Our approach is also “broad and shallow” [Bates et al., 1991]. A *broad* architecture is one that provides *all* functions necessary for a complete autonomous system (including sensing, decision-making, action, self-diagnosis and repair).

A *shallow* architecture is a specification of components and their interactions, where each component implements a scaled-down simplified version of an algorithm or mechanism that implements the component’s function. We assume that a shallow component can be “deepened” by replacing its algorithm with a more complex and realistic version. The shallowness is a practical necessity in the early stages of design and exploration.

4.3.3 Informal rapid prototyping

The design process need not initially be constrained by a formal definition, although a formalisation may gradually emerge as a result of increased understanding acquired during iterated design and testing. Such an emergent formal definition may, however, be very different from an initial one used to specify the design at the beginning. For example we do not define precisely what a “software component” is, since the definition of a “component” evolves depending on the kind of software being designed. In other words, we accept a certain degree of “scruffiness” [Sloman, 1990] initially.

For this reason, the initial results of the investigation are not quantitative, but instead answer the following type of question:

> “What kind of architecture can survive in what kind of hostile environment?”

It is basically a process of “matching” architectures to environments (or matching “designs” to “niches” as they are called in [Sloman, 1995]). The result is an existence proof stating that a distributed reflective architecture exists with matches at least one kind of environment that can be defined as “hostile” according to the definition in 4.2. The different classes of hostile environment are defined according to restrictions on the types of hostility that can occur. Examples of restrictions will be given in Chapters 6, 8 and 9.

For our investigation, a large part of the problem is designing an environment that will test the distributed reflection and quality evaluation components but does not require complex processing in unrelated domains that we are not investigating (such as low-level pattern recognition).

4.3.4 Deepening assumption

We assume that a shallow architecture can be deepened without changing the fundamental properties of the architecture (i.e. the necessary properties are preserved). In other words, a deepening would not make the results of the shallow architecture completely meaningless. However, even if this assumption is wrong, the experiments with the shallow architecture may lead to the development of a new kind of deep architecture that may still overcome some of the problems of a hierarchical reflective architecture.

4.3.5 Exploring design space

Figure 4.1(b) only gives one example of a CRN architecture. There are many other architectures that would satisfy the same schema. For example, an agent could monitor all its neighbour’s components (instead of just its meta-level) and none of its own. The main reason for choosing the current version is that an agent can be observed detecting and reacting to an anomaly in its own object-level, thus allowing its neighbour to acquire a model of "normal" anomaly-detection. More details are given in Chapter 5.

The proportion of its own components that an agent monitors and the proportion of its neighbour's that it monitors can be regarded as two dimensions of a *design space*. Another important design decision is the extent of specialisation of agents. For example, figure 4.2 shows the following dimensions:

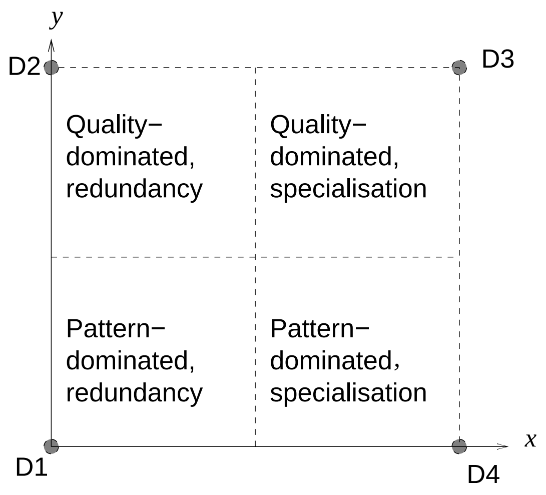


Figure 4.2: A schematic design space

- x: the extent to which the two agents are specialists at different aspects of the primary task. If 0, there is 100 % redundancy, where both agents have identical capabilities but one agent only is in control and carries out the whole task; the other agent is a backup that can override the control of the primary agent if there is an anomaly. If greater than 0 there is a degree of specialisation and one agent gives control to its neighbour whenever its specialist help is required. Each agent has some ability to do all subtasks although not equally well. This means that if one fails, the remaining agent can still carry out the whole task without assistance (although not optimally).
- y: the extent to which a participating agent evaluates "quality" of the external and internal worlds when determining whether or not there is a problem that requires recovery. If 0, there is no quality evaluation (minimal case).

In figure 4.2 the design space is partitioned roughly into regions. The vertices are extremes. For example, at D3, all anomaly-detection is based on quality evaluation and the agents are maximally specialised (no redundancy). These dimensions will each have design subspaces (e.g. if quality is evaluated, to what extent is knowledge of the primary task used?). In the next couple of chapters (in Part 2) we present an implementation of a minimal architecture in which both dimensions are 0.

4.4 Summary and Conclusions

We have now applied the control systems paradigm to the architecture schema in Chapter 3 and we have ensured that a "closed" version of the architecture is possible (Section 4.2.4).

We have explained more precisely why the system is "reflective" in the required way (Section 4.2.5), with "horizontal" reflection (between agents) having the same form as "vertical" reflection (within one agent). In addition, we have explained the relationship between the component map and the model of normal (correct) behaviour. We will see in Chapter 6 how these requirements can be implemented in practice.

The following concepts have also been made more precise:

- Issues about “authority” raised in Chapter 3: notions of “capability”, and “obligation” have also been introduced (Section 4.2.6). We now have a specification of when an agent can override the control of another one.
- The “design space” methodology has been explained and applied to the architecture (Section 4.3).

4.4.1 What remains to be done?

The architecture can now be implemented. The next stage is to define an example primary task, along with specific algorithms and methods to be “inserted” into the various slots of the architecture. For example, how will the object-level solve the primary task? What will the internal sensors be observing? What algorithms are used to detect anomalies? This will be done in Chapter 6 (Part 2 since it will be the first implemented prototype).

Before this, however, we will compare our architecture with related work in distributed architectures in the next chapter.

Chapter 5

Distributed Systems and Fault Tolerance

5.1 Introduction

In this chapter we look at existing distributed systems paradigms. It is necessary to ask whether systems built on these concepts might implicitly have similar properties to a closed reflective network due to the decentralisation and concurrency in their architecture. We also identify useful features in such systems that may be incorporated into a real world CRN architecture.

In the literature a distributed architecture is not usually viewed as a single intelligent system that must survive in a hostile environment and consequently its decentralisation does not serve the purpose of improving reflective coverage. Instead, such an architecture is typically one of the following:

1. a distributed fault-tolerant system (e.g. to provide redundancy and diversity)
2. a society or "team" of agents
3. a biologically inspired distributed system (such as distributed artificial immune system architectures and neural network models)

The categories are not mutually exclusive. We list them here because they correspond roughly to different sub-disciplines, each with its own literature. Work on agent teams, for example, is typically associated with symbolic reasoning and model-building while the distributed fault-tolerance literature is often concerned with timing properties and majority voting.

The chapter is divided into two main sections. The first section introduces fault-tolerance concepts, since they are necessary to understand the distributed fault-tolerance paradigm. The second section compares the various distributed systems paradigms with a CRN architecture, beginning with distributed fault-tolerance.

5.2 Fault-tolerance: main concepts

Most of the concepts in this section are taken from [Avizienis et al., 2001], [Verissimo and Rodrigues, 2001] and [Cristian, 1991] unless otherwise stated.

5.2.1 Faults, errors and failures

[Laprie, 1995] and recently [Avizienis et al., 2001] define *dependability* as "the ability of a computing system to deliver service that can justifiably be trusted". Laprie also defined the following causal chain of "impairments" to dependability:

- **fault**: adjudged or hypothesised cause of an *error*. E.g. design flaw, faulty programming, intrusion, operator mistake, hardware fault.

- **error**: manifestation of a fault on the *system state* that can lead to *failure*, e.g. occasional storage overflow.

- **failure**: manifestation of an error on the service delivered so that actual service *deviates from the requirements*, e.g. an incorrect value is delivered to the user.

A failure occurs when the behaviour of the system deviates from its intended purpose. An error may occur internally without affecting the "service delivery" of the system. When a fault causes an error, the fault is said to be *activated*. When an error causes a failure, the error has been *propagated* to the service interface of the system under consideration and causes the service to be unacceptable.

A failure of one component (e.g. a crash) becomes a fault in the enclosing system, which can in turn produce an error and possibly a failure of that higher system. In our implementations in this thesis there are two levels: (a) the system, which provides a service to the user; (b) its components, which provide different services to the other system components (for example, they deliver values that other components depend on). Thus a fault exists in the system if there is a failure of one or more of its components.

The "fault-error-failure" terminology is not universal in the fault-tolerance literature, but will be useful for comparisons between cognitive systems and fault-tolerance in later chapters.

[Laprie, 1995] also defines a detailed taxonomy of circumstances in which faults arise. The following distinctions are relevant for this thesis:

- physical faults caused by environmental conditions such as radiation, temperature extremes etc;

- accidental design faults (such as software bugs);

- malicious faults such as intrusions and deliberate modification of code;

In particular, malicious faults involve some sort of agent that normally requires *knowledge* about the targeted system. Therefore encryption may be used to restrict this.

5.2.2 Failure models

There are many ways in which a system can fail. Important examples are:

- *Timing failure*: delivery of a service at the wrong time. In particular, a *performance failure* happens when a service is delivered late.

- *Value failure*: delivery of an incorrect value (e.g. a temperature sensor is wrong)

A failure can be *consistent* or *inconsistent*. For example, a value failure is consistent if the same incorrect value is always delivered. By contrast, an inconsistent value failure (or Byzantine failure) happens when the value is sometimes correct and sometimes incorrect, or when different incorrect values are delivered. For example, a component may deliver the correct value to one recipient but a wrong value to another. Note that inconsistency can be malicious or accidental: For example, it is possible to deliberately send a selected user the wrong value, while sending the correct value to all other users.

If there are multiple failures it is also important to consider whether they are *independent* or *related* [Laprie et al., 1995]. For example, the same software fault may be activated in different components.

For our purposes the following distinction is particularly important:

- *Omission* failure: a component does not carry out a required action (for example, a non-delivery of a message). This includes timing failures where the action is not carried out within the required time (performance failures). A component that fails only in this way is often called *fail silent* (e.g. if it drops some messages). If the component permanently stops to deliver any service, the term *fail stop* is commonly used.

- *Assertive* failure: a component performs the wrong function or produces the wrong result (value failure), or it acts when it is required to be passive. This includes non-omission timing failures where the component acts too soon.

In the above, a "component" can also be the highest level system, providing the service to the end-user.

A failure model is a set of assumptions about the kind of failures that can occur. There are three categories:

- *Controlled failure assumptions*: assume components can only fail in certain ways (qualitative bounds) and that there are upper limits on the number of components that can fail within a specified time interval (quantitative bounds). For example, the assumptions may specify that only omission failures occur and that a maximum of two components can fail during a given interval.

- *Arbitrary failure assumptions*: assume that there is no restriction on the types of failure that can occur, although in practice a restriction is usually specified on the number of failures that can occur within a given time interval.

- *Hybrid failure assumptions*: different assumptions apply to different parts of the system. For example, some components can deliver an incorrect value, but other components can only fail by stopping.

We have seen above that some failures are timing failures. If a system is implemented on a physically distributed network, the properties of the network will affect time measurement and the circumstances in which a timing failure occurs. We now discuss this in detail, since it is one of the most important considerations in fault tolerance.

5.2.3 Synchrony models

In the real world, processes may be distributed over a physical network in which there is no global clock. Local clocks can drift and run at different speeds depending on physical circumstances (such as processor type, temperature etc.). Typically a global time service is used to synchronise clocks (e.g. GPS satellite). The computer that receives the time signal has to broadcast it to all others and there will be different message delays depending on distance and other physical properties. Consequently synchronisation will be approximate.

Models of distributed systems are divided into the following categories, depending on the timing assumptions they make:

- Fully asynchronous models: no assumptions are made about timing. In particular, no upper bounds are assumed for processing delays, message delivery time, drift of local clocks or differences between clock readings on different machines.

- Fully synchronous models: upper bounds are known for processing delays, message delivery time, clock drift and differences between clock readings. If these bounds are exceeded then the component is faulty.

- Partially synchronous models: some aspects of the system are assumed to have upper bounds on their timing.

A fully asynchronous model can describe situations where timing limitations either do not exist or they are not known. If timing bounds exist, it may be difficult to determine them by observing the system (e.g. because of its unpredictability and variability). For example, we can imagine a network where correct message delivery always happens within a certain maximum time due to the physical properties of the network. However it may be difficult to estimate this value if actual message delivery time is observed to be variable (even if it is always well within the maximum). Therefore it would be appropriate to make no assumptions about time limits and use an asynchronous model to describe the system. On the other hand, if the network (or part of it) becomes predictable at certain times, it may be possible to describe it using a synchronous model at those times.

5.2.4 Definition of fault tolerance

Having introduced the necessary concepts and models, we are now in a position to define fault tolerance and discuss how it can be attained. [Avizienis et al., 2001] state that "fault tolerance is intended to preserve the delivery of correct services in the presence of active faults".

Fault tolerance is most often achieved using error detection and recovery, along with replication (which will be discussed in the next section). Error detection involves recognising that a potential problem exists (e.g. a parameter value is not in the expected range). Recovery is about stopping the propagation of the error before it causes a failure, and if possible preventing the error from happening again by identifying its cause (the fault). Thus recovery is in two stages:

- *Error handling*: remove errors from the system state; one common way to do this is *rollback*, where the system is returned to a saved state (called a *checkpoint*) that existed before the error was detected. This method is used in our implementations.
- *Fault handling*: prevent a fault from being re-activated. This is a sequence of steps as follows:
    1. *Fault diagnosis*: identify the cause of the error (the fault), which may be a failed component;
    2. *Fault isolation*: exclude the faulty component from service delivery;
    3. *System reconfiguration*: replace the faulty component with a backup or re-allocate tasks among non-failed components.
    4. *System re-initialisation*: update records and tables to agree with the new configuration.

Fault tolerance does not always require error detection or recovery. For example, an algorithm may be fault-tolerant if it is designed to give the *correct result* in the presence of faults, but it does not detect or respond to any error.

5.2.5 Replication as a means to fault-tolerance

Replication of components is often used to provide fault tolerance. Its role is different depending on whether fault masking or error detection is used:

- Replication for fault masking: the redundant architecture of the system prevents any errors from causing service deterioration. There is no error detection or response. Replicas process the same input and their results are combined by a voting mechanism. Thus for n replicas, the following applies:
    - If up to $n - 1$ omission failures occur, at least one component remains to deliver the service.
    - If less then $n/2$ components deliver the wrong value, the majority (correct) value will be propagated by the voting mechanism.

- Replication for error detection and recovery: additional backup components are available that are kept up to date with the changing state of the system. During reconfiguration, a faulty component can be shut down and a backup component selected to take its place. A specialised component (often called a "failure detector") is typically responsible for detecting errors and reconfiguring the system and may itself be replicated.

Replication may be active or passive. For example, three options are supported in the Delta-4 architecture [Powell et al., 1988]:

- *Active replication*: all replicas are equally active and closely synchronised. They all process the same input and produce the same output. Any missing or incorrect outputs can be detected by cross-checking with the remaining outputs.

- *Passive replication*: one replica (called the primary) receives input and delivers the service. All other replicas (the backups) do not participate in service delivery. At specified points in the processing, the primary replica checkpoints its state to all backup replicas.

- *Semi-active replication*: one replica is the primary and delivers the service. The backup replicas process the same input as the primary but they do not deliver output.

If active replication is used, the replicas must be coordinated so that they agree on the state of the input and the order of processing. An example of coordinated active replication is the state machine approach of [Schneider, 1990]. The architecture is composed of clients and servers, where each server is programmed as a state machine that accepts requests from clients. Error detection and recovery is decentralised in this architecture. The equivalent of a failure detector is a collection of components called "configurators", which may be specialist clients or integrated into existing clients or state machines. They can make special requests to servers to reconfigure and re-initialise the system.

One problem that arises with replication is that components must agree on the state of the world. In particular, if the replicas are active or semi-active, they must agree on the content of the input. We will discuss this problem in the next section.

5.2.6 Fault-tolerant consensus

In the fault-tolerance literature (e.g. [Verissimo and Rodrigues, 2001]), the distributed consensus problem involves a number of processes each sending an initial value (called a proposal) to all others. All correct processes must agree on one of the initial values sent by a correct process. This is usually expressed as a set of conditions to be satisfied:

- Termination: all correct processes eventually decide on a value

- Agreement: all correct processes decide on the same value

- Integrity: all correct processes decide on an initial value proposed by a *correct* process (in other words, they must not decide on a value proposed only by a faulty process, since it might cause errors in the consensus application)

Note that consensus involves knowing what everybody else is proposing, and choosing one of those values. It is not the trivial case where processes agree on *any* value simply by executing the same algorithm.

A simple algorithm for distributed consensus is as follows: Every process sends its initial (proposed) value to all others. When all proposals have been received, each process will have a list of values. If every process receives the same list, they can execute a decision function to select one. For example, it might be a majority vote (the value that appears most frequently in the list) or if there is no clear majority, the first item in the list with the highest frequency may be selected. Since all processes execute the same decision function they must decide the same value (assuming the function is not probabilistic). Therefore, once each process has received the same list and the list is complete, they are "destined" to decide the same value and the value is said to be "locked".

Consensus is important for an autonomous system with a distributed control system because each of the participating agents may have a different view of what the state of the world is, or what the "goal" should be. They must be reconciled in some way if the system is to take coherent actions and follow them up consistently. For example, an autonomous vehicle should not start moving towards one goal and then be interrupted repeatedly to move in a different direction towards another destination. A key word here is "repeatedly". A degree of inconsistency or "indecisiveness" may be acceptable for an autonomous system although it would be unacceptable for an information system such as a database service. In fault-tolerance terminology, the system must have the properties of "saftety" (meaning that nothing unacceptably bad happens) and "liveness" (meaning that something good eventually happens). The same conditions apply to interruption of a (presumed) faulty process. A faulty error-detection mechanism that fails occasionally may be acceptable, since it may only cause occasional unnecessary interruptions or a few inconsistent actions.

5.2.7 Byzantine Generals problem

The problem of achieving consensus in the presence of arbitrary faults is addressed by [Lamport et al., 1982]. They imagine a situation where a group of n generals must agree on a common battle plan in the presence of traitors who can send inconsistent false information. They show that it can be expressed as a simpler problem where one general is the commander and the others are subordinates. They call this the *Byzantine generals Problem* and formulate it as follows. A commanding general must send an order to $n - 1$ subordinates such that:

- All loyal subordinates obey the same order.
- If the commander is loyal then every loyal subordinate obeys the order he sends

If there are at least three generals and one traitor, there are two kinds of fault that a single traitor can cause:

1. The commander is a traitor and says "attack" to one subordinate and "retreat" to another.
2. A subordinate is a traitor and lies about what the commander has said (for example if the commander said "retreat", then he tells the other subordinates that the order was "attack").

In the case of processes with no "commander" Byzantine faults can appear as follows:

1. A process sends different values to different recipients in the group. For example it may give correct information to one process and give false information to another, or it may send different kinds of false information to the different recipients.
2. A process lies about a message sent by another process.

Note that these faults are not possible if less than three processes are participating. The solution to the problem depends on whether "oral" or "signed" messages were sent. An oral message satisfies the following assumptions:

A1: Every message sent is delivered correctly.

A2: The receiver of a message knows who sent it.

A3: The absence of a message can be detected.

Note that A3 requires that the system follows the synchronous model, since detecting the absence of a message requires an accurate timeout.

Lamport's group proved the following results for oral messages:

- It is impossible to solve the Byzantine Generals problem with only three processes (one commander and two subordinates) even if there is only one traitor.
- In the general case with m traitors, at least $3m + 1$ generals are required to solve the Byzantine Generals problem.

They presented a recursive algorithm (OM) for oral messages, which can tolerate up to m traitors for $3m + 1$ generals. For example if there are four generals, $m = 1$. The algorithm includes steps in which a process either receives a value within a time limit or assumes there is none and replaces it with a default. Details are as follows:

- $OM(0)$:
  1. Commander sends value to all subordinates;
  2. Subordinates decide on the value received from the commander, or "retreat" if no value received.
- $OM(m), m > 0$:
  1. Commander sends value to all subordinates;
  2. For each subordinate i: call $(OM(m - 1))$ to send received value v_i to $n - 2$ remaining subordinates or send "retreat" if no value received;
  3. For each subordinate i: if it received a value in step 2 from subordinate j (where $j \neq i$) or it detected an absent message then: decide on the final value using $majority(v_1, v_2, .., v_{n-1})$ where an absent value is assumed to be "retreat".

As with distributed consensus, each process accumulates values into a list. The algorithm works because the loyal generals eventually agree on the contents of the list. For example, if there are 4 generals and the commander is a traitor, each general will execute $OM(1)$ (because $m = 1$ in this case). During this step the commander initially sends conflicting values to the three subordinates. However, each loyal subordinate will correctly propagate the received value in step 2. Eventually they all have the same list of values and they execute the majority function. If all three values are different, the majority function selects a default value such as "retreat". The important result is that all loyal generals *agree* on a set of values, even if the values within the set are conflicting.

Similarly if a subordinate is a traitor, all loyal subordinates will receive two correct values (one from the commander and one from the other loyal subordinate) along with one false one (from the traitor). So they also agree on a set of values, except that in this case there is a clear majority. A proof that the algorithm satisfies conditions 1 and 2 of the Byzantine Generals Problem is given in [Lamport et al., 1982].

The used of "signed" messages restricts the ability of a general to lie. For signed messages, an additional assumption is added:

A4: (a) A loyal general's signature cannot be forged and any illegal modification of the signed message content can be detected.
(b) Any general can verify the authenticity of any other general's signature.

Orders received from the commander are propagated as in the OM algorithm, except that in this case messages are signed. The commander signs his orders and every general that receives an order adds his signature to it. At the end, each subordinate has a list of orders, where each order has a sequence of signatures attached, indicating the path traversed by that order. Thus, a subordinate can detect whether the commander has given his signature to two different orders.

The signed messages algorithm requires only $m + 2$ generals for m traitors (if there are less than two loyal generals the agreement problem is trivial).

We have looked at the Byzantine General's problem in some detail because it involves situations where software can be corrupted in complex and possibly malicious ways. For example the meta-level in Figure 4.1 may be modified so that it "lies" about faulty behaviour of another agents and seizes control from it.

5.2.8 Consensus in asynchronous systems: Impossibility Result

One of the most significant discoveries in distributed fault-tolerance is that consensus cannot be guaranteed in a fully asynchronous system if there is one faulty process [Fischer et al., 1985], even if it is only an omission fault. This result is normally called the FLP Impossibility result. It applies even if there are no link failures, meaning that it would also apply if the processes are running concurrently on a single processor.

A synchronous consensus algorithm assumes that a crashed process can be detected correctly. For example, if a process has sent no proposal to any other process, its value can just be "empty". Once every process has received the same list of values (with one empty one), the decision value is "locked" because nothing can prevent the processes from selecting this value (due to the deterministic function).

In a fully asynchronous system, however, a correct process may delay sending a proposal for an arbitrarily long time. It is impossible to tell whether such a process has crashed or whether it is just slow. The FLP impossibility proof showed that it is always possible for such a process to prevent the "locking" of a value, thus preventing consensus.

The Impossibility proof assumes just two values (0,1) as initial values. A central idea is that the current state of correct processes must change from a state in which they might agree on either value ("bivalent") to a state in which they are "destined" to agree on one or the other ("univalent"), even if this decision is not reached yet. Since they can start off in a bivalent state and they will be in a univalent state when they agree, there must be a critical step that changes the system to a univalent state. Fischer's group proved that there is always a possible sequence of events ("schedule") that can avoid this step. Therefore, consensus is not guaranteed.

What this really means in practice is the following: In a fully asynchronous system a timeout value gives no indication of whether something has really failed (because the actual upper bounds are not known). This means that a correct process can be wrongly regarded as failed. For example, there may be a time when all processes have received an identical list of values and they begin to select one from this list. If one process then receives a proposal from a "crashed" process, the new value may cause a different selection to be made by the decision function, thus violating the agreement condition (1) of the consensus problem. Even if a delayed value is discarded or never arrives during the decision, the agreement condition has not been satisfied because there is a correct process that has not participated in the agreement.

Conversely a failed process can be regarded as correct for an arbitrary length of time, preventing the consensus from stabilising (this may for example be a series of retries of the consensus procedure that never terminates). Thus if we design a system so that the agreement condition is guaranteed then the termination condition (2) is not guaranteed. In practice, systems are designed so that either

- imperfect agreement is acceptable (i.e. use a timeout and deliberately exclude any late process from the consensus, even if the process is correct)

- occasional non-termination is acceptable (e.g. for a background problem-solving application that can be stopped if it makes no progress)

5.2.9 Partial synchrony models

The impossibility result depends on the fact that no assumptions can be made about timing. To circumvent this problem, some models of partially synchronous systems have been developed. Approaches to partial synchrony include the following:

1. A consequence of the impossibility result is that perfect failure detection is also impossible a fully asynchronous system. Asynchronous systems augmented with imperfect failure detectors are investigated in [Chandra and Toueg, 1996]. For consensus this model assumes that there are periods during which the system behaves synchronously and these periods last long enough to allow some degree of accurate failure detection. In particular, there is a "minimal synchrony" that allows for the "weakest" kind of failure detection necessary to solve consensus. It has the following properties:

   (a) *Weak completeness*: at least one correct process eventually detects a real failure (weak completeness). In contrast "strong" completeness means that *all* correct processes eventually detect all real failures.

   (b) *Eventual weak accuracy*: eventually there is some correct process that is not regarded as failed by any other correct process. In contrast, "strong" accuracy means that there is *no* correct process that is regarded as failed by any other correct process. The term "eventual" allows for a stabilisation period during which a failure detector can change its decisions about suspected failures. After stabilisation it no longer changes its decision. In contrast, "perpetual weak" accuracy (or simply weak accuracy) means that there is *always* some correct process that is never regarded as failed.

   The above requires that a majority of processes are correct and that communication faults do not cause the network to split up into partitions. The main assumption is that the system eventually "stabilises" into a sufficiently synchronous state required for consensus. Chandra and Toueg also defined a taxonomy of failure detectors with different combinations of strong, weak and eventually weak accuracy and completeness.

2. Timed asynchronous model [Cristian and Fetzer, 1999]: This assumes that there are some properties of the system for which timing bounds can be predicted. In particular, the clock drift is assumed to be bounded (reasons are given for this assumption for typical workstation networks). The maximum message delivery delay is calculated by observing actual message delivery times using the local clock and taking the clock drift into account. The information is used to generate realistic timeouts and to detect performance failures. [Fetzer and Cristian, 1995] showed that consensus can be solved in such a system, provided there are sufficiently long periods of "stability" in which a majority of processes follow the timeliness expectations.

5.2.10 Introducing non-determinism

The above models of partial synchrony assume that there are periods of stability. However, a malicious adversary can manipulate these stability periods or prevent them from occurring. A different approach is to make no assumptions about the synchrony of the system and to consider how such an adversary can be defeated.

The impossibility result proves the existence of a trajectory ("schedule") that the system can follow that will prevent it from reaching consensus. The worst case scenario is an adversary that can force the system to follow this trajectory. For example it may ensure that the consensus never terminates.

To “seize control” of the trajectory it is assumed that the adversary can acquire some knowledge about the system such as message content and internal process states. The question is then asked: “How can such an adversary be defeated so that the conditions for consensus can be satisfied with very high probability?”

One approach is to use random variables to determine the next action of a process. Randomisation for asynchronous consensus is surveyed in [Aspnes, 2003]. The basic idea is that each process has an additional operation called *coin flip*, which returns a result based on a specified probability distribution. Thus, in the worst case scenario there remains some non-determinism in the system, which an adversary cannot control. Even if the adversary can determine every step of a process, it is impossible for it to determine the outcome of a coin flip operation.

Asynchronous consensus generally involves the slower processes changing their proposed values to agree with the fastest processes. A process can use a coin flip to change its proposed value randomly. One of the earliest uses of randomisation was [Ben-Or, 1983]. This algorithm is a series of iterations, each composed of two stages. In the first stage, each process broadcasts its own proposed value (called its “preference”) and then receives values from other processes. If it receives enough messages to indicate a majority vote for a particular value, it broadcasts a “ratify” message for this value. In the second stage, any process that receives a “ratify” message for a value changes its own preference to that value. In the next iteration it proposes its new value. A process makes its final decision on a value when it receives enough “ratify” messages in favour of it (from more than half the processes). Any process that receives no “ratify” message flips a coin to determine its preferred value for the next iteration.

The above introduces randomisation into a process’s choices. An alternative approach is to use “noisy scheduling”, which exploits randomness in the external environment [Aspnes, 2000]. In Aspnes’s “Lean Consensus” algorithm, each process starts with a “preference” for either 0 or 1 (binary consensus). The algorithm allows a “race” between those processes that prefer 0 and those that prefer 1. The value of the fastest process is the decision value. This consensus is enforced by the rule that if a slow process sees any group of faster processes that agree, the slow process changes its value to those of the faster processes (if it initially preferred a different value). The main assumption is that some process will eventually be faster than all others, even in the presence of a malicious adversary. To prevent the consensus, the adversary must ensure that the fastest processes are exactly synchronised. To resist exact synchronisation, the algorithm uses “noisy scheduling” by which the timing of actions is perturbed using independent external variables such as clock drift and memory contention. The main disadvantage is that it relies on randomness in the environment (state of hardware clocks etc).

Recent work [Cachin et al., 2000b] combines randomisation and cryptography to solve the Byzantine agreement problem in an asynchronous setting.

5.3 Comparison of Distributed Paradigms with Distributed Reflection

In this section we consider examples of the various distributed paradigms and compare them in turn with distributed reflection. We begin with distributed fault-tolerance.

5.3.1 Distributed fault-tolerant systems

A distributed fault-tolerant system with error detection and recovery has the basic infrastructure for a cognitive system as defined in Chapter 1. For example, error detection fits into the context of sensing and perception; error handling and fault handling can potentially involve reasoning, decision-making and autonomous action. A “failure detector” in a fault-tolerant system plays a similar role to an agent meta-level in a CRN architecture (Figure 4.1(b) in Chapter 4). A distributed fault-tolerant system can also be “closed” in that all components may be subjected to monitoring and reconfiguration (including the failure detection mechanism).

However, there are some important differences between a “closed” error-detection and recovery network and a closed reflective network:

1. Reflection in general requires an *explicit representation* of the internal processing of the system and the ability to reason about it. For example, it should not just detect errors; it should also reason about problems and explain what is going wrong. This should include “what if” reasoning, which involves the generation of hypothetical states. In other words, CRN assumes that each agent has available to it all existing AI techniques. This is not normally the case in a distributed fault-tolerant network. Such a system may, however, be integrated with reflective systems. An example is the FRIENDS architecture of [Fabre and Perennou, 1998], which applies reflection to fault-tolerance, security and distribution using metaobject protocols. The result is that the low-level mechanisms of fault-tolerance have a transparent representation, which could possibly be integrated with AI techniques such as hypothetical reasoning.

2. Reflection allows access to a system’s internal operations (including software states) using internal sensors. It should not be restricted to observing certain aspects of its behaviour (normally specified as the “input and output” of components). Distributed reflection in particular aims to make available all operations of single agent reflection to a multi-agent system. In other words, all operations that a single agent can apply to its own operations can be applied to another agent, as if it were part of the same agent (although there may be limitations to this in practice, if the system is based on design diversity).

3. A distributed error-detection and recovery network does not emphasise the need for a self/nonself boundary, and in particular it does not generate such a boundary autonomously. However, the requirement to avoid a single point of failure may lead to a design that acts *as if* it were distinguishing between self and nonself, simply by recognising the correct behaviour of its own components and rejecting faulty behaviour (e.g. by removing a faulty replica from a group). It may satisfy the requirements in Section 3.4. However, we cannot say that this boundary between “correct” and “faulty” is autonomously generated because the definition of correctness is usually externally specified. In addition, it has no explicit representation of correctness. Therefore it is also not able to reason about possible threats or *explain* why something is not correct behaviour.

These differences probably relate to the fact that a fault-tolerant system is not normally regarded as a cognitive system, although it may have some behavioural features that could be included in the reactive layer of a cognitive system. It is possible that fault-tolerant algorithms such as consensus may also be incorporated within this reactive layer. The processes of this layer would not be directly accessible to the reasoning of the cognitive system, and may play a similar role to “instinct” in natural systems.

5.3.2 Hofmeyr and Forrest’s distributed artificial immune system architecture

A distributed architecture ARTIS for an artificial immune system is presented by [Hofmeyr and Forrest, 2000]. The architecture is very closely modelled on the natural immune system and is represented as a network of nodes, each of which runs an instance of the negative selection algorithm described in Chapter 2. This means that each node acquires a partial model of “self”. However, it is not an explicit symbolic representation of “self”, which the system can reason about. We will return to this issue in the next chapter and show how our prototypes are different.

This architecture may survive an attack against the immune system using the decentralisation and diversity inherent in the natural immune system. It differs from distributed reflection in the following ways:

1. In ARTIS, there are no explicit requirements on the interaction between nodes, such as mutual monitoring.

2. ARTIS does not use explicit representation and reasoning.

Although ARTIS is not designed according to the fault-tolerance paradigm (for example, there is no concept of a failure assumption), its inherent distribution and diversity give it properties similar to a distributed fault tolerant system with low-level adaptation capabilities.

The focus of investigation in ARTIS is on the mapping of natural immune system components (such as proteins, antigens etc.) onto a suitable representation in a network security environment using low-level bit strings. In contrast, our work is focused on the interactions and control relationships between the architecture components on an abstract level that should not depend on specific representations methods.

Distributing the artificial immune system may overcome the limits listed in Section 2.4 because of redundancy and diversity, both of which are also principles of fault-tolerance. The main difference between ARTIS and a fault-tolerant network is the considerably greater degree of learning and model-building that has taken place in an immune system. However, the model of "self" is not the kind of model that allows reasoning about the status of specific components and modifying these components. Distributed reflection overcomes this limitation.

For our purposes, a distributed AIS may be regarded as a distributed fault-tolerant system in which an autonomously generated self/nonself boundary is achieved to a limited extent.

5.3.3 Agent Teams and Societies

It may appear that distributed reflection does nothing more than a multi-agent society model, in which agents mutually observe each other. In particular, the idea of using multiple agents to compensate for the "blindness" of a single agent is also the basis of "social diagnosis" [Kaminka and Tambe, 1998, Kaminka and Tambe, 2000] which is based on "agent tracking" [Tambe and Rosenbloom, 1996]. Agents observe other agents' actions and infer their beliefs to compensate for deficiencies in their own sensors. For example, 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 deficiency in its own operation by observing the reaction of other agents.

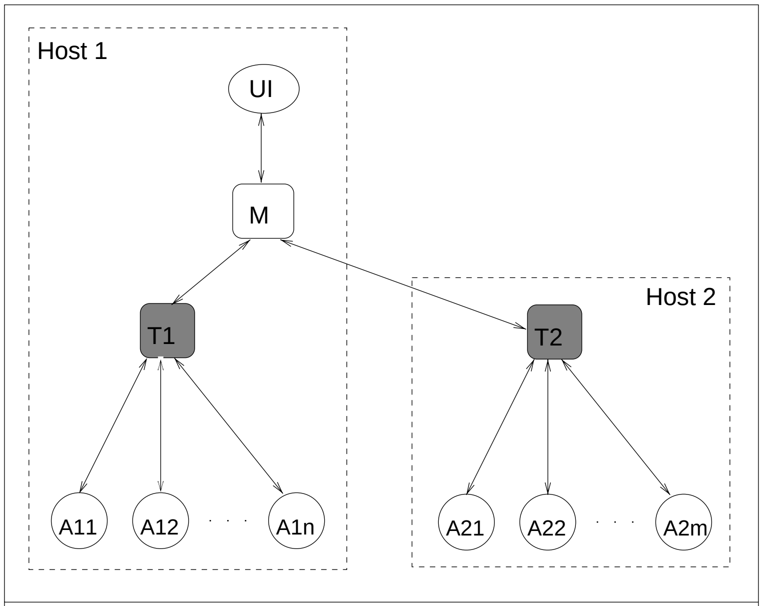
However, Kaminka and Tambe's agents are not intended to be components in a single cognitive system, but a team of separate agents. In contrast, distributed reflection is more like a distributed "nervous system" for a single control system. This means that there are important differences: first, monitoring within a team is restricted to observation of external actions only; in our system the mere observation of the inputs and outputs of anomaly-detection would be insufficient. Access to the internal states of the observed agent is necessary. In the same way, an agent in a distributed control system can modify and influence another agent more directly than in a society. In short, we have made available all the features of single-agent reflection in distributed reflection. Distributed reflection should not allow *arbitrary* use of this modification capability in practice, but this is also true of single agent reflection (it makes no sense for an agent to delete itself for example).

Secondly, a team of independent agents requires that many representations (of self and of the world) simultaneously determine the effectiveness of the whole team. ("Representation" is as defined in Section 4.2.2). In our implementation, only one representation determines the actions of the cognitive system at any one time (that of the primary agent). In contrast to single-agent reflection, however, *several* representations have equal "authority", since the backup agent's representation can be used as a basis for interrupting the primary agent when it is believed to be faulty. This has the advantage that the "ruling" representation can be "questioned" in certain circumstances. Nevertheless, during *normal* operation, the system has all the advantages and simplicity of a single cognitive system with a hierarchical structure.

Architectures combining features of the team model and distributed reflection may be feasible. For example, the agents may have different tasks or require diverse representations. The more "team-like" the system becomes the greater the difficulty of internal monitoring and modification of other agents (since they may not understand each other's internal representations). A "team" in which every agent reasons about the external behaviour of all other agents would be a limiting special case of distributed reflection, provided that the agents can influence each other directly - without the need for communication or negotiation.

5.3.4 Agent-based intrusion detection systems

Multi-agent architectures can be used for distributed intrusion detection. An example is AAFID [Balasubramaniyan et al., 1998] which is based on the concept of multi-agent defence introduced by [Crosbie and Spafford, 1995]. An example of a simple AAFID configuration is shown in figure 5.1. Each agent observes some aspect of network traffic and acquires a model of its normal activity during a training phase so that anomalies can be detected. However, there is no requirement for agents to observe each other. Similarly, there is no explicit concept of "self" as with artificial immune systems. Because the system is distributed, an attack on the intrusion detection system itself (subversion) may



A11, A12, ...: specialist anomaly−detection agents
T1, T2: transceivers manage agents on one host
M: monitor manages transceivers on multiple hosts
UI: user interface on host 1

Figure 5.1: An example AAFID system

be tolerated by using a kind of redundancy where several agents observe the same activity and cross-check their results.

However, AAFID has a hierarchical structure and has the same disadvantages of layered intrusion-detection mentioned in Chapter 1, Section 1.6. Agents do not mutually acquire models of each other's behaviour. On the other hand, a strictly hierarchical architecture has the advantages of being easy to maintain and to visualise.

Other distributed intrusion-detection systems with a hierarchical structure include the following:

- MAIDS: “Mobile Agents Intrusion Detection System” [Helmer et al., 1998]. This is similar to AAFID, except that mobile agents are used on the lower levels of the hierarchy.
- JAM: “Java Agents for Meta-learning” over distributed databases [Stolfo et al., 1997]: meta-learning agents combine results of specialist data-mining agents on different hosts to produce a high-level model of normal financial transactions. Meta-learning agents are controlled by management software and a user interface.

Non-hierarchical versions of AAFID or other agent-based intrusion detection systems are not known at the time of writing. A non-hierarchical version of AAFID may involve agents mutually observing each other. In its simplest form this would still have the limitations of an agent society. In Chapter 12 we will consider how an AAFID type system could be transformed into a distributed reflective system.

5.3.5 Intrusion-tolerant Intrusion Detection Systems

The “Intrusion Tolerance” paradigm of the MAFTIA project ([Dacier, 2002] and Chapter 4 of [Powell and Stroud, 2003]) presents a distributed intrusion detection architecture that has much in common with an agent-based architecture. The main difference is that it aims to counteract attacks against the intrusion detection itself (“intrusion tolerant intrusion detection system”). An example MAFTIA architecture is shown in Figure 5.2. The architecture makes a distinction between *sensors* and *event analysers*. A sensor records raw events from observed activity (e.g. packets on the network or file access activity on a host). An event analyser determines whether the events constitute an intrusion and generates an alarm if it believes this. Sensors and analysers may be combined in different ways. Figure 5.2. A second layer of analysers can be used to determine which alarms may be false-positives (in the diagram, analysers 4, 5 and 6 may have this role). For example, they may use knowledge about classes of false alarms acquired using data mining (this is another area of the MAFTIA project).

Unlike most distributed intrusion detection systems, the MAFTIA architecture has a clearly defined “meta-level” containing “meta-IDS” analysers (for “meta-intrusion detection system”). A meta-IDS observes another analyser and determines whether it is functioning correctly or whether it have been compromised. In Figure 5.2, any number of the analysers 4, 5, 6 or 7 may be meta-IDS’s. The MAFTIA “meta-level” has two main building-blocks:

1. *Knowledge-based meta-IDS*: uses *knowledge* about which alarms *should* be raised by the observed analyser in which situations.
2. *Behaviour-based meta-IDS*: analyse behaviour of observed IDS. It may build a profile of normal behaviour for purposes of anomaly-detection.

If we refer back to Figure 4.1(b) showing the distributed reflective system, MAFTIA’s behaviour-based meta-IDS is very similar to the meta-level of an agent in its role as observer for *another agent’s meta-level*. In the CRN architecture, an agent’s meta-level is comparable to an intrusion detection system (IDS). However, the levels (IDS and meta-IDS or alternatively “meta-level” and “meta-meta-level”) are not separate components but are instead different *roles* for the same component. In fact we will see later that having the same component for both roles is advantageous for the purposes of compact model-building.

Countermeasures against attacks

The MAFTIA report of [Dacier, 2002] enumerates the following methods of attacking an intrusion detection system:

- Tamper with communication channels between sensors and analysers;

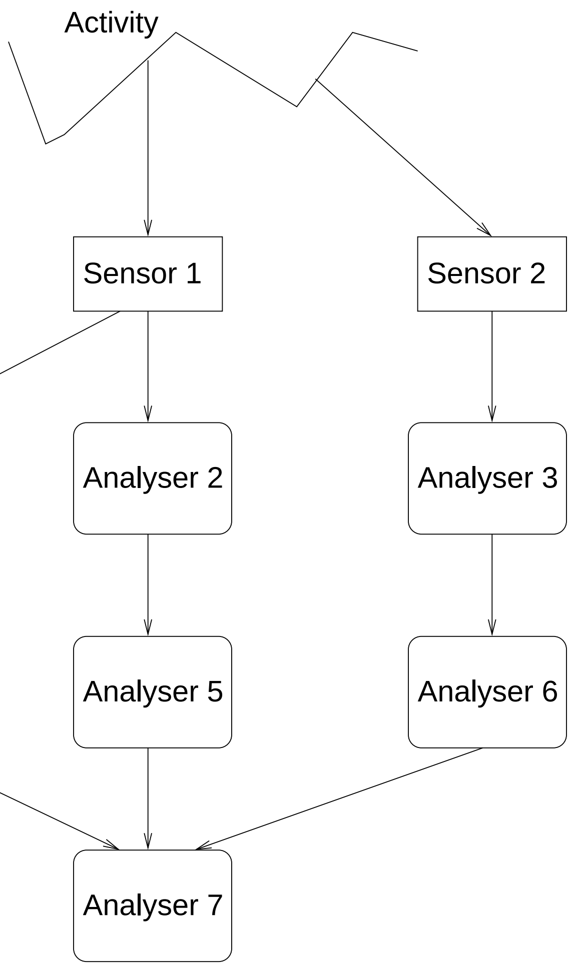


Figure 5.2: An example MAFTIA intrusion tolerance architecture

- Launch a Denial of service (DoS) attack against an event analyser, or just crash it;
- Assume control over an event analyser, but do not crash it.

Disruption of a channel can be detected using a heartbeat to ensure the channel is alive. More complex forms of compromise (such as eavesdropping or hi-jacking the channel) can be counteracted using authentication and encryption. We will return to this issue later in Chapter 11.

MAFTIA proposes the following countermeasures for different kinds of compromise of an event analyser:

- Crash or DoS attack:
  - “Immortalizer” process: watch over each software component of the intrusion detection system. Use a heartbeat to show that they are alive.
  - Continuous online testing: periodically launch mock “attacks” to ensure IDS is working correctly
- Generate large numbers of false positives (“noise” attack):
  - meta-IDS compares targeted analyser’s output with other analysers.

- Modify the analyser so that it no longer reports problems ("Covering of tracks"):
  - replicate the analyser: compare their inputs and outputs; detect discrepancies.

We will compare these features with the capabilities of our prototypes in subsequent chapters.

To summarise, the main differences between the MAFTIA architecture and a CRN are as follows:

1. The meta-level layers are hierarchical in all currently known examples of the architecture. There is no mutual monitoring between meta-IDS's. However, a single point of failure may be avoided using standard fault-tolerance techniques. For example, the MAFTIA general architecture [Powell and Stroud, 2001] may allow the top-level meta-analyser to be replicated as a group of fault-tolerant processes. In this case the differences between distributed fault-tolerance and distributed reflection apply (Section 5.3.1).
2. The concept of the same component carrying out different meta-level roles is not present in the MAFTIA architecture.
3. It seems to have the limits of an agent "society", in that only inputs and outputs of components are observable. However, this may be a practical limitation in real-world systems, due to lack of availability of internal code sensors and program profiling.

The MAFTIA intrusion tolerance architecture is not a cognitive system paradigm. It may, however, be a starting point for the integration of AI techniques with fault-tolerance concepts in the real world. We will discuss this further in Chapter 11.

5.4 Summary and conclusions

In Table 5.1 we summarise the degree to which the distributed paradigms satisfy our requirements for a cognitive system in Chapter 1, Section 1.2.6.

	Self/nonself distinction	Reasoning	Reflection	Adaptation
Fault tolerance	potentially	No	No	possible
Immune systems	potentially	No	No	Yes
Agent teams	weak	Yes	No	Yes
MAFTIA system	potentially	possible	possible	possible
CRN	Yes	Yes	Yes	Yes

Table 5.1: Requirements met by various distributed systems paradigms

"Agent teams" include AAFID and other agent-based systems mentioned in Section 5.3.4. "Potentially" means that it may be possible to incorporate the new requirement into the existing paradigm. For example, a distributed fault-tolerant network using error-detection and reconfiguration may approximate the requirements for self/nonself distinction in Section 3.4, although the "component map" would not be a high level representation that it can reason about. The MAFTIA intrusion tolerance system may support non-hierarchical configurations of analysers, which combined with the fault-tolerant "middleware" might also satisfy this requirement.

In an agent team, a weak form of collective self/nonself distinction is possible. "Nonself" detection may be the detection of behaviour that breaks the team "rules" or norms. "Self" protection may involve excluding a team member who repeatedly breaks the rules. Reasoning about other agent's external behaviour is not normally called “reflection” because the agents are treated as separate “individuals” (Section 5.3.3). In a closed reflective network, by contrast, an “agent” is merely a useful design construct. The distinctions between agents are not very significant from the point of view of the cognitive system. An “agent” is merely a replaceable component (the highest level in the component hierarchy). We will return to this issue in later chapters.

Part II

A PROOF-OF-CONCEPT IMPLEMENTATION

Chapter 6

A Minimal Prototype

6.1 Overview

In this chapter we describe a working implementation of the closed reflective architecture introduced in Chapter 4. Due to the nature of rapid-prototyping, initial implementation often reveals areas of the architecture (or even the architecture schema) that need to be changed or made more precise. Hence, in the chapter structure, some sections relating to implementation may be followed by sections revisiting the architecture.

This reflects approximately the discovery sequence that actually took place during development. It also shows that there is often a kind of inductive reasoning where a specific implementation reveals something about requirements for a whole class of implementations (architecture-related).

The "evolutionary" process of iterative design and implementation produces changes and refinements on different levels as follows:

- Architecture schema: add new constraints or remove or "soften" some others.
- Architecture: add or remove components, specify new mechanisms for interaction between components, etc.

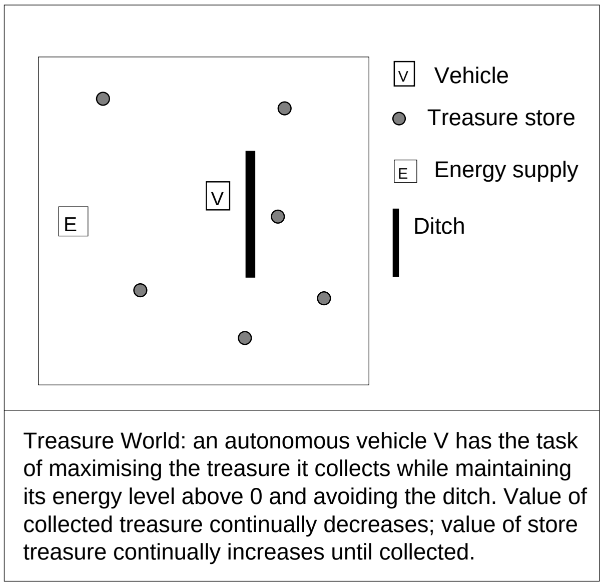
6.2 Primary Task

Returning to the control system architecture in Figure 4.1, we now define an external world and the variables within it that should be maintained within acceptable values. The "variables" do not have to be numeric; some may be in the form of logical expressions which must be satisfied.

6.2.1 Treasure: a simple control scenario

We designed a simulated world called "Treasure" which is a variant of an existing scenario called "Minder" [Wright and Sloman, 1997]. A configuration of the Treasure scenario is shown in Figure 6.1. The virtual world is made up of several treasure stores, one or more ditches and an energy supply. An autonomous vehicle V must collect treasure, while avoiding any ditches and ensuring that its energy supply is regularly restored. The "value" of the treasure collected should be maximised and must be above 0. Collected treasure continually loses value as it gets less "interesting". Treasure stores that have not been visited recently are more interesting (and thus will add more value) than those just visited. The system "dies" if any of the critical requirements are violated.

The main purpose of this simple scenario is to allow quick visualisation of the changing state of the world, and hence its *quality*. E.g. it is easy to see if the vehicle has fallen into the ditch or has run out of energy, and this would show immediately that a particular design is unsuccessful.



Treasure World: an autonomous vehicle V has the task of maximising the treasure it collects while maintaining its energy level above 0 and avoiding the ditch. Value of collected treasure continually decreases; value of store treasure continually increases until collected.

Figure 6.1: Treasure Scenario

Parameters specifying growth and decay rates of treasure and energy were selected so that the architecture is sufficiently “challenged”. Details are in Appendix A, Section A.1. The selected values ensure that a system without anomaly detection really does fail when attacked. Some output traces showing this are in A.3. Conversely the values should not be so “harsh” that the system fails when it is not attacked or when it responds to an attack correctly (for example if the decay rates are too high). An output trace of normal operation is in A.2.

In the Treasure scenario, distributed reflection means that control of the vehicle is shared between two agents A and B. In other words, multiple agents run on the same vehicle. Figure 4.1(b) shows the situation where A is in control. More complex configurations are possible, in which agents are specialists.

6.2.2 Why choose this scenario?

The maintenance of safety, energy and treasure values is the primary task whose satisfactory performance needs to be protected in a hostile environment. We have chosen this scenario partly for practical reasons (because it is easily visualised) and partly because it involves two different levels at which intrusions and responses can take place, namely the vehicle control and the value of the “treasure” produced. It can be compared with a scaled-down version of a science scenario which contains the following components:

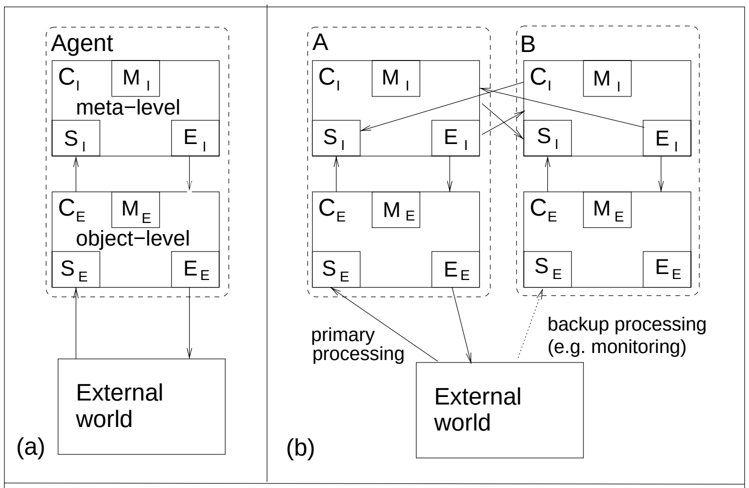
1. A remote autonomous vehicle which collects data (treasure) and whose functionality the system has to preserve without external intervention (energy supply, safety); an intrusion can disable or take unauthorised control of the vehicle;

2. Scientists who request information services from the vehicle and where the information can vary in “value”; an intrusion can withhold the data, modify it or allow unauthorised access;

Real world examples include astronomical observations or environmental monitoring using autonomous ocean sampling vehicles such as in [Turner et al., 1996]. It is important to keep such examples in mind to show up areas where our current architecture and simulated environment are extremely shallow and to identify the kinds of scaling up that would be required. For example, in the Treasure scenario, the fault- and intrusion-tolerance capability of the software (which is real) does not depend on the energy-level of the vehicle (which is simulated). In the real world, it probably *would* depend on the energy-level of the vehicle, however, since it might have to conserve computational resources.

6.2.3 Design space revisited

If we return to the design space in Chapter 4, Section 4.3.5, the two dimensions are 0 for the minimal prototype (no quality evaluation, no specialisation respectively). This means that the primary task is carried out asymmetrically by one agent only (the primary) while the other is a passively monitoring backup. Figure 4.1(b) in Chapter 4 shows only one agent interacting with the external world. For convenience, Figure 4.1 is reproduced in Figure 6.2.



C_I	control of internal world	C_E	control of external world
M_I	model of internal world	M_E	model of external world
S_I	internal sensors	S_E	external sensors
E_I	internal effectors	E_E	external effectors

(a) Single−agent reflection; (b) Distributed reflection (simplest case): agent is divided into primary and backup copies A and B. Only the primary changes the external world; both have read/write access to each other's meta−levels and their own object−levels

Figure 6.2: Reflective agent architectures (reproduced)

An alternative to the primary/backup design would be to make one agent a specialist in treasure maximisation while the other is a specialist in safety and energy.

We chose the primary/backup configuration because the *trajectory* from centralised to distributed is simplest (only a change in the meta-level configuration) and a simple *duplication* of the object-level. Moreover, sharing a task is more complex, since it often involves interaction and communication between agents. The additional programming effort is not justified. As in many software engineering situations, some design decisions are made to enable ease of implementation and not because they are fundamentally the "best" or only design.

6.3 Implementation of external control layer

We now outline the implementation of the architecture's object-level (or application level) which does the primary task. This is the external control system in the Treasure scenario.

The implementation is based on the SIM_AGENT package [Sloman and Poli, 1995]. In SIM_AGENT, an agent execution is a sequence of sense-decide-act cycles which may be called *agent-cycles*. Each agent is run concurrently with other agents by a scheduler, which allocates a time-slice to each agent. For simplicity we assume that agents are run at the same "speed" and that exactly one agent-cycle is executed in one scheduler time-slice (although there may 0 or more agent-cycles in a given time-slice). We use the term "cycle" to mean an "agent-cycle" unless otherwise specified.

The architecture can be described using a synchronous model (Chapter 5, Section 5.2.3. There is no physical distribution and the agents run at the same "speed". Local clock drift rate is 0 (so effectively there is a global clock) and the message delivery time is constant. "Messages" between agents are not messages across communication channels. They are internal sensor readings and they have the same form *between* agents as they have *within* an agent.

Each agent architecture is encoded in a set of pre-defined rules (called a rulesystem), which is divided up into modules (rulesets). Rules are generally either reactive or deductive. Selected modules of a single agent are shown schematically in Table 6.1 (for simplicity, many components have been omitted, e.g. interpretation and evaluation of sensory data). Note that the meta-level is also included as a set of rules (to be defined later). The table also shows a component hierarchy of rules, rulesets, ruleset groupings and whole agents. The lowest level components are the rules; the highest level components within an agent are the ruleset groupings "Sense", "Decide", "Act" and "Meta-level".

For example, the "Sense" component contains one ruleset "external_sensors", in which a rule fires

Function	Ruleset	Rule
Sense	external_sensors	see_treasure?
		see_ditch?
		see_energy?
Meta-level	internal_sensors	...
	acquire_model	...
	use_model	anomaly?
		repair_required?
Decide	generate_motive	low_energy?
		low_treasure?
	select_target	new_target?
		target_exists?
Act	avoid_obstacles	near_an_obstacle?
		adjust_trajectory?
	avoid_ditch	near_ditch?
		adjust_trajectory?
	move	move_required?
		no_move_required?

Table 6.1: Selected architecture modules

if a certain object is recognised. The system continues to work correctly if objects are moved around (although adding or removing objects requires re-running the simulation with different parameters).

For space reasons, rules in Table 6.1 are in the form of questions. For example, the rule "see_ditch?" is effectively asking the question: "is there an object whose dimensions and features match those of a ditch?" and taking the appropriate action if the conditions are true. Similarly, the rule “adjust_trajectory” asks “should the trajectory be adjusted?”. For example, the vehicle may be moving very close to the ditch and it may have to make some corrections to avoid it.

An agent’s rules are run by a rule-interpreter, which runs each ruleset in the order specified in the agent architecture definition. During an agent-cycle, a particular ruleset may be run zero or more times by the interpreter. In our implementation, each ruleset is run exactly once.

The correct functioning of the architecture depends on mechanisms to implement the architecture correctly such as the rule interpreter. As mentioned earlier, we have to ignore such implementation meta-levels since they do not belong to the simulation being evaluated.

6.4 Designing a Hostile Environment

Normally an autonomous system (whether multi-agent or not) has no difficulty surviving in the Treasure environment as defined above. We imagine there is an “enemy” that can interfere destructively in any of the following ways:

1. *Damage* to the system’s control software in the form of omission failures only. A damaged component is “fail-stop” [Laprie, 1995], meaning that it permanently ceases any activity (until it is repaired).

2. *Modification* of a system’s control software by inserting hostile code that can behave destructively or cause deception (value failure).

Note that a “component” can mean a single rule, a ruleset, a collection of rulesets or a whole agent. For example, disabling an agent would cause it to crash. Disabling a single rule would cause the rule to be ignored by the rule interpreter).

Deception is mostly covered by (2) above. The simplest example is direct modification of sensor operation so that they give false readings. It is possible that all these types of interference can be detected as anomalies in external or internal sensors.

Note that the design of the environment is the exploration of “niche space” in the design-based methodology (Section 4.3). A “niche” is similar to a set of failure assumptions in fault-tolerance.

6.4.1 Failure assumptions

In the prototypes in this thesis we are assuming that failures are *persistent* and *independent*. In the case of an omission failure, a damaged component becomes permanently inactive (unless it is restarted or replaced). In the case of a value failure, a component delivers the same incorrect value persistently. It does not change intermittently. This means that we are excluding *Byzantine* failures [Lamport et al., 1982]. In our scenario, an example of a Byzantine failure would be an agent executing hostile code sending contradictory messages to different agents (for example, it might tell agent A that a component is faulty but to agent B it might say that the same component is operational). We are assuming that any hostile code that “lies” will tell the same lie to all agents.

6.4.2 Simulating an enemy

Destructive behaviour and “lying” (even “Byzantine” failure) may be caused by damage or accidental faults. In the implementation, however, we introduce an additional “adversary” that attacks the software in various ways. This is similar to existing “fault-injection” techniques such as [Hsueh et al., 1997].

6.4.3 A random failure environment

Initially, we choose an environment where randomly selected components are disabled at random intervals, and where the minimum interval between attacks is long enough to allow recovery if a problem is detected. The agent that simulates the "enemy" selects a target agent at random (either A or B) and deletes a randomly selected component (in this case a rule). The attack is time-triggered.

We have chosen this environment initially for practical reasons, since it is easier to disable correct behaviour than to "design" hostile behaviour. One advantage of this restriction is that it may be possible to *guarantee* that an absence of normal activity of an essential component can be detected, given that it is possible to design the system so that the critical components show a minimum amount of activity on every cycle. Hence the architecture being tested can be more precisely understood and explored. To avoid making the problem too trivial, however, we are also assuming that the precise nature of the correct activity *as it shows on an observing agent's internal sensors*, is not known in advance and must be learned by observation.

6.4.4 Events in an internal world

Events in an internal world must be observable, that is there must be some mechanism for leaving a trace that can be picked up by internal sensors. It was necessary to extend the SIM_AGENT toolkit to enable this. The mechanism is as follows:

During each agent-cycle, a trace entry is produced for each rule that has its conditions checked, along with an additional entry if it fires. We can call these two types of trace entries *event types*. An *event* is a particular occurrence (instance) of an event type, and an *event record* is a record of the event as it appears in the trace. During one cycle t, two types of event are recorded: beginning of condition-checking of a rule and beginning of rule firing (this can be refined by identifying the checking of individual conditions or the performing of individual actions). In the next cycle $t+1$, the execution trace containing the list of event records is loaded into the agent's database by its internal sensors (data access procedures). Thus the agent can compare the sensed trace with the expected trace predicted by the model. In accordance with Figure 6.2(b), an agent's meta-level evaluates traces from two different sources, namely its own object-level and its neighbour's meta-level. The trace of its neighbour's meta-level is the one produced by the neighbour's last execution cycle. The cycle-number (timestamp) is recorded along with the trace. In this way, agents can observe (and possibly repair) each other's self-observation and self-repair processes.

Note that if speed differences were allowed, the observed agent's last cycle-number would differ from that of the observing agent, but since we are excluding this possibility, we can talk unambiguously about the "last observed cycle" of both the observing agent itself and its neighbour.

Note also that we are talking about the *real* operation of the agent's software, so that the internal world is *not* a simulation (although the external world is).

6.4.5 Terminology

The term "execution pattern" is generic and can be used on three different levels:

1. a single event record is a pattern (of atomic symbols),
2. a trace for a single agent-cycle is a pattern (of events),
3. the sequence of traces over many agent cycles is also a pattern (of traces).

Initially we are mostly concerned with (1) and (2). Similarly the term "software component" is used on different levels:

1. a whole ruleset - identified by the ruleset name.

2. a rule within a ruleset - identified by a (ruleset, rule) pair

A condition or action of a rule may also be regarded as a component but our current implementation does not treat them as such.

An event record is also a declarative statement which uniquely identifies a component, i.e. a (ruleset, rule) pair, along with what happened to it. An example of an event record is: "Rule R of ruleset S of observed agent A had its conditions checked at cycle T". Consequently a whole trace for one agent-cycle is a conjunction of statements that are true for that cycle (usually the last observed cycle). The resulting database may be used for high-level reasoning and diagnosis as well as for low-level statistical pattern-matching used in existing artificial immune systems such as in [Forrest et al., 1994]. EMERALD [Porras and Neumann, 1997] is an example which combines both these levels.

6.4.6 Environmental restrictions

Environmental restrictions say what can and cannot happen in an environment in which an architecture is being tested. The restrictions are necessary in order to give the architecture a *possibility* of surviving. Choosing these restrictions means finding a middle-ground between making it impossible to survive and making it trivial. The type and number of attacks, and their timing should be such that the novel features of distributed reflection can be tested.

There are some environments in which it is always impossible for a system to survive, even if its software is operating correctly. In our investigation, we exclude "impossible" situations by imposing the following restrictions:

U1: *Effects of an attack can be sensed:* The effects of an attack causes changes in the execution trace of an agent; that is, the attack should cause the agent to behave differently in a way that is recorded in its event trace. The attack itself is invisible (the enemy agent's activity leaves no trace). We choose the more difficult situation where only its *effects* can be sensed, and these may be delayed or indirect.

U2: *Minimum time interval between attacks:* The time interval between attacks is above the minimum time required to allow detection and recovery from the problem when the software behaves correctly; in other words, it should be physically possible for it to recover. This would exclude a situation where only one agent is attacked but several of its components are attacked sequentially while it is still recovering from a previous attack.

The "U" means "universal", because the restrictions apply to the testing of all our architectures. To test any two-agent reflective architecture, we apply the following restriction:

Arc1: For a two-agent system, a single attack only affects *one agent*. Two agents are not attacked simultaneously. However, the number of components within one agent that are simultaneously attacked is not limited. "Simultaneously" may mean that each component is attacked in rapid succession, provided that anomalous event-records in the trace can easily be grouped together in time and regarded as the effects of a single attack. For an n-agent system (where $n > 2$), $n - 1$ agents can be attacked simultaneously.

If this restriction is not upheld, the system may fail because the anomaly-detecting components of both agents can be simultaneously disabled, making it impossible to explore the capabilities of mutual reflection between agents.

The restriction is labelled **Arc1** for "architecture-related", meaning it can be varied if the architecture is varied (in this case by increasing the number of agents). We consider the more general form of the restriction in Chapter 9.

6.5 Implementation of Internal Control Layer

We can now describe the implementation of the *internal* control layer or meta-level in Figure 6.2(b).

6.5.1 Artificial Immune System algorithms

In accordance with the "broad-and-shallow" approach, a scaled-down version of Forrest's artificial immune system (introduced in 2.4) was used to carry out the function of anomaly-detection required by C_I. Hence, an agent's model of "self" (which we call M_I) is acquired during a training phase. The self-model is a signature of "normal" rule execution events.

An important difference between our approach and Forrest's approach is that we are aiming for a model containing symbolic statements. Each statement says something about which components are normally active in which situations. In contrast, the model of "self" acquired by typical artificial immune system (AIS) algorithms are sets of subsymbolic strings that are matched with real execution patterns using numerical and syntactic string comparison methods. The "picture" of the whole system behaviour has a more fine-grained resolution. However, an abnormal string or a significant mismatch between sets of strings does not necessarily identify a component that may be causing a problem. In short, an AIS does not reason about its components, but our system does because we require this capability for reflection (as explained in Sections 5.3.1 and 5.3.2).

We should mention that the meta-level C_I need not be an artificial immune system of the above type. There are other intrusion detection systems that are composed of a model-acquisition algorithm and an anomaly-detection system (for use in training and operational modes respectively). Recent work includes the following:

- Real-time intrusion detection based on a model produced by data mining algorithms [Lee et al., 2001b].
- Intrusion detection using program behaviour profiles learned by a back-propagation neural network algorithm [Ghosh et al., 1999b]
- Learning predictive rules from a feature vector representation of system call traces [Helmer et al., 2000]. Genetic algorithms are used to select a subset of features to represent the data.

We have focused on immune system algorithms because their stated objective is to build a model of "self" and to make a distinction between self and nonself.

In Chapter 2, Section 2.4 we outlined two artificial immune system algorithms, namely a normal activity "signature" [Forrest et al., 1994] and negative selection [Dasgupta and Forrest, 1996]. Negative selection does not detect *absences* of patterns associated with the normal functioning of components, which would be useful for our random failure environment. We therefore implemented a simple version of Forrest's signature-based immune algorithm to deal specifically with absences caused by unauthorised disabling of components (i.e. rules). In particular, we addressed the following question:

> Is it possible to *guarantee* that an omission of essential parts of the signature can be detected, given that it is possible to design the system so that the critical components leave a trace on every cycle?

The assumption of ensuring that critical components are active on every cycle is reasonable because similar techniques are used in fault-tolerant software systems, e.g. message logging and checkpointing [Pradhan and Banerjee, 1996]). For our purposes, we define the critical components to be the meta-level rules implementing the immune system itself (shown schematically in Table 6.1). Although a critical component can be designed to be active on each cycle, we do not know how this activity will register on an observing agent's sensors because of our requirement that the precise sequences of events and their frequencies are not known in advance. Nor do we know which events (in the whole event space) actually occur in the execution. It is possible that some software components (rules, or even whole rulesets) are never active.

This particular combination of restrictions is necessary in order to find a useful middle-point between a problem that is too difficult initially and one that is too trivial. In spite of this, this problem may not be completely unrealistic, since both the observing and observed agent may be based on different software and developed independently. The only requirement is that the sensors are "suitable" for the particular internal environment (observed agent's patterns) in that they can be used to build a sufficiently accurate model of it.

6.5.2 Training and operational phases

The simplified artificial immune system that we use is based on two algorithms: one for use in the training phase, and another for the operational phase.

During the training phase, a "positive detector" is generated for every new event record encountered and given a weight of 1. For every subsequent occurrence of the same event during the training phase, the weight of its detector is incremented. At the end of the training phase, the total weight can be expressed as a percentage, where 100% means that the event occurred on every training phase cycle and 0% means that it never occurred (for example, if a rule's conditions were never checked or its actions were never run).

During the operational phase, if a positive detector fails to match something in the trace and the detector's weight is 100 %, this is regarded as an "absence" anomaly. We only need this simple version of the algorithm to satisfy our restricted requirement stated above. (Since the critical components must indicate that they are alive on each cycle).

To detect more subtle anomalies (such as modification or resource diversion), it is possible to detect changes in the overall trace, while tolerating noise. There will be some mismatches for detectors whose weights are less than the total number of cycles (as activity in each cycle varies). The simplest method is to define a threshold above which the weighted sum of mismatches would be considered as an anomaly, as in [Forrest et al., 1994].

At the end of the training phase, a "model" of the observed system has been acquired, based on three sets of events:

P: What events occur on every cycle? (positive detectors with weight = 100)

T: What events sometimes occur? (tolerance: $0 <$ weight < 100)

N: What events never occur? (negative: weight = 0, those events not in P or T).

P and T together form the "signature". For our random failure environment, however, we only need to think about the set P. (In more noisy environments, the requirements may be modified so that P includes weights *close to* 100% or that N includes events in T with extremely low weight).

Note that a very simplified form of negative selection would use the set N instead of P, since an anomaly would be any activity which was never detected in the training phase.

6.6 Bootstrapping of a Self-Model

We now return to an important feature of the architecture, namely the mechanism by which it bootstraps a self-model. As happens frequently in rapid prototyping, the details for this architectural feature were only precisely understood after the first stages of implementation. Therefore we mention it here rather than earlier.

6.6.1 Design of a training phase

In accordance with the meta-level configuration in Figure 6.2(b), our first building block is a single agent system which can build a model of its its own object-level and use this model to survive. The end result is a working meta-level (immune system in this case) which can be observed in operation.

In the next stage, both agents must acquire models of each other's meta-levels. However, the following problem arises: while their immune systems are both in training phase, they cannot observe the *operational phases* of each other's immune systems. Hence, as soon as they enter operational phase, they will find each other's immune patterns to be "foreign" (failure to tolerate "self" in immune system terminology). In the current implementation, this is a failure to tolerate absences of the training phase events, which is the same problem in principle. We solve the problem by allowing agents to "take turns" observing each other in different states. To explain our solution in detail, we introduce some terminological conventions.

6.6.2 Terminology

We have been using the term "phase" (training phase and operational phase) to refer to stages in the agent's timeline. Phases follow each other in sequence and are not repeated. As a more general concept, we use the term *mode* to refer to any state which lasts for a period of time, independent of history. For example, an agent can be in the mode of acquiring a model (training) or in the mode of using the model to detect anomalies (operational). Hence, if the agent returns to training mode after an operational phase, it is in a different phase (training phase 2 not phase 1) but in the same mode as it was in earlier. This distinction is important because the model being built does not need any historical information on phases, but only the modes that the agent can be in. In general, we can think of a phase as a historical instance of a mode.

To show how an agent acquires a model of another one, the following naming convention is useful: the agent whose observation process we are currently describing is labelled r, while the agent it is observing is labelled d (for observe*r* and observe*d* respectively). Although both agents simultaneously observe each other, we must describe how one agent does the observation. When we are describing the process of building a model then r is in training mode, but d may be in either mode.

6.6.3 Temporal diversity

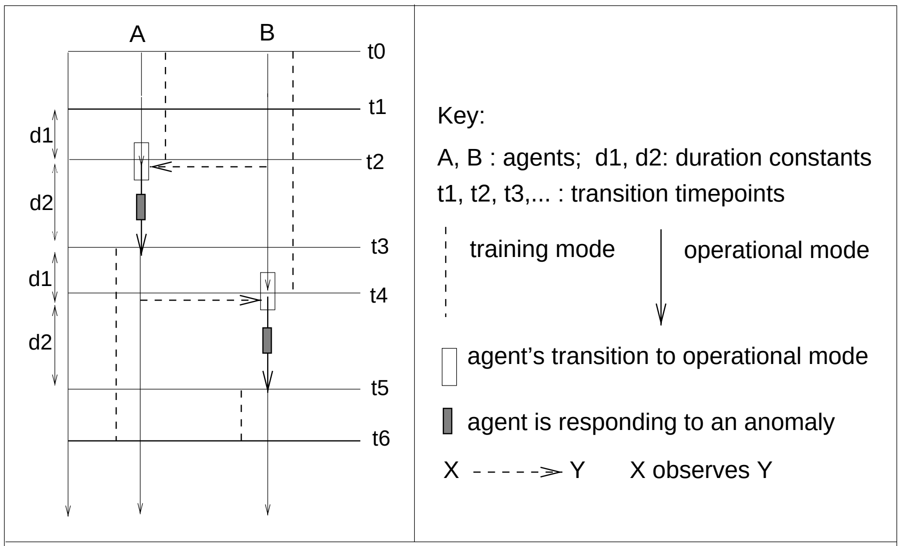
If r is to build a model of d's meta-level it must observe d making a transition between training and operational modes when it (r) is in training mode. Since this cannot happen if both agent's training modes are synchronised, there is a requirement for "temporal diversity", so that agents can observe each other in different modes for variable durations.

In natural evolution and learning, this diversity is provided by massive asynchronous parallelism. On a sequential processor, the same effect may be approximated by a sequence of interleaved phases during which agents observe each other in different modes.

6.6.4 Mutual observation in stages

An observed agent d should make a transition between training and operational modes while its observer r is in training mode. The roles are then reversed: the agent which we called d plays the role of r and vice versa.

Figure 6.3 shows a possible timing pattern for mutual observation. Each agent's timeline is shown as a vertical line which is "interrupted" by an "interim operational phase" (*iop*). This is the part of the timeline with the filled rectangle in the diagram.
During the interim operational phase (iop) of any observed agent d, the fault injection agent is permitted to place faults in d's object-level so that its "normal" reaction to an anomaly (including its


A
B
t0
t1
t2
t3
t4
t5
t6
d1
d2
d1
d2
Key:
A, B : agents; d1, d2: duration constants
t1, t2, t3,... : transition timepoints
training mode
operational mode
agent's transition to operational mode
agent is responding to an anomaly
X - - - - -> Y X observes Y

Mutual observation for 2 agents: initial training between t0 and t1; subsequent training between t1 and t6 (bold horizontal lines). Transitions from training to operational mode are highlighted with an unfilled rectangle and are observed by the neighbouring agent.

Figure 6.3: Mutual observation in stages

self-repair) can be observed by the agent in training phase r. Thus, the anomaly-detection and self-repair process of d can be incorporated into r's model.

To ensure agents have an opportunity to observe each other in both modes, it is not necessary to follow a strict timing pattern. The switching between modes may happen at random timepoints, provided it follows temporal constraints. In particular, the following constraints apply:

1. Only one agent may be in interim operational phase at any time.

2. The phase durations d_1 and d_2 in Figure 6.3 should be long enough to be called "phases"; e.g. a duration of 1 or 2 cycles could be regarded as a fluctuation, not as a phase. Otherwise, they need not be constants and they need not be the same for each agent.

Formal temporal constraint networks [Dechter et al., 1991] as applied to autonomous control systems (e.g. [Muscettola et al., 1998]) may be relevant in a more realistic version of the current implementation. Vulnerabilities of unnecessarily strict timing patterns are discussed in Chapter 12.

6.6.5 Limits of mutual observation

The observation process (and hence the resulting model) is not expected to cover *everything*. For example, the training phase now has an additional class-discovery subphase, which is not recognised by r as being different from "normal" training phase. Similarly there are smaller fluctuations associated with transition detection itself. This is not important for survival in our scenario, however.

Similarly, r can only observe the meta-level of d as it responds to problems in its own object-level; it cannot observe d's detection and response to problems in r's meta-level. To do this would require a 3-agent configuration, requiring a more complex mutual observation scheme.

The limits of a 2-agent system may be minimised by ensuring that the same meta-level code is used by an agent to observe both its own object-level and the other agent's meta-level. If d were then to execute faulty code while observing a meta-level, the same code is also likely to be executed during d's monitoring of its own object-level (which is observable by r). We have implemented this in practice whenever possible.

If a refined model of meta-level operation is required, a parallel implementation may be advantageous. This is considered in Chapter 12.

6.6.6 Single agent meta-level training

A single agent system may also acquire a model of its meta-level by oscillating between training and operational modes. For example, it may allow itself to be attacked in a "non-fatal" way during an interim operational phase and accumulate trace data on its response pattern. It could then switch back to training mode and form a model of its meta-level from this history of traces.

If a single agent acquires a model of its meta-level, however, this does not enable it to detect a disruption to its anomaly-detection process while in main operational phase (although it may help in some kinds of meta-level attack).

We did not implement meta-level training for a single agent because we are not intending to compare single agent reflection with distributed reflection (see Chapter 4, Section 4.3.3. It is not necessary to do this because it is clear that a single agent architecture has disadvantages that can be overcome by a multi-agent system.

6.7 Autonomous Recovery

Self-repair is necessary for survival and must also include meta-repair (repair of the repair mechanisms), since the environment allows *any* part of the software to be damaged. Because the architecture has to be shallow, however, there is only a very rudimentary form of diagnosis and repair in our prototypes. It may be deepened later to include more complex diagnostic reasoning.

For self-repair or any other kind of self-modification we require a component map as described in 4.2.5. In SIM_AGENT, an agent has a representation of its currently operating rulesystem in its database. It can access and modify its rules in a causally connected way. The rulesystem is represented as an ordered list of rulesets, each containing rules in the order of execution. Each ruleset in the list is executed sequentially in every agent-cycle (depending on parameters, a ruleset can be executed zero or more times in a cycle - the default is once). For example if an agent removes a ruleset from one position and inserts it into a different one, the agent may change its behaviour (because the order of execution is changed).

For repair, a copy of the correct rulesystem is held as a "backup", so that rules or larger components from it can be copied and restored into the correct position in the executable list (that is, the component map).

6.7.1 Repair agents

In our prototypes an agent does not directly repair itself. Instead each agent has a subordinate "repair-agent" which holds a correct "backup" copy of the agent's rulesystem. When requested, the repair-agent restores a component to its main agent's rulesystem in the correct position. "Subordinate" means that the repair agent acts only as requested by its controlling agent. (Such subordinate agents are discussed in more detail in Chapter 11 where they are called "service agents"). Although subordinate in the sense of providing a service, the repair-agent is part of the main-agent's meta-level because it is concerned with the internal world. (The object-level is concerned with the external world).

Repair is initiated by the main agent on detection of an anomaly in one of the following:

1. in the agent's own object-level;

2. in its neighbour's meta-level;
3. in its neighbour's repair agent (also part of the neighbour's meta-level)

In situation (1), an agent sends a message to its repair agent and suspends its own anomaly-detection and vehicle control activities. When its repair agent receives the message, it determines which rule in the main agent is most likely to be damaged (based on the missing event(s)), then it selects a replacement from its backup rules and inserts this into the main agent's rulesystem (which is in the database). It will conclude that there is no need to repair anything if it finds that the original rule exists in the correct position (it has not been deleted). In other words, it is a false alarm. An object-level repair sequence is shown in Figure 6.4.

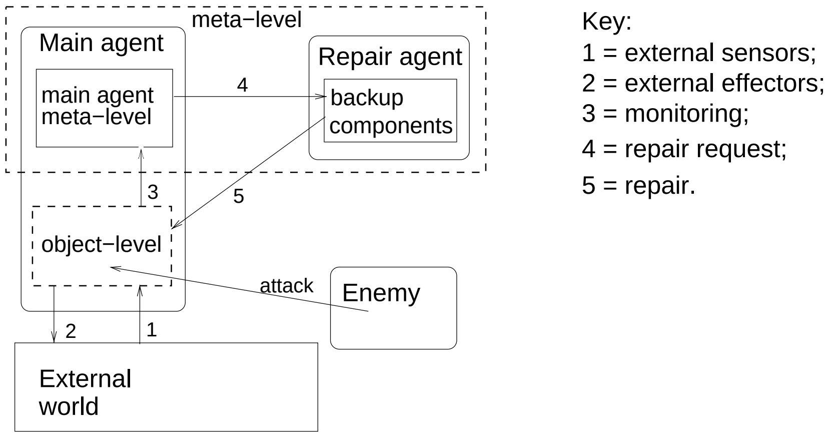


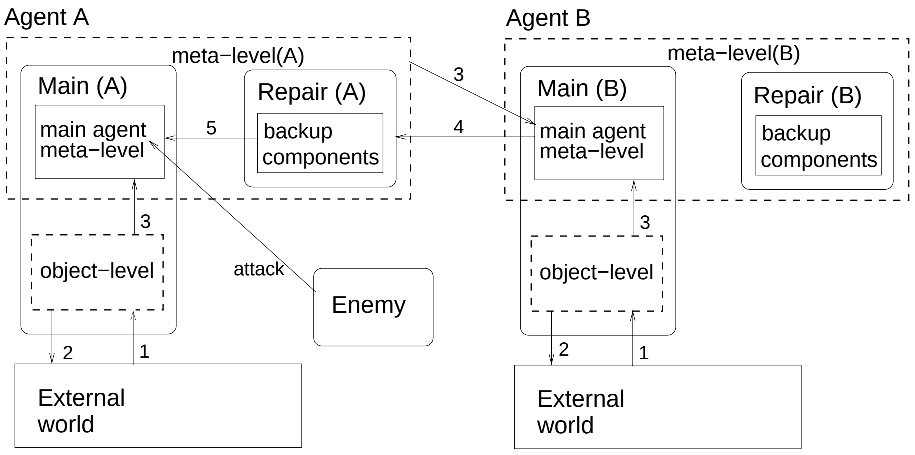
Figure 6.4: Repair of an object-level component

In situations (2) and (3), the main agent repairs its neighbour directly, since it has backup copies of all its rules (including its neighbour's repair rules). Direct repair assumes that the agents are not independently designed, since they must know about each other's code.

6.7.2 Diversity-tolerant repair

An alternative version was implemented where the agents did not require knowledge of each other's code, but they were required to know the names of each other's components. The new version allows some design and implementation diversity which is normally advantageous (see Chapter 12 for a discussion on this). An agent detecting an anomaly in its neighbour's meta-level activates its neighbour's *repair agent* by sending it a message with details of the anomaly, which normally includes the missing component's name. The repair-agent uses this information to determine which component needs to be repaired. For meta-repair (repair of a repair-agent), a repair request is sent to the *main agent*, which has backup copies of its repair-agent's components and can restore them on demand.

Figure 6.5 shows the sequence for repair of a "main" agent in simplified form.
Agent B detects an anomaly in A's meta-level (arrow labelled "3"). Note that B monitors *all* of A's meta-level. It has access to the event records of the main agent and its repair agent. (The repair agent is normally in a "waiting for request" mode). If the anomaly report indicates that a component is missing from A itself (called "main(A)" in the diagram), then B sends a request to A's repair agent, which attempts to repair A's meta-level. If the anomaly indicates that a component is missing from A's repair agent, the request is sent to A. An agent and its repair agent both share the same *receive_activation* ruleset which processes incoming repair requests.



Key:
1 = external sensors; 2 = external effectors; 3 = monitoring;
4 = repair request; 5 = repair.

Figure 6.5: An agent repairs its neighbour's meta-level

On completion of the repair attempt, a message is sent to the requesting agent giving the result, which is one of the following: successful, unsuccessful (e.g. the required component is not available) or unnecessary (false-positive).

6.7.3 Static monitoring

If a backup rule in a repair agent is deleted, this will not be detected until an agent attempts to repair itself or its neighbour using the backup rule (it finds only then that the required component is missing). This limitation is unavoidable due to the nature of execution monitoring: an error can only be detected in something that is executing (or should be executing). One way to overcome this problem is to monitor database access patterns in addition to execution patterns.

An alternative way to avoid the above problem is to monitor the *static* representation of the rulesystem (that is, the component map) for possible anomalies. Its normal state might be copied and continually compared with its actual state.

We decided against this method because it does not require learning (unless the rulesystem is being continually modified as part of normal operation, which is not the case in our prototypes).

6.8 Results

As stated in Chapter 4, Section 4.3.3 on methodology, the results are not in a numerical or statistical form but instead confirm or refute the hypothesis that a certain kind of architecture can survive in a hostile environment.

Appendices A and B give output traces for the damage-resistance prototype along with an explanation on how to interpret the traces and a user guide to its different parameters.

The first trace in A.2 shows that a single agent can perform the primary task in a harmless environment. The second trace in A.3 shows that the hostile environment really does disrupt the primary task if the meta-level is not active. The system "dies" in a short number of cycles, either because it falls into a ditch or one its critical variables is not prevented from reaching 0.

The traces in Appendix B show the operation of a distributed architecture. We can conclude that the system is robust in this particular environment by observing from the trace that:

- The values of the critical variables (treasure and energy) are stable (treasure values slowly increase as in the friendly environment)
- There is repeated recovery from attacks against critical meta-level components, including those concerned with anomaly-detection and self-repair.

Also interestingly the system recovers from an attack against its internal sensors without external intervention, although it leads initially to many false-positives (last trace in B.4).

As would be expected, a short training phase is associated with some false positives initially, but they gradually diminish as the “lifetime” of the system progresses. This is because a limited amount of “learning” still happens during operational phase. By cycle 300, false positives occur very rarely.

6.9 Conclusions

This chapter has supported the main thesis statement by providing a working prototype of the kind proposed by Minsky in Section 1.3. During the experience of prototype development, the following issues became apparent, which would not have been known otherwise:

- The need for temporal diversity or “noise” in order to provide opportunity to observe and learn about a wide range of normal behaviours. (Section 6.6.3).
- The advantage of re-using the same code for different meta-level roles (Section 6.6.5).

6.9.1 Contributions to fault- and intrusion-tolerance

In Chapter 5, Section 5.3.1 we maintained that the ability of a system to represent and reason about its internal processing is a major advantage that the cognitive systems paradigm can offer fault-tolerance. We have seen that this prototype does in fact involve reasoning about the behaviour of the system. This is explained in Sections 6.5.1 and 6.4.5. The response to an anomaly can easily be enhanced to include planning and diagnosis, since the architecture and representation already supports this.

The prototype has also demonstrated the use of internal sensors that are not restricted to measuring certain kinds of behaviour (usually specified as agent inputs and outputs) as would be the case in an agent team (Section 5.3.3). Similar restrictions tend to be assumed in agent-based intrusion detection, as in the MAFTIA intrusion tolerance system.

We have also seen that an agent’s reasoning about the meta-level of a neighbouring agent has the same form as its reasoning about its own object-level. Therefore it is reflective as required in Section 5.3.1. There is no boundary between one agent and another. As far as possible we re-use the same code for both functions in order to maximise observation while minimising additional complexity (Section 6.6.5). This also contrasts with the MAFTIA intrusion-tolerance architecture, in which separate components perform the different meta-level functions (Section 5.3.5).

Note that the boundary between agents is a somewhat artificial one as regards the distinctions made by the system itself. The designer needs this boundary, however, for practical software engineering reasons.

6.9.2 Self/nonself boundary

In Section 5.3.2, and in the above section 6.5.1 we maintained that distributed reflection produces a self-model that overcomes the limitations of artificial immune systems. The prototype satisfies the following requirements (in Section 1.2.6 but not shared by artificial immune systems:

- The self model is a set of statements about specific components and therefore allows reasoning about these components. “Nonself” is detected when any of these statements is not true (keeping in mind the above-mentioned qualifications about noise and uncertainty) For example: “Rule X in Ruleset Y always has its conditions checked”. If in cycle n, the rule did not have its conditions checked, the “nonself” is the “silence” of the rule. Furthermore these rules may be *any* part of the system, including the rules making the decisions about self and nonself (thus overcoming the limitation in Section 2.4. (See also the traces in Appendix B to show that the system explains that a meta-level rule action is missing)
- A “behaviour” (or omission in this case) is associated with a component so that the component can be identified and replaced. The system takes self-protective action.

As explained in Section 6.6, the prototype generates a boundary between “self” and “nonself”, although this is limited and many aspects are predetermined at design time. In real world applications, however, it is very likely that significant predetermination will exist. In the next chapter we will discuss the model acquisition algorithm in detail and show how some significant aspects of the model are autonomously generated.

6.9.3 Limitations

The current prototype has the following limitations:

- It tolerates omission faults only
- It responds to any *unexpected* behaviour as if it were hostile. Anomalies may often be harmless.

In Chapter 8 we will extend the prototype so that it has a definition of “quality” of the world and of its own behaviour. Only those anomalies that are associated with a degradation of quality are regarded as “threats”.

Chapter 7

Autonomous Model Acquisition

7.1 Overview

This chapter presents the model acquisition algorithm that was specifically developed to build the prototype architecture. We explain it in detail to enable independent reconstruction of the same or similar prototype.

In accordance with the "broad-and-shallow" methodology, we have chosen initially to build the model on the basis of simple and high-level discontinuities in the observed execution patterns. Later, this may be refined by taking noise and uncertainty into account (using weights and more general pattern matching). The focus on sharp distinctions is similar to the idea of selecting features with maximum "information content" used in many existing concept-learning and data-mining algorithms, for example [Quinlan, 1986], [Setiono and Leow, 2000] and [Lee et al., 2001a].

The last chapter described the training phase as a series of stages marked out by *transitions* between modes. It was important to allow each agent to observe the normal operation of its neighbour's meta-level and its response to an anomaly. In this chapter, We look at the "experience" of a single agent r (for observe*r*) during one of its training phases as it observes the execution of its own object-level and the meta-level of a neighbouring agent d (for observe*d*). Note that the meta-level of d is also active (it is also observing) while it is being observed by r but we are not *describing* d's observation.

In the initial part of the training phase (before the meta-level phase) r builds a model of its own object-level according to the algorithm in 6.5.2.

7.2 Meta-level Observation

Returning to Figure 6.3, an agent r in training mode must be able to detect a transition between training and operational modes of a neighbouring agent d's meta-level. r detects a transition between modes when a number of event records suddenly stop appearing followed immediately by a number of new events that were not observed previously (in other words, when there is a major discontinuity in d's execution trace). The pseudocode is shown in Table 7.1. Among the many options in POPRULEBASE, two execution modes are provided: "run first runnable rule only" or "run all runnable rules". In the former, only the first rule with matching conditions is run. In the latter, all matching rules are run. In Table 7.1 and all other pseudocode presented here, the execution mode is set to "all runnable rules". This means that when there are multiple matches between a rule's conditions and the database contents, a "rule instance" is created for each one of them, and the rule's actions are executed for each instance. For example, the first rule in Table 7.1 could be read as "*for every* event that stopped occurring this cycle that occurred continuously for at least `mode_duration_threshold` cycles ..." The rule variable "event" binds to different instances in the database, creating a rule instance

```
define RULESET discover_classes
  RULE transition_hypothesis1:
  if time in training-phase is at least mode_duration_threshold and
    some events stopped occurring this cycle which
    occurred continuously for at least mode_duration_threshold cycles
  then
    hypothesise that a mode transition has taken place where
    stopped events belong to mode 1

  RULE transition_hypothesis2:
  if a new transition hypothesis has just been made and
    some new events started occurring this cycle
  then
    hypothesise that the new events belong to mode 2

  RULE create_activity_type:
  if number of events in each mode is at least mode_event_threshold
  then
    create an activity type index for the observed agent, with two modes;
    identify current mode of the activity as mode 2;
enddefine;
```

Table 7.1: Acquiring a model of a meta-level

for each consistent combination of bindings. As explained earlier an “event” can be defined on the “summary” or the “detailed” level (meaning execution of a ruleset or a rule respectively).

7.2.1 Modes and activities

The pseudocode above involves two levels of distinction: “modes” and “activity types”. The modes are intended to be *contexts* in which certain kinds of behaviour is expected. Many important kinds of anomaly can then be detected as things that are true in the world but the model says that they should not be true in that context (a physical anomaly such as anti-gravity would be an extreme example, e.g. something does not fall when it is dropped; but it is normal that it should not fall if it not dropped).

The contexts are discovered by the system when something *changes* context. Instead of saying that the whole system has changed context, however, it is useful to find out exactly which events were involved in the context change and which were not. Those involved in the change are labelled together as an “activity type”. We therefore have two distinction levels, which can be characterised as follows:

- “Things which change” vs. “things which stay the same” (what kind of activity changed modes?) This is like separating an object from its background.
- “Old content” vs. “new content” (difference between the two modes).

It may seem overly complex to use two levels of distinction when possibly one would do. The two levels are useful because we wish to identify those groups of events that mutually exclude each other. Hence we must separate those groups of events that have this clear structure from those background events that are not part of it.

7.2.2 Class discovery

In Section 6.5.2 we defined the "signature" as the union of two sets P and T. We can now explain how the pseudocode in Table 7.1 relates to these sets. The ruleset is in two stages:

1. If r detects a transition, it creates a set of all events that stopped occurring, and a set of new events that began appearing. Effectively, events previously belonging to P and N respectively are now moved to T.

2. r then creates a label for all events involved in the transition, that is the union of old events and new events. This label identifies a *type of activity* that has changed modes. The events involved in the newly discovered activity are the new additions to T and can be defined as "activity set i" of T, where i is the activity label.

To show how this works more generally, we define a set of activity *labels* $A = \{0, 1, 2, ...\}$ where each activity has at least two modes $M = \{1, 2\}$. P contains event records that are always observed whether in training or operational mode or whether responding to an anomaly or not. The set T of event records that are sometimes observed is divided into partitions as follows:

1. "Structured": event records that have been associated with certain modes (they switch on and off in clusters)

2. "Unstructured": event records that switch on and off individually (not in clusters) and follow no discernible pattern.

Thisis based the first level of distinction above ("activity types" associated with clearly defined mode changes). The "structured" partition of T may be defined as a function $Struct : T \rightarrow PowerSet(T)$ as follows:

$$Struct(T) = Act\_set(T, 1) \cup Act\_set(T, 2) \cup ... \cup Act\_set(T, n)$$

where $Act\_set(T, i)$ is "activity set" i of T. In our case the unstructured part of T is empty because all intermittently occurring events are clustered into modes, i.e. $Struct(T) = T$.

Each $Act\_set(T, i)$ is further partitioned into m "mode sets" each containing the events that occur in each mode:

$$Act\_set(T, i) = Mode\_set(T, i, 1) \cup Mode\_set(T, i, 2) \cup ... \cup Mode\_set(T, i, m)$$

where $Mode\_set(T, i, j)$ contains all event records observed in mode j of activity i. In our case $m = 2$ for all activity types because boolean modes were found to be sufficient for our requirements (e.g. training phase vs. not in training phase, anomaly-detected vs. not anomaly-detected). Thus the complement of a mode-set within its activity set is the one remaining mode-set. We use the term "mode's complement" as an abbreviation.

7.2.3 Mutual exclusion

Modes of an activity are mutually exclusive. This means that events observed in one mode are never observed in another mode of the same activity. If we have two modes in an activity then: $Mode\_set(T, i, 1) \cap Mode\_set(T, i, 2) = \{\}$ An event may, however, be associated with modes of different activities.

Perfect mutual exclusion is extremely unlikely in the real world. In a minimal prototype, however, the idealised mutual exclusion simplifies anomaly-detection because there are events that are *uniquely associated* with training or operational mode. Furthermore, the system *must* be in a recognisable mode of each activity. If the mode of an activity can be identified, the agent knows what events should or should not be occur in that mode. If there is an activity whose current mode cannot be identified (e.g.

neither mode's event are observed or some combination of both are active), there is a non-specific pattern anomaly called "mode unidentified". Context-sensitive anomaly-detection (including mode identification) is shown in the pseudocode in Table 7.2.

7.2.4 Hypothesis revision

All mode-associated events are expected to *always* occur in that mode. When r detects a transition, it hypothesises that all newly occurring events are essential aspects of the new mode. If in a subsequent cycle it finds that some of an activity's new events do not occur then these event records are removed from the mode set. This is called *hypothesis revision* and may involve deleting an activity completely if its modes do not continue to mutually exclude each other, or if the number of continuously occurring events false below a threshold.

Hypothesis revision is not actually required in this particular implementation because the system is designed so that it is easy to build a crisp model of it, with no unstructured features.

In a scaled up version, noise and intermittent features may be included in a statistical model of behaviour. Alternatively, if execution patterns are very unstructured, a different kind of monitoring might be used. For example, database access patterns may show clearly defined regularities which can be associated with the execution of identifiable components.

7.2.5 Context-sensitive anomaly-detection

Context-sensitive anomaly-detection is shown schematically in Table 7.2.

```
define RULESET Detect_Anomaly;
  RULE detected_inconsistency:
  if there exist two events belonging to different modes of an activity and
    these events are both absent from the trace
  then
    store both events temporarily as "absent";

  RULE identify_mode:
  if there is an activity associated with "absent" events and
    its current mode was last recorded as $m$ and
    the trace contains no events belonging to $m$ but
    contains all events belonging to the mode's complement $m^c$
  then
    set current mode to $m^c$;

  RULE mode_omission_anomaly:
  if there is an activity with "absent" events and
    its current mode is $m$
  then
    record absent event(s) belonging to $m$ as omission anomalies;
    record absent event(s) belonging to $m^c$ as correctly missing;
enddefine;
```

Table 7.2: Context sensitive anomaly detection

7.3 Making the Model Acquisition more General

In this implementation, the following activities were found to be sufficient for the level of anomaly-detection required:

Type 1: pattern-monitoring activity: either training or operational

Type 2: quality-monitoring activity: either training or operational

Type 3: anomaly-response activity: either it responds to an anomaly or it does not.

Quality monitoring activity is not identical with pattern-monitoring and will be discussed in Chapter 8.

Currently these activity descriptions are held in a linear list. However, in a more general implementation, it would be useful to show that some activity types are "children" of others. For example, "anomaly-response" activity cannot happen if the agent is in training phase.

7.4 Related Work

Our model acquisition method has a similar purpose to existing data mining methods for intrusion detection, e.g. [Lee et al., 2001a]. The main difference is that our method was designed specifically to build a model of the different modes of a meta-level. It is "special-purpose". Data mining tends to be general and looks for any kind of regularity. In a deepened version of our architecture, it is expected that such data-mining algorithms can be used. (Deepening is discussed is Chapter 10).

Another related area is that of concept acquisition for mobile robots. Of particular interest is the algorithm used by [Cohen, 2000] for clustering the robot's "experiences" in time, where each experience is a time series of low-level physical events. One of their aims is to get the clusters to correspond to human-recognisable events (such as bumping into a wall). In principle, we are addressing the same kind of problem on a simpler and more abstract level, in that the classes discovered should correspond to the normal operation of human-defined software components (such as checking for an anomaly).

Our context-sensitive anomaly-detection method is related to other work on context-sensitive reasoning in autonomous systems, in particular, the ORCA autonomous underwater vehicle project [Turner, 1993, Turner, 1995]. An important feature of this architecture is the idea of a "current context" and the kind of actions that are appropriate or disallowed within it. This is similar to our concept of a "current mode" within which certain events are expected to happen and others are "prohibited".

7.5 Conclusions

This algorithm provides only minimal capabilities to deal with the likely situations for a very specific problem (acquiring a model of a meta-level). During development, we found that some principles were important and they had to be included in the minimal algorithm:

- Context sensitivity: statements about normal operation are true in some situations and false in others. The system must recognise these contexts in order to detect abnormal operation. In particular, it must recognise the difference between training and operational modes.

- Autonomous generation of contexts: it is not always possible to know in advance what collections of components tend to work together, or what kinds of actions normally accompany each other. Therefore it is advantageous to generate contexts autonomously.

- Discovery of discontinuities as a means to generate new contexts.

Although there are obvious limitations (such as noise intolerance) the broad-and-shallow approach to model-building results in a top-down conceptual model and has the following advantages:

- It is easier to satisfy the requirement to allow explicit representation and reasoning in Section 1.2.6. It also provides more transparency: a conceptual model with sharp distinctions is easier to understand and explain.
- It is better suited to situations where we want to *guarantee* that critical parts of the system are protected (as was our goal stated in Section 6.5.1). If the model includes strict boolean requirements such guarantees are easier than they would be with an artificial immune system "signature".
- Ease of diagnosis: we know which events are missing from what mode. This gives a good idea of which component is faulty (because an event record contains the component identity).

As part of the "deepening" assumption we are assuming that this algorithm can be replaced by a more noise-tolerant one in a scaled up system. We will consider the problems of this in Chapter 11.

Chapter 8

Extending Reflection to include Quality Evaluation

8.1 Overview

This chapter shows how the minimal prototype architecture can be extended to enable an autonomous system to detect negative effects of hostile code on the performance of its software components, even if some of the anomaly-detection components are themselves disabled. In particular, we wish to show how the system's reflective capability was extended to include *evaluation of quality* of its own performance as well as pattern anomaly-detection.

The chapter then identifies and discusses the problems that arose during development of the new prototype.

8.2 A Hostile Modification Environment

A hostile modification environment is a special case of a hostile environment in which "enemy" agents can cause the software of the autonomous system to behave in a destructive (or self-destructive) way. The "enemy" agents are assumed to be deliberate intruders that ensure that destructive behaviour is produced.

The targeted autonomous system must recognise that the new code is hostile and restore itself to its former state by suppressing the execution of the hostile component and carrying out any self-repair actions as necessary. The scenarios are therefore more similar to deliberate insertion of hostile code (such as Trojan horses).

The objective of this stage in the investigation is to ensure that the principle of distributed reflection is not restricted to one particular kind of environment but can also be applied to hostile code insertion scenarios.

8.2.1 Environmental restrictions

In fault-tolerance terminology we are introducing *value* failures into the system instead of merely *omission* failures. Otherwise the failure assumptions remain the same as introduced in Chapter 6, Section 6.4.1. We are excluding intermittent and Byzantine failures. Value failures are persistent. In other words, if an incorrect value is delivered once, it will continue to be delivered.

The particular environment that "matches" the two-agent quality-monitoring prototype has a number of additional restrictions, which we list below.

Arc1: *A single attack only affects one agent*: already explained in Section 6.4.6 and required by any two-agent system.

Arc2: *Destructive code on object-level only*: In this first stage of the hostile code investigation, actual destructive behaviour only happens on an object-level; for example, hostile code on the object-level may cause the vehicle to move into the ditch. The meta-level only experiences *omission* failures (equivalent to deletion). That is, a meta-level component is only replaced by a "dummy" component which behaves externally like the real one but has no code for carrying out its task (such as anomaly-detection or recovery). This means that all quality evaluation is related to the *primary task* only.

Arc3: *Quality degradation is accompanied by pattern anomalies*: We are excluding situations where the system can detect a quality degradation but it cannot detect any unfamiliar patterns that may point to a suspect component.

It is useful to show what kind of situations are excluded by restriction **Arc2**. Destructive behaviour on a meta-level (*non*-omission failures) fall into the following categories:

- False reporting of anomalies and/or quality problems, initiation of unnecessary recovery actions such as the suppression of correctly functioning components. An example would be false-positive "flooding" to prevent the system from operating normally.

- Internal attacks: because meta-level components have more access to the internal software of the system than do object-level components, a single modified meta-level component may contain hidden code to attack other components on the meta- or object-level and be activated on demand.

We have found that restriction **Arc2** is necessary for any two-agent system. This is because a meta-level with hostile code can lie about anomalies and suppress components in its neighbour before it has time to react. In the next chapter we address this problem by including an additional agent and majority voting. The two-agent prototype only detects that the meta-level is *not* responding as it should.

Restriction **Arc3** is a "shallowness" restriction. It may be relaxed for an architecture which can replan around a recurring problem which is undiagnosed (but we are not investigating replanning in this prototype).

8.2.2 Design of Intrusion Scenarios

In the modification environment, deletions and insertions of components occur. In contrast to the random deletion environment, it is not realistic to guarantee recovery from any random modification. In practice, we found that many random modifications had no effect on performance on the primary task (they are not "hostile") but they sometimes caused the rule-interpreter itself to fail. We are excluding an attack that triggers an immediate failure because it is then detected before it can cause more subtle damage. We instead use hand-crafted hostile code which is specifically intended to undermine performance on the primary task.

8.3 Quality Monitoring in a Single Agent Architecture

Before considering the distributed architecture, we first show the architecture of a single agent (a reflective agent in Figure 4.1) and present a scenario in which its object-level is attacked. A single agent is capable of the following types of monitoring:

1. Detection of anomalies in software execution patterns (e.g. an unknown component leaves an event record in the trace)

2. Detection of an actual or imminent quality degradation in the external world (e.g. vehicle's energy level is abnormally low)

3. Detection of an *internal* quality-related problem in one of its own software components; this may be an actual performance degradation (e.g. it is taking too long to perform an action) or the component may be acting in a way that is expected to cause quality degradation in the world or in other components (e.g. interval between repeated actions is abnormally low).

Each type of monitoring requires a model of "normality", which is acquired during the immune system training phase and used to detect discrepancies in the operational phase.

8.3.1 Pattern anomaly detection

The damage-resistant prototype was only designed for absences of expected events in the execution trace. The hostility-resistant prototype allows an agent to detect two different types of pattern anomaly:

1. Omission - characteristic activity associated with an essential component is missing from the trace. In our shallow architecture, critically essential components have to show that they are active in each trace by leaving an "essential" event record. If one of these records is missing, it is assumed that its associated component is inactive.

2. Unfamiliarity - unfamiliar activity appeared in the trace. In our shallow architecture, an event-record is "unfamiliar" if it contains an unrecognised component identifier; in a deeper architecture, "unfamiliarity" may mean a novel sequence of lower-level events such as program behaviour profiles [Ghosh et al., 1999a].

A third kind of anomaly, which we have not included due to the shallowness requirement, is a *non-specific* anomaly in which the whole event trace for one cycle is unusual, or the content of traces over several cycles has some abnormal features (e.g. the same kind of fluctuations repeat every couple of cycles). A nonspecific anomaly is useful only to generate an increased state of alertness, since it does not point to a specific component.

As a source of hostile code, we have therefore focused on "unfamiliarity" anomalies in the form of unknown component identifiers. Although this is a shallow form of anomaly-detection, unfamiliarity in the form of an unknown component identifier may happen in the real world. Some Trojan horses exist as independent "components" (e.g. executable files) in that they have unusual identifiers (e.g. randomly generated names - designed not to be recognised by humans or typical scanning programs). [Gordon and Chess, 1998] summarises the problem of Trojan horses. [Tygar and Whitten, 1996] gives examples of Trojan horses in Java applets.

8.3.2 Quality monitoring

If the quality of the Treasure environment deteriorates after a new component is added then the component should become untrusted and its behaviour regarded as a "threat"; otherwise its behaviour should be accepted as part of the system's signature.

To detect a deterioration in quality, the system must know what "normal" quality is. Therefore the immune training phase now also includes a "quality" training phase (which takes place at the same time as immune training). During this observation period, the agent observes its own internal decision-making and the state of the external world to obtain the following boundary values of "normality":

1. Internal observation of own decisions:

   (a) Shortest observed interval between repeated actions for each kind of action (to help in detecting infinite loops later)

(b) Longest observed durations of each action (to help in timeout detection)

2. External observation:

   (a) Lowest observed values for treasure and energy

   (b) Closest observed distances between vehicle and ditch

If any of these boundaries are exceeded during the operational phase then it is regarded as a quality deterioration. This is a very rough form of quality evaluation but it has been successfully used to detect hostile code, although with some delay. It is easy to extend it in various ways, e.g. by including statistical methods.

8.3.3 Quality monitoring is part of the meta-level

In our implementation, quality-monitoring is considered as part of an agent's meta-level because its function is similar to that of pattern-monitoring. The results of both are used to determine whether there is a problem or not and whether it can be diagnosed. They both act as triggers for internal responses, which involve *modification access to executable components*, i.e. suppressing or repairing rules. Consequently it makes sense that they should be considered together as either fully trusted or not fully trusted by the same agent.

Some object-level rules for selecting actions (such as which treasure store should be the next target) also involve a degree of quality evaluation (e.g. if the energy level is very low, move towards the energy store). We can compare meta-level quality-monitoring with object-level quality-monitoring as follows, using the example of energy-level:

Object-level: if the energy-level is low then make the energy store the next target for the vehicle.

Meta-level: if the energy-level is *abnormally* low, then something might be wrong with the software; take *internal* action to suppress execution of untrusted software components, and restore (repair) trusted ones.

Any component that triggers actions in the internal world (that is actions on the system's own components, such as suppression or repair) is included in the meta-level. Object-level components are triggers *only* for actions on the external world or the planning of such actions.

In summary, the object-level does not have access to the internal world; the meta-level has access to both. We are not claiming that this is the only way to make the division, however; it is merely conceptually simple from a software-development point of view. An overview of meta-level and object-level components is shown in Table 8.1.

8.3.4 Single agent anomaly response

We assume that the participating agents are "optimistic" in that they will not respond to a pattern-anomaly unless it is accompanied by a quality deterioration. Once a pattern anomaly occurs, however, the new state of the system is not fully trusted until the anomaly has been present for a substantial time-period without undue problems. For example, any checkpointing of trusted data is suspended.

For an *unfamiliar* pattern anomaly the procedure is as follows:

- Wait for some time to see if the quality deteriorates (possibly increasing alertness)
- If a quality problem is detected within a certain time period, *reject* the code by suppressing it (removing it from the current executable list) and if necessary restore the old code by activating the repair agent (if an old component was replaced).

Function	Ruleset
Sense	external_sensors
Meta-level	monitor_state internal_sensors monitor_patterns monitor_quality diagnose_problem suppress_execution repair recover_data
Decide	evaluate_state choose_target
Act	avoid_obstacles avoid_ditch move
Anticipate	next_state

Table 8.1: Overview of meta-level and object-level components

A series of snapshots is shown in Figure 8.1 There is a difference between an *actual* current quality deterioration and *anticipated* quality deterioration (e.g. possible loop). Currently both are treated in the same way, and simply called a "quality-alert". From the point of view of the agent the following conditions can be true:

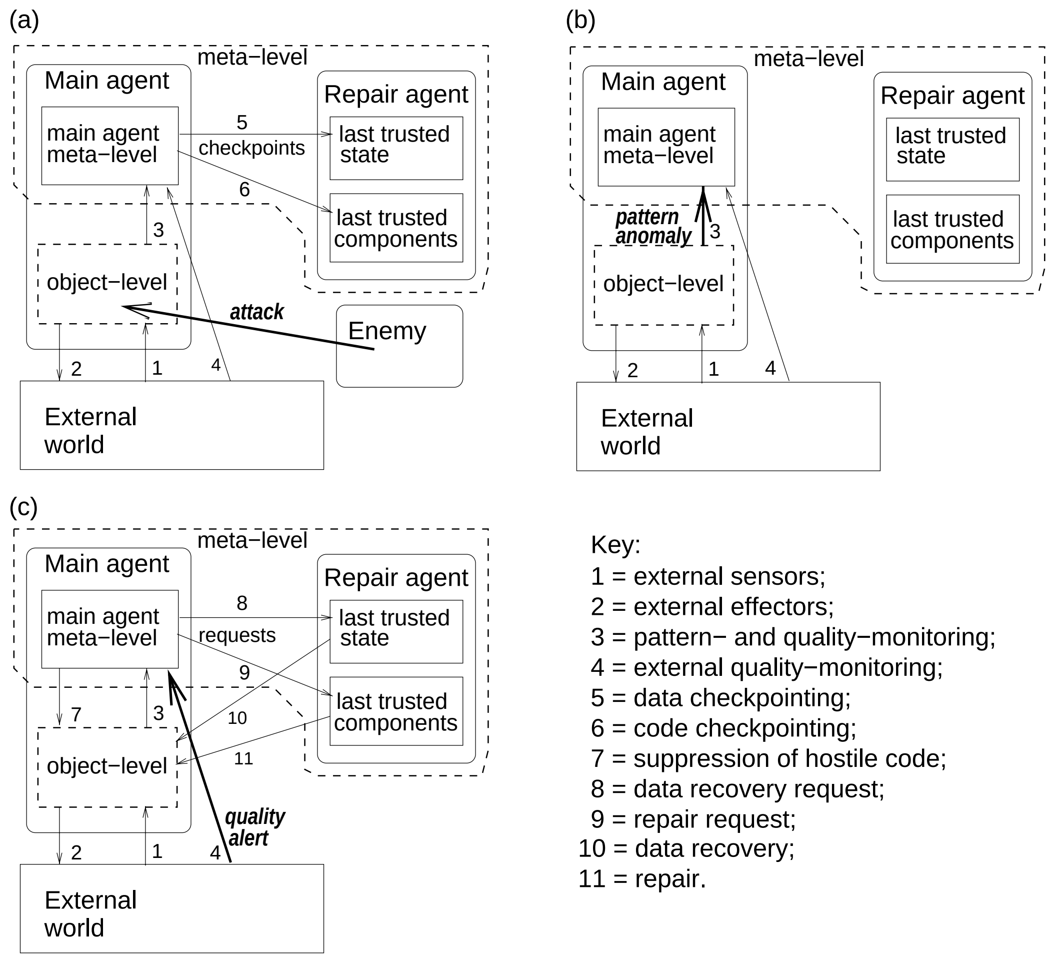
- $\texttt{quality\_alert}(World)$:
  there is a quality problem (that is, an actual or anticipated quality deterioration) in the world
- $\texttt{quality\_alert}(Object\_level)$:
  there is a quality problem in the agent's own object-level
- $\texttt{pattern\_anomaly}(Object\_level)$:
  there is a pattern anomaly in the agent's own object-level

The agent takes action if the following is true:
$(\texttt{quality\_alert}(World) \lor \texttt{quality\_alert}(Object\_level)) \land \texttt{pattern\_anomaly}(Object\_level)$

Our current implementation is a very limited form of response. A more complete solution might involve life-long learning and serendipity exploitation. For example, the following actions may be added to the above:

- If there is an unfamiliar pattern and no quality problem is detected (or there is an improvement - "serendipity") after a specified time period then *accept* the new code: i.e. include its events within the signature.
- Use background concept-learning and data-mining methods to develop high-level descriptions and summaries involving classes of pattern-anomalies and their associations with different kinds of quality deterioration or serendipity.

Future work is expected to include these additional features.



Key:
1 = external sensors;
2 = external effectors;
3 = pattern− and quality−monitoring;
4 = external quality−monitoring;
5 = data checkpointing;
6 = code checkpointing;
7 = suppression of hostile code;
8 = data recovery request;
9 = repair request;
10 = data recovery;
11 = repair.

(a) Enemy attacks during normal operation; (b) Pattern anomaly results, agent stops checkpoin
(c) Once a quality problem is detected, the agent uses the anomalous event records to identify and suppress the likely hostile component, then send data−recovery and repair requests to the repair agent as necessary

Figure 8.1: Hostility Triggered Autonomous Recovery

8.3.5 Related work on autonomous response

Our work assumes that autonomous response to intrusion (in particular suppression of execution) is technically feasible. Some recent work has begun to address this problem. For example [Somayaji, 2000] presents an algorithm which slows down a process depending on the extent to which it is "anomalous" (how different it is from "expectation"). This is purely numerical and does not use any form of "quality" evaluation.

[Jansen et al., 2000] discusses automated response using mobile agents in a general way. [Overill, 2001] and [Overill, 1998] addresses technically feasible responses to an intrusion along with legal and ethical problems.

In the fault-tolerance literature, a simple kind of anomaly response is the removal of a faulty component (called a "replica") in a redundant system. For example, [Bondavalli et al., 1995] presents a scheme for removing a replica based on its history of agreement or disagreement with other replicas during voting. A large number of disagreements with the majority would be equivalent to low “quality”. We consider voting in the next chapter.

8.3.6 Implementation summary

A pattern anomaly is recorded as an *anomaly report*, which summarises the anomaly using the following fields:

$$[\texttt{anomaly}\ \ type\ \ agent\_name\ \ component\_name\ \ timestamp]$$

where *type* is either “omission” or “unfamiliar”. For example it may report that an unfamiliar component was active in agent A’s execution at time t1, or that a normally active component was no longer detected in agent B’s execution at time t2. Since we are currently discussing one agent only, the agent-name always refers to the one agent and the component is always one of its object-level components.

An actual or anticipated quality degradation is recorded as a “quality alert” with the following fields:

$$[\texttt{quality\_alert}\ \ agent\_name\ \ problem\ \ timestamp]$$

For example, the agent may find that it is selecting the same target more frequently than normal.

Pseudocode for the rudimentary form of diagnosis used in the implementation is shown in Table 8.2. Suppression and repair modules are outlined in Tables 8.3 and 8.4 respectively. The pseudocode

```
define RULESET diagnose_problem
  RULE identify_disabled:
  if a quality problem was recently detected and
    at least one normally active component recently became inactive
  then
    mark the inactive component(s) as requiring repair;

  RULE identify_hostile:
  if a quality problem was recently detected and
    at least one unfamiliar component recently became active
  then
    mark the unfamiliar component(s) as untrusted;
enddefine;
```

Table 8.2: Outline of diagnosis component

presents only a summary of what the rulesets do. Some details are excluded for reasons of space and clarity. The “executable list” is the sequence of components to be executed by the SIM_AGENT scheduler in the order specified. The “untrusted list” is the list of unfamiliar components whose activity has been correlated with a quality degradation. The “disabled list” contains familiar components whose abnormal *in*activity is associated with a quality degradation.

In our intrusion scenarios (which will be explained in detail later), we assume that a correctly functioning component is replaced by a hostile one. In order to have the intended destructive effect, the enemy agent places the hostile component in the correct position in the executable list and deletes the correct component in that position. (As mentioned above, inserting code at random locations is difficult to test in a controlled way, since it often has no effect). This means that a response always involves both suppression and repair.

```
define RULESET suppress_component
  RULE suppress_target:
  if any component(s) were identified as untrusted and
    no other components are currently suppressed
  then
    remove the component(s) from the agent's executable list
    keep a copy of the component(s) and associated agent-identifier

  RULE restore_component:
  if any component(s) are suppressed for duration D and
    there is no reverse in the quality deterioration
  then
    restore the component(s) to their agent's executable list
    eliminate them from the "untrusted" list
enddefine;
```

Table 8.3: Outline of suppression component

```
define RULESET repair
  RULE activate_repair:
  if any component(s) were identified as disabled and
    no other components are currently being repaired
  then
    send message to repair-agent to restore component(s) to agent's executable list

  RULE repair_completed:
  if reply indicates that repair is complete
  then
    wait for possible quality improvement

  RULE repair_unnecessary:
  if reply indicates that identical component(s) already exist(s)
  then
    eliminate component(s) from "disabled" list
enddefine;
```

Table 8.4: Outline of repair component

Software "repair" is analogous to hardware repair where a faulty component is replaced with a trusted backup. For software, however, replacing with a backup only makes sense if the current component is missing or is not identical to the backup. If the repair-agent finds that there is already an identical component in the position in which the old component is to be re-inserted then it takes no action.

Suppression and repair are not symmetrical operations. Suppression may be reversed by restoring a suppressed component (suppression may worsen the situation). By contrast, we assume that repairing something cannot *cause* additional damage; it can only fail to correct existing problems.

8.3.7 Object-level attack scenarios

We now present two hostile code attack scenarios on the object-level:

- **Scenario 1**: modify the *evaluate_state* ruleset, which evaluates the current situation in order to select the next action (e.g. seek a new target, continue to collect treasure at the current location, continue to seek the current target etc.)
- **Scenario 2**: modify the *external_sensors* ruleset, which collects and summarise information provided by the external sensors.

Scenario 1: To describe the first attack scenario, we give pseudocode outlines of two important rulesets on the object-level which determine the agent's decisions about which target to seek and whether it should interrupt its current activity to recharge its energy level. "Subjective interest" is the value the agent assigns to a treasure store. The first ruleset, shown in Table 8.5 evaluates the external world by revising the agent's level of interest in the various treasure stores. The next ruleset to be executed is outlined in Table 8.6 and uses the revised interest levels to generate a new "motive" as necessary. Simply changing one line of code can have a significant negative effect on quality. Table

```
define RULESET evaluate_state
  RULE evaluate_close_treasure:
  if vehicle has arrived at a treasure target
  then
    adjust subjective interest in the target according to observed value of target

  RULE interest_growth:
  if vehicle is not located at any treasure target
  then
    increase subjective interest in each target by I_growth

  RULE interest_decay:
  if vehicle is located at a treasure store
  then
    decrease subjective interest in it by I_decay
enddefine;
```

Table 8.5: Correct ruleset for evaluating external world

8.7 shows a "hostile" version of the *evaluate_state* ruleset. A single action in the *interest_growth* rule is modified so that the interest level is *decreased* instead of increased. The precise effect depends on the values of *I_growth* and *I_decay*, but the general effect is that the longer something has not been visited, the less interest the agent has in it (until it reaches 0). This is the opposite of what normally happens. The agent cannot observe a store's actual value unless the vehicle arrives close to it. The vehicle does not get close to "uninteresting" stores because they are never selected as targets but are instead treated as obstacles. When the agent's interest in the current target becomes sufficiently low, it leaves and selects a new one. However, it will tend to select the same target because its interest in

```
define RULESET generate_motive
  RULE low_energy_before_target:
  if vehicle is currently moving towards a treasure target and
    its energy level has become low
  then
    interrupt path to treasure;
    make energy the next target

  RULE low_energy_at_treasure:
  if vehicle currently located at a treasure store and
    its energy level has become low
  then
    interrupt treasure collection;
    make energy the next target;

  RULE low_interest:
  if vehicle currently located at a treasure store and
    interest in it has decayed to a critical level
  then
    prepare to move away from treasure store;
    prepare to select new target based on maximum interest;
enddefine;
```

Table 8.6: Correct ruleset for generating a motive

```
define RULESET hostile_evaluate_state
  RULE evaluate_close_treasure:
  if vehicle has arrived at a treasure target
  then
    adjust subjective interest in the target according to observed value of target

  RULE rogue_interest_growth:
  if vehicle is not located at any treasure target
  then
    DECREASE subjective interest in each target by I_growth

  RULE interest_decay:
  if vehicle is located at a treasure store
  then
    decrease subjective interest in it by I_decay
enddefine;
```

Table 8.7: Hostile ruleset for evaluating external world

all the others will have decreased below its interest in the old target. The effect is to produce repetitive selecting of the same target, which the meta-level interprets as a possible infinite loop and hence a quality alert.

An alternative is to change the *interest_decay* rule so that there is a *growth* in the controlling agent's interest in treasure it is currently collecting (instead of a gradual decay). The agent will then continually choose to remain at the current target, even if it is no longer collecting any "objective" value from it. (Objective value is also "consumed" by a vehicle located at it, while the objective value of other stores tends to grow). The agent's meta-level observes that it is spending too much time at the same treasure store and interprets this as a quality problem (timeout).

Scenario 2: The second attack scenario involved interfering with the interpretation of external sensor readings. The sensors themselves are provided by the SIM_AGENT toolkit and cannot (easily) be modified. However, the agent has a ruleset *external_sensors* which extracts the relevant sensor information and stores it in summarised form. The hostile version of the *external_sensors* ruleset is shown in Table 8.8. In the hostile code, the position of the energy source is falsified as shown in the

```
define RULESET hostile_external_sensors
  RULE check_vehicle:
  if new vehicle position can be determined
  then
    update current vehicle position

  RULE see_treasure?
  if a treasure store can be identified
  then
    store its position and dimensions

  RULE see_ditch?
  if a ditch can be identified
  then
    store its position and dimensions

  RULE rogue_see_energy?:
  if energy source can be identified
  then
    set its position to a false value,
    store other details correctly
enddefine;
```

Table 8.8: Hostile ruleset for interpreting external sensors

"rogue" rule in Table 8.8. When the agent arrives to recharge the vehicle, it wrongly believes it is positioned at the energy source. It will attempt to recharge in the wrong position and will only detect a problem when the energy level fails to increase as expected and instead continues to decrease until it is abnormally low.

8.3.8 Results

In test runs, the first problem to be detected was a timeout due to the recharge operation taking longer than expected (the time it takes to be fully charged).

As with the *evaluate_state* scenario, the agent was able to recover after some delay.

The ability of the agent to recover from the situation depends on its distance from the true position and on parameters specified in advance. For example, should the vehicle only be recharged when the energy level is dangerously low or should the threshold be set at a more “cautious” level?

To implement each attack, an “enemy” agent was activated during the agent’s operational phase which replaced the correct ruleset in the executable list with a hostile version. For both scenarios, the meta-level was able to detect a pattern anomaly caused by the presence of the new component, followed shortly afterwards by a quality alert which in turn triggered successful diagnosis and recovery actions. Output traces from the two object-level attack scenarios are given in Appendix C.

8.3.9 Limitations

There are some scenarios for which this (optimistic) method will not work. For example, there is a ruleset for ensuring that the vehicle does not fall into the ditch when moving close to it (last_minute_correction). If this goes wrong, there is no way of detecting a problem until the vehicle is already in the ditch. It cannot detect that it is too close because it *normally* has to get very close in order to arrive at some treasure stores.

8.4 Quality Monitoring in the Distributed Architecture

We now return to the distributed architecture in Figure 4.1(b). In this architecture, a participating agent’s monitoring and repair of its own object-level happens in the same way as in the single agent case. The main addition is that an agent’s internal sensors also have access to the meta-level components of its neighbour.

Anomalies in the neighbour’s meta-level can be correlated with a quality deterioration in the external world as can anomalies in the same agent’s object level. The other major difference is that now we have one agent in control of the vehicle and the other acting as a backup, but they both monitor and evaluate each other.

8.4.1 Increased shallowness required

Due to the increased processing load of distributed quality monitoring, the *pattern*-monitoring was modified so that it processed execution trace data on a “summary” level. The summarised trace data only indicated which rulesets were active in the last cycle. A ruleset is known as a “component”. This was necessary to enable interactive testing with an acceptable response-time. Increasing the shallowness at this point is also reasonable because we already know that the pattern-monitoring and anomaly-detection works on a lower level.

8.4.2 Primary/backup organisation

In the implementation, one agent (the primary) is in control of the vehicle while the other agent (the backup) executes the same identical software internally but does not “act out” its decisions. Hence vehicle control decisions made by both agents should be virtually identical (although there may be slight differences in resulting vehicle position due to the use of random numbers for small differences in vehicle direction to avoid getting stuck in a loop, e.g. if it is between obstacles).

If the backup agent detects a problem in the primary agent, it immediately takes over control. The primary agent can detect quality problems by comparing its internal quality model with actual values (e.g. if an action has taken longer than the longest duration ever experienced or if the energy level has reached its lowest level ever). The backup agent also holds the same model and can compare its expectation with reality. It can also detect anomalies in the vehicle position. For example if it has chosen a target and made a decision on how to move the vehicle, it will expect a certain vehicle position in the next sensory update. If there is a repeated large discrepancy between the actual and expected position of the vehicle, this may indicate a problem (e.g. the primary agent's target selection criteria may have been illegally modified - a scenario which was tested in practise).

8.4.3 Acquiring a distributed quality-model

In distributed reflection, agents acquire models of each other's meta-levels by observing each other in various states during the training phase. For example they should observe each other detecting and recovering from an anomaly so that they have a model of normal execution patterns during these states.

At the end of the training phase, an agent divides up the execution events of its neighbour's meta-level into mutually exclusive classes called modes. The meta-level events are divided into training or operational mode; operational mode events are further subdivided into normal events and anomaly-response events. In other words, if the meta-level is in operational mode then it is either in normal mode or responding to an anomaly.

In the current implementation, meta-level components (rulesets) are clustered together in modules (rule-families) in which related components work together. Within a family, only one "current" ruleset is active in a particular cycle; all others in the family are inactive. This makes it easier for the model-acquisition rulesets to make the required distinctions. (The current design is shallower than the damage-tolerance version because an event simply indicates whether a whole ruleset is active or inactive).

Control is passed between rulesets as the agent enters or exits the various modes. For example, if an agent has just detected a recoverable problem (quality alert and pattern anomaly), it enters "anomaly-response" mode which means that the currently active rulesets transfer control to a set of new rulesets. A rule-family typically contains a ruleset which just "waits" for a set of conditions indicating a recoverable problem. As soon as this happens it transfers control to a diagnosis or recovery ruleset. When the new components have completed their tasks they wait until sufficient time has elapsed for the recovery to complete (e.g. a repair must have been acknowledged and the data should have been rolled back to a trusted state) and then transfer control to the "wait" rulesets.

The complete list of meta-level rule-families is shown in Table 8.1.

8.4.4 Quality training phase

An agent's model of a meta-level now contains two new modes: "quality training" and "quality operational". The quality training mode only takes place within the initial training phase, in which an agent acquires a model of its own object-level and the *training phase* of its neighbour's meta-level. Some time before the first agent changes to operational phase (to be observed by its neighbour), both agents change to "quality operational mode" and do not return to "quality training mode" (in this version).

The reason for not grouping together the quality-monitoring modes with the pattern-monitoring modes is that quality should be treated separately. For example, there may be future extensions in which execution events associated with normal quality should be known as well as those with deteriorated quality. When acquiring this knowledge, the observing agent should not still be learning what constitutes "normal" quality.

At present, the observing agent only detects that certain kinds of response *exist* and that they mutually exclude each other in well-defined ways. In a future stage of our investigation, when quality-monitoring of a meta-level is planned, it will become important to evaluate whether the meta-level's response is “correct” given a deteriorated or non-deteriorated quality. To do this the observing agent must have associated a deteriorated quality with a characteristic response by the observed agent (and normal quality with no response).

In the minimal design we have chosen, both agents have the *same* internal model of quality because their model acquisition code is identical, as is their object-level code. Hence, the agents can only disagree on whether quality has deteriorated if one agent’s quality-monitoring code has been compromised. At present, the only disagreement that can happen is when a meta-level is attacked and it does *not* respond as it should. If the attack involves a quality-monitoring component, it will merely fail to detect a quality problem and will not trigger a response.

8.4.5 Basic attack scenario

The basic attack scenario is in two stages as follows:

1. *Disable* a meta-level component in the primary agent A (replace with dummy code)
2. After a delay: replace an object-level component in the same agent A with hostile code.

When the meta-level component is attacked, the neighbouring agent should detect a *pattern* anomaly, meaning that an unfamiliar component has become active (and the normally active component has become inactive). When the object-level component is attacked, a quality deterioration results. Then the meta-levels’ lack of response can be detected.

The object-level attack is the same as for the single-agent case discussed above. The meta-level attack may involve more than one component simultaneously and we will describe this in detail later. One or more meta-level components are replaced by “dummy” versions. Their interaction with other components appears normal but they contain no internal code; they simply take control and relinquish control in the correct way. For example, if a diagnosis component is activated, it will just give up control again after the normal delay, but will not do any diagnosis. (This is similar to camouflage techniques in Trojan horse attacks, for example, making them look like standard applications [Beattie et al., 2000]).

8.4.6 Situations involving a quality alert

We now justify the selection of this particular attack scenario. From the point of view of any agent, there are four conditions that can be true (in contrast to three in the single agent case):

- $\texttt{quality\_alert}(World)$:
  quality problem in the external world
- $\texttt{quality\_alert}(Object\_level)$:
  quality problem in the agent’s own object-level
- $\texttt{pattern\_anomaly}(Object\_level)$:
  pattern anomaly in the agent’s own object-level
- $\texttt{pattern\_anomaly}(Meta\_level)$:
  pattern anomaly in the neighbour agent’s meta-level

The possible situations involving a quality alert are now more complex and are listed below for each agent. Some combinations are excluded from the investigation because of the assumptions or because they would be redundant. The remaining conditions are tested by the scenario.

For the primary agent the quality alert situations and their responses are as follows:

1. $(\texttt{quality\_alert}(World) \vee \texttt{quality\_alert}(Object\_level)) \wedge$
   $\texttt{pattern\_anomaly}(Object\_level)$

Response: hypothesise that the problem is in the primary agent's object-level; try to identify and repair it.

2. $(\texttt{quality\_alert}(World) \vee \texttt{quality\_alert}(Object\_level)) \wedge$
$\texttt{pattern\_anomaly}(Meta\_level)$
Possible response: hypothesise that the problem is in the backup's meta-level; try to identify and repair it.
Excluded in the current environment because a meta-level can only cause a quality problem in the world (or in an object-level) by executing hostile code. Current attack scenarios only delete meta-level components (restriction **Arc2** above). See also argument below.

3. $(\texttt{quality\_alert}(World) \vee \texttt{quality\_alert}(Object\_level)) \wedge$
$\neg\texttt{pattern\_anomaly}(Meta\_level) \wedge \neg\texttt{pattern\_anomaly}(Object\_level)$
No response: problem cannot be diagnosed because there are no pattern anomalies which can be used to identify suspect components.
Excluded because of restriction **Arc3** above.

For the backup agent, the quality-alert situations and their responses are as follows:

1. $\texttt{quality\_alert}(Object\_level) \vee \texttt{pattern\_anomaly}(Object\_level)$
Excluded because this scenario introduces nothing more than is already covered by an attack on the primary agent. (See also argument below).

2. $\texttt{quality\_alert}(World) \wedge \texttt{pattern\_anomaly}(Meta\_level)$
Response: identify and repair the problem in primary's meta-level.

3. $\texttt{quality\_alert}(World) \wedge \neg\texttt{pattern\_anomaly}(Meta\_level) \wedge$
$\neg\texttt{pattern\_anomaly}(Object\_level)$
No response: problem cannot be diagnosed.
Excluded because of restriction **Arc3** above.

Since only deletions can happen in the backup agent's meta-level then the meta-level cannot itself have caused any quality problems. There are, however, two situations where its inaction can allow a problem to continue.

1. The primary's meta-level has not responded to a quality problem caused by hostile code in its own object-level and the backup's meta-level is also not responding; this is excluded by the assumption that the time interval between attacks on two different meta-levels must be sufficient for the correctly functioning one to repair the one that was attacked (given that it can detect the problem)

2. The backup's meta-level has not responded to a problem in its own object-level, and the hostile code in its object-level is illegally interfering with the actions of the vehicle (it is no longer acting as a passive backup)

The second case is a mirror image of an attack on the primary which the backup should respond to. However, there is no guarantee that hostile code in the backup's object-level will make itself known by interfering with the vehicle; it may not cause any deterioration in quality until the backup agent takes over control during a recovery operation following a later attack.

It therefore makes little sense to use an optimistic attitude when a pattern anomaly is detected in the execution of any component of the backup agent. If the primary detects a pattern anomaly in the meta-level, we assume it is best to correct it immediately, and similarly if the backup's meta-level detects an object-level pattern anomaly. Hence, it is not useful to include these situations in the current scenarios, since the pessimistic attitude has already been covered by the damage-tolerance prototype and it would be redundant to include examples of it here.

8.4.7 Data recovery

During normal operation, both agents regularly checkpoint the most essential data to their repair-agents. After an agent has suppressed or repaired something, it rolls back the data by requesting its repair-agent to broadcast the old trusted data to both agents. Once they receive the broadcast, both agents resume processing from the same initial state and their subsequent decisions are synchronised as required.

The last trusted versions of the software may also be checkpointed regularly. In our scenarios, the last software version is always the initial one, so there is no need for regular checkpointing. In an application where there are frequent software changes, an upgrade may be "accepted" after a period of intensive quality monitoring. It may then be checkpointed as a "last trusted state" of the software. The last trusted upgrade (and not the initial components) are then restored in the event of an intrusion or serious quality problem with a subsequent upgrade.

8.4.8 Meta-level attack scenarios

Each scenario involved the disruption of a different meta-level component, followed shortly afterwards by an insertion of hostile code on the object-level of the same agent. Attacks on the following meta-level components were tested.

1. a pattern-anomaly detection component
2. a quality evaluation component
3. a diagnosis component
4. a code suppression component
5. a repair component, either in a main agent or in a repair agent
6. multiple meta-level components (including *all* components in the meta-level)

8.5 Results

In each meta-level attack scenario listed above, the backup agent was able to detect a quality deterioration in the external world some time after detecting an anomaly in the primary's meta-level (it cannot detect any anomaly in the primary's object level). The initial quality problem was a warning in the form of a vehicle position anomaly. This happens because the hostile *evaluate_state* ruleset causes the compromised agent to select a different vehicle target from that chosen by the backup (in its "private" decisions). The actual vehicle position (as controlled by the faulty code) is now different from that anticipated by the backup, which leads the backup to hypothesise that there is a problem.

Depending on which components in the primary's meta-level were disabled, the primary may also detect a vehicle anomaly, but its capability to diagnose and respond to it may be impaired.

Not all meta-level attacks actually disabled the primary agent. For example, a quality-monitoring component that is disabled may be one that detects problems in the external environment such as low treasure. In this case, the primary agent can still detect a vehicle anomaly, correlate this with a pattern anomaly in its own object-level and use the information in the pattern anomaly report to identify and suppress the hostile code.

8.5.1 Multiple simultaneous meta-level attacks

In Appendix C, an output trace showing an example recovery from a meta-level attack scenario is shown in C.3. In this attack, *all* meta-level monitoring components (including quality evaluation and pattern monitoring) were replaced with useless "dummy" versions. Hence it is a variant of scenario type (6) above. As expected, recovery from multiple simultaneous attacks required more time, since it takes longer to repair multiple components. Although the system "survives" the vehicle treasure level remains lower than normal for some time after recovery.

8.6 Conclusions

The detection of "nonself" has now been enhanced to include the following:

- Detection of anomalies that are not just omissions but indicate a new kind of behaviour; ability to suppress the new behaviour if necessary.
- Reasoning about the *meaning* of the state of the world and the system state, instead of merely about execution patterns with no evaluation of what they mean;
- Detection of deteriorating *quality* of the world and the system's internal state;
- *Anticipation* of quality degradation and possible failure (non-survival).

For practical reasons, the sensed information had to be on a "summary" level (but we do not believe that this makes a difference to the main argument).

8.6.1 Limitations

A major limitation of this prototype is the need for precise synchronisation between the primary and backup agents so that the backup can detect anomalies between its expected next vehicle position and the actual vehicle position produced by the primary (see Section 8.4.2). The ability to detect such anomalies is important. However, its dependence on precise timing patterns makes the whole system vulnerable in other ways. For example, an attacker may use the predictability of the timing to disrupt the system (e.g. by slowing down one agent). In general, any system that can be modeled as *synchronous* (defined in Section 5.2.3 as having known bounds on processing delays, clock drift etc.) is vulnerable in this way. For example, hostile code may delay a message until just after a timeout, and the process will be wrongly assumed to have crashed. Alternative architectures involving diversity and relaxed timing constraints will be considered in Chapter 12.

Another problem is that an omission anomaly in a neighbour's meta-level is correlated only very indirectly with a quality problem in the external world. It would be more effective to evaluate "quality" of a meta-level directly. For example, it may be possible to infer that a meta-level has failed by comparing its decisions with that of other agents (assuming that we have a larger number of agents). We did not implement this because it is already used in some intrusion detection systems, in particular, [Dacier, 2002] which was discussed in Chapter 5. According to the "deepening" assumption, we assume that the current "shallow" methods can be replaced by more complex ones (such as those used by MAFTIA) while still preserving the esential features of distributed reflection. "Deepening" the architecture is discussed in Chapter 11.

An unavoidable problem with a simulation environment is that the "strength" of the enviromental restrictions ("failure assumptions") has to be increased as the architecture is enhanced. Although as much as possble is included within the rule-system (and hence can be attacked), there are some low-level utilities provided by the SIM_AGENT toolkit itself that are not subject to interference. For example, it is not practical to attack the mechanism for accessing an agent's database because this it is an in-built utility and not within the rule-system. It would be necessary to attack the simulation software itself, which is not workable in practice. This problem is discussed further in Chapter 11.

8.6.2 What remains to be done

So far we have only implemented two agents. This has serious limitations if an agent's meta-level "lies" about the existence of anomalies. In the next chapter we present the second extention to the basic architecture, involving an additional agent.

Chapter 9

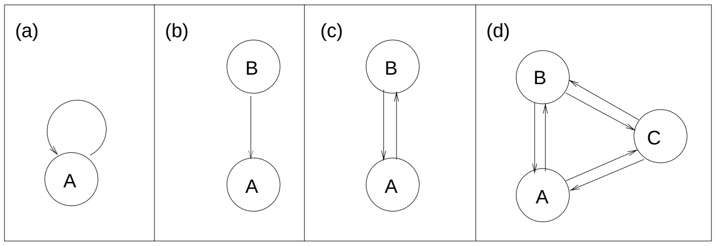
Increasing the Number of Agents

9.1 Overview

This chapter presents the second extension of the minimal prototype in which we include an additional agent and relax some of the restrictions applying to the two-agent system. The three agent system uses some concepts from fault-tolerance introduced in Chapter 5. The objectives of the third prototype are as follows:

1. to show that the concept of distributed reflection is not limited to two agents and can work in more general scenarios;
2. to show that the most serious limitations of a two agent system can be overcome;
3. to understand more clearly the relationship between cognitive architectures and distributed fault-tolerance methods such as voting and consensus;

The three-agent architecture is of the form shown in Figure 3.1(d).



(a): A's meta−level monitors itself

(b): "Open" distributed reflection: a second agent B monitors A's meta−level.

(c): "Closed" distributed reflection: A and B monitor each other's meta−levels

(d): "Closed" distributed reflection with an additional agent C

Figure 9.1: Reflective configurations used so far

9.2 Resisting hostile code on a meta-level

We use the term "hostile" to indicate any code that has destructive effects (not just "malicious" code). In particular we address the situation where hostile code detects a non-existent anomaly (or responds without detecting any anomaly) thus causing it to suppress correctly functioning components.

The two-agent implementation has a simple conflict-resolution mechanism that works as follows: the first agent to detect an anomaly in its neighbour begins its response by setting its neighbour's status to "failure" and taking over control from it if necessary. An agent in "failure status" has its anomaly-detection suppressed in order to avoid a situation in which agents "fight" for control of the vehicle. A conflict would take the form of an agent repeatedly overriding the control of the other and might be deliberately instigated as a kind of "denial of service" attack.

The "failure status" mechanism causes no problems if a meta-level is only disabled in an attack (as in the previous implementations). The worst case scenario is that a disabled meta-level A will not detect a real anomaly in its neighbour B and will fail to take over control when it should.

By contrast, if a meta-level contains hostile code, it can *immediately* detect a non-existent anomaly and take control of the vehicle with no delay. This leaves no possibility of correction within the system itself because the only remaining agent has its monitoring turned off. (We excluded this scenario in the two-agent hostile code recovery system).

For example, suppose that hostile code in an agent A's meta-level falsely reports an anomaly in B's meta-level. Then A will suppress B's monitoring and take control of the vehicle if it is not already in control. This leaves it in a position to execute further hostile code while in control of the vehicle. Even if hostile code was not inserted on the object-level at the time of the first attack, there is an arbitrary amount of time for this to happen as part of a subsequent attack. (Note that an "accidental" false positive for which there is no hostile code does not present this problem; it would only cause an unnecessary switch from one agent to another).

As a solution to the above problem, three agents can determine a response based on a majority vote. Table 9.1 shows a simple example. In the table, each row indicates the status of an agent's decisions

Agent	A	B	C
A	0	1	0
B	1	0	0
C	1	0	0

Table 9.1: Example of conflicting anomaly reports

on whether there is an anomaly or not. The top row states A's decisions about all other agents. The value of "1" under "B" indicates that A detects an anomaly in B's meta-level. The value "0" in the remaining positions means that A does not detect an anomaly in the remaining agents. In accordance with the distributed reflection principles, an agent's meta-level does not look for anomalies in itself, meaning that the diagonal is always 0.

The values in the table show what happens when A's meta-level contains hostile code. It wrongly reports an anomaly in B. However both B and C say that A contains an anomaly but B does not. Therefore the conflict can be resolved by majority voting. There remains the problem of how B and C know about each other's decisions and which one should do the responding. A simplified solution to this is presented in the next section.

If we assume that only one agent of the three can be attacked with hostile code simultaneously and that the other agents are operating correctly (including their capability to send and receive messages), then a correct decision based on a majority vote is always possible if the individual agents can detect an anomaly correctly, with no "harmless" false-positives or negatives. This follows because the two situations where an agent can report a non-existent anomaly are the two cases above, where the compromised agent indicates that either one or both of the remaining agents are faulty respectively. In both cases the remaining agents must agree that the hostile agent is faulty, given the above assumption. It should be noted, however, that this is a strong assumption of *independence* of faults

[Laprie et al., 1995].

9.2.1 Limited fault-tolerance

As mentioned in Chapter 6 we are excluding "Byzantine" failures [Lamport et al., 1982], where an agent can lie to some agents but not to others (or it may lie in different ways). We are assuming that the communications infrastructure operates independently of the participating agents so that an "anomaly-detected" message is broadcast to all agents and the sender identifier is appended to a message in a way that cannot be influenced by the sending agent itself. It is reasonable to exclude this kind of failure because we only wish to show the interface between fault-tolerance and distributed reflection. Inclusion of Byzantine failures would involve a duplication of work already done in the fault-tolerance community, where this kind of problem has already been investigated in depth.

We are also initially excluding situations where further consequences of the initial fault can interfere with the voting procedure in an arbitrary way. For example, if an agent is executing hostile code, it may subvert the voting procedure and take over control before a non-faulty agent can identify and suppress it. This situation would also be covered by existing Byzantine agreement algorithms. Our prototype does, however, present a *cognitive systems* approach to this problem, which has some advantages stated in Section 5.3.1. We will consider later how it may be integrated with fault-tolerant algorithms.

9.3 Resisting simultaneous attacks on different agents

The following assumptions held for the two-agent system: First, a single attack only affects *one agent*. Secondly the time interval between attacks is above the minimum time required to allow detection and recovery from the problem when the software operates correctly. In other words, it should be physically possible for it to recover. The second assumption still holds, but the first one can be relaxed if the number of agents can be increased.

"Simultaneous" does not necessarily mean "at the same instant". It may mean that more than one attack happens in rapid succession. However, we assume that there is insufficient time to begin recovery from the previous attack, so that the attacks are approximately "simultaneous". This is simply the second assumption above in a different form, namely that recovery from one attack is not interrupted by a subsequent one.

The likelihood of two separate agents being attacked in rapid succession is reduced if agents are based on independently designed software. We will discuss the advantages of software diversity of this kind in Part 3, along with the related issue of "independence".

9.3.1 Simultaneous disabling attacks

We first consider the case where an attack only causes omission failures on the meta-level; that is, the meta-level is disabled so it cannot detect an anomaly and also cannot send messages or respond in any way. If D is the number of simultaneous disabling attacks that can be tolerated and n is the number of agents, the following relationship should hold:

$$D = n - 1$$

assuming that every agent monitors every other agent as shown in Figure 3.1(d). This only says that one agent should be left intact to diagnose and respond to the problem. We will show later how this works in practice.

9.3.2 Simultaneous hostile code attacks

Given the restrictions in Section 9.2.1, the number H of simultaneous *hostile code* attacks on the meta-levels of different agents that can be tolerated is:

$$H < n/2$$

This is to enable a majority vote. It assumes that every agent monitors every other agent. For example, if n = 2 it cannot tolerate any hostile attacks; if n = 3, only one hostile code attack can be resisted; if n = 5 then 2 agents can give false reports and so on. If there is one hostile code attack, there cannot be a simultaneous disabling attack, since a majority vote requires the remaining agents to be operational. We will discuss later how an n-agent system determines whether to wait for a majority vote and whether to respond immediately.

9.4 Environmental restrictions

We now state some necessary restrictions on the environment for an n-agent system (where $n > 2$). They are called **Arc1**, **Arc2**, etc. for *architecture* related).

Arc1: The maximum number of agents that can be attacked simultaneously with disabling attacks is $n - 1$

Arc2: The maximum number of agents that can be attacked simultaneously with hostile code is $n/2 - 1$.

Arc3: If hostile code intrusions are combined simultaneously with disabling attacks on other agents, then they are all regarded as hostile code attacks.

Arc4: If two “simultaneous” attacks occur in rapid succession, there is insufficient time to begin recovery from one attack before the next one occurs.

Arc5: If hostile code causes an agent to lie, the agent tells the same lie to all other agents (no Byzantine failures).

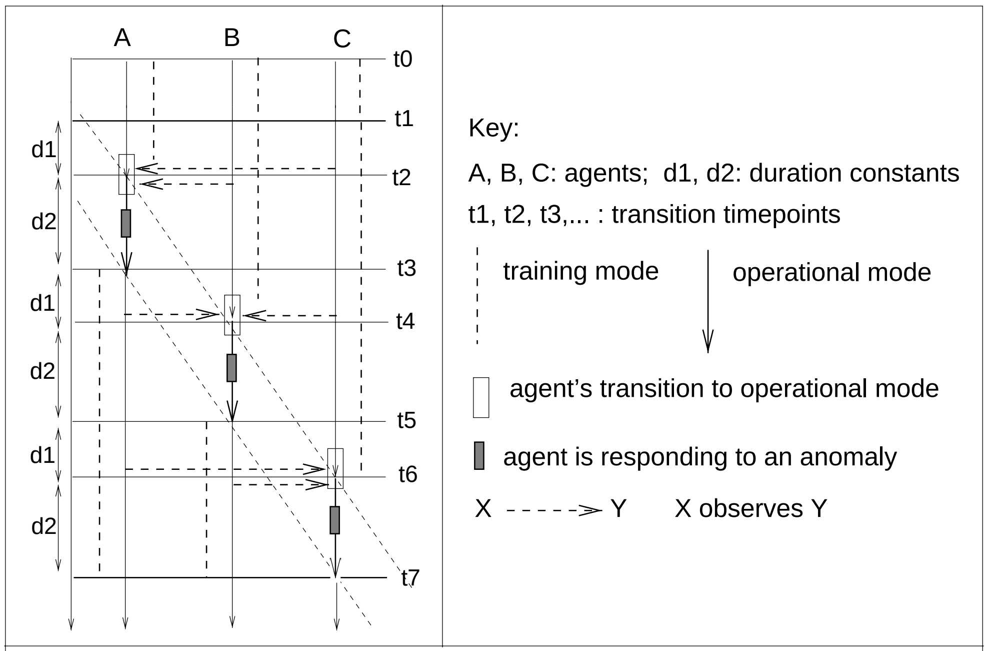
Restriction **Arc1** was already stated in a more specific way in section 6.4.6 for two agents.

9.5 Design of a three-agent system

9.5.1 Training phase structure

For three agents, the training phase has to be extended to enable each agent to observe the training and operational phases of all others. A training phase design for n agents is shown schematically in Figure 9.2. An observation window is a concatenation of two segments labelled d_1 and d_2 in the diagram and represents the minimum duration of an uninterrupted training phase required to observe the different modes of an agent’s behaviour. (In practice, a continuous training phase segment is often longer than this, as can be seen in the diagram).

Both segments of the window must be sufficiently long to allow observation of (a) an agent making a transition from training to operational mode and (b) the agent responding to an anomaly during operational mode. Normally $d_1 < d_2$ because we are more interested in the different submodes of operational mode than in the training mode (although this need not always be the case). d_1 only has to be long enough to enable recognition of a “mode” (as different from a fluctuation). It was defined in Chapter 6 as the “mode duration threshold”. d_2 is the duration of the “interim operational phase” (iop) and may be increased depending on the different kind of things we want to observe in this phase.


A
B
C
t0
t1
t2
t3
t4
t5
t6
t7
d1
d2
d1
d2
d1
d2
Key:
A, B, C: agents; d1, d2: duration constants
t1, t2, t3,... : transition timepoints
training mode
operational mode
agent's transition to operational mode
agent is responding to an anomaly
X - - - - -> Y X observes Y

Mutual observation for n agents where n = 3. Initial training between t0 and t1. Subsequent time in training mode is divided up into n−1 observation windows, one for each agent observed. An observation window has duration d1 + d2. Transitions from training to operational mode are highlighted with an unfilled rectangle and are observed by all other agents.

Figure 9.2: Staggered observation windows in training phase

9.5.2 Training time scalability

If d_T is the total duration of the training phase, d_I is initial training phase duration, w is the observation window duration and n is the number of agents, then

$$d_T = d_I + nw$$

This can be seen from the diagram for 3 agents but it can easily be extended for any n agents. d_I is not shown but its ending is marked by t_1 the first transition timepoint.

The implementation of the extended training phase for three agents behaved as planned, with no new problems being encountered.

9.5.3 Response based on majority vote

Since in the closed reflective architecture there is no single global observer, we must consider how a majority vote "causes" a response. We divide this up into the following subproblems and their provisional solutions:

- How does the voting mechanism work?
  When an agent detects an anomaly it broadcasts this to all other agents. If the agent receives a threshold T number of messages *which agree with its own decision* on whether there is an anomaly or not, then the agent believes there is sufficient support for a response.

- How does an agent know the value of T?
  The value of T is a parameter depending on the number of agents and can be varied depending

on whether a fast "unthinking" response is desirable or a slow "cautious" one. It may also be learned during an extended training phase (see below). In a 3-agent system it is trivial however (T = 1).

- How does an agent know whether it should be the one to respond?
  The first agent to receive the threshold number of messages may be called the "winner". The winning agent then starts its response by inhibiting all others: broadcast to them that the situation is already being taken care of and there is no need for further counting of agreement messages etc. Then it simply acts in the same way as an agent in a 2-agent system (stochastic scheduling can be used so that the "winning" agent is non-deterministic).

The multi-agent network becomes similar to a "lateral inhibition" neural network as described in e.g. [Aleksander and Morton, 1995] in which the strongest firing neuron (the "winner") inhibits those around it.

Responding to an anomaly is non-trivial and may require identification and suppression of hostile code. In practice, it may be unlikely that several agents are ready to respond with a precise plan of action; it may be more probable that none of them can arrive at a precise identification of the hostile component. In the same way as for the two-agent architecture in the preceding chapter, we make the simplifying assumption that a diagnosis and response is always possible (this is one of the shallowness restrictions listed in that chapter). The implementation and scenarios are designed to fit this restriction.

For clarity we use the term "inhibition message" to mean an *informative* message which simply says that the situation is already being taken care of and there is no need for further action. In contrast, "suppression" means preventing hostile code from executing and may include any means available to do this (including slowing down execution).

We assume that *only* those agents that detect an anomaly are able to take action, since the anomaly information is required to point out the hostile component (as was done in our quality monitoring implementation).
If any agent detects an anomaly it does the following:

- broadcast its decision to all other n-1 agents
- wait for the other agent's results and count the number of votes that agree with its own ("agreement" messages).
- if a threshold number of agreement messages have been received then inhibit other agents and initiate response.
- if an inhibition message has been received while there are still outstanding messages then wait for remaining messages to confirm that the inhibiting agent is supported by a majority vote.

In a 3-agent system, an example broadcast message might contain (0, 1, 0) meaning that an anomaly was detected in the second agent (B). The sending agent's identifier is given as part of the message. As stated above we assume that an agent will indicate 0 for its own status (but its field is included in the message to form a complete "fault-pattern").

If agent C detects an anomaly first and sends the broadcast, it can proceed with a response as soon as it receives one other "echoing" message (expected from agent A). The threshold is only 1 in this case. Agent C then inhibits the responses of other agents and changes the status of B to "failure". Note that the agents are not running at different speeds here (because of our assumption of a synchronous system). The existence of a "winner" is due to the sequential scheduling in the SIM_AGENT toolkit, in which agents are processed in a given order.

If any agent does not detect an anomaly but receives one or more "anomaly-detected" messages it does the following:

- broadcasts a "no anomaly-detected" message to all other n-1 agents;

- possibly increase its alertness, allowing it to revise its decision;
- It does not need to receive an inhibition message because it does not want to respond (it ignores any such messages).
- It should suppress any attempts to override vehicle control or put an agent into failure status until there is a majority vote.

9.5.4 Implementation summary

The algorithm in 9.5.3 was implemented as three additional rulesets whose pseudocode is shown in Tables 9.2, 9.3 and 9.4 respectively.

Table 9.2 shows the initiation of the voting procedure. The agent that detects an anomaly starts by broadcasting an "anomaly-detected" message to all others (RULE *start_voting1*). As soon as other agents receive this, they start waiting for further votes (RULE *start_voting2*) and broadcast their own decision (either "suspect" or "OK" covered by last two rules).

In the ruleset *wait_for_majority* in Table 9.3 the STOP after each rule means that execution of the whole ruleset stops if the rule's conditions were satisfied. Rule *failure_timeout* is the special situation where agents believed to be faulty have not broadcast any vote, thus supporting the hypothesis that they are faulty and need to be "repaired". The system responds in this situation; otherwise it cannot cope with multiple failure scenarios (to be defined later).

Rule *voting_timeout* deals with the general situation where no majority has been obtained after waiting for the maximum allowable time. Hence there is not enough "evidence" to respond to the anomaly that caused the vote to be started. The voting procedure is aborted and all records of previous votes are deleted. If the anomaly persists, a new voting procedure will be initiated. In rule *check_for_agreement*, *Threshold* = 1 in the 3 agent case. Normally it is *Majority*-1.

The ruleset *inhibit_neighbours* in Table 9.4 is a simple braodcasting of an inhibition message by the agent that switched on its response in the previous ruleset *wait_for_majority*. Note that the rulesets are not fault-tolerant. For example, an "inhibition" message is accepted without question. The algorithm is the same as the non-fault-tolerant consensus algorithm of Chapter 5, Section 5.2.6, except that an agent executing the above rulesets does not actually wait for the whole list of values. It simply waits for the threshold number of messages that agree with its own view. The rulesets do not prevent a hostile agent from sending an inhibition message with malicious intent to take over control.

Another vulnerability with our prototypes is that they are fully synchronous (see Chapter 5, Section 8.6.1). A maximally fault-tolerant solution would require an asynchronous system with a probabilistic consensus algorithm such as randomised Byzantine agreement described in Section 5.2.10. It would, however, have been impractical to implement such fault tolerance methods in the rule-based system and it would only duplicate existing work.

9.5.5 Including the voting behaviour in the self-model

Instead of applying an existing fault-tolerance algorithm, we asked the question whether the *cognitive* capabilities of the system may be extended to detect and respond to anomalies occuring during the voting process.

We approached the problem by looking at the added complexity of the meta-level code due to the voting algorithm and the need to distinguish between anomalous behaviour and correct behaviour of this additional code. To recognise such anomalies, we modified the training phase so that it includes observation of voting behaviour. Effectively this increases the reflective capability of the system.

During an interim operational phase when "mock" attacks are permitted, an agent is observed detecting and responding to an anomaly in the same way as in a normal training phase. Before the agent responds, however, it activates the same components that are active during a vote, so that the

```
define RULESET broadcast_vote
  RULE start_voting1:
  if not already responding to a problem and
    not inhibited by another agent and
    not waiting for votes and
    an anomaly was detected in any neighbour agent
  then
    broadcast vote “suspect” about the suspect agent;
    wait for votes;

  RULE start_voting2:
  if not already responding to a problem and
    not inhibited by another agent and
    not waiting for votes and
    a neighbour has voted 1 about a suspect agent
  then
    wait for votes;

  RULE general_voting_OK:
  if waiting for votes and
    there are neighbours for which NO anomaly was detected
  then
    broadcast vote “OK” about the neighbour(s);

  RULE general_voting_suspect:
  if waiting for votes and
    there are neighbours for which an anomaly WAS detected
  then
    broadcast vote “suspect” about the neighbour(s);
enddefine;
```

Table 9.2: Outline of broadcast_vote component

characteristic signature of a vote can be learned, even if it is only an “exercise”. The components include broadcasting, waiting for a majority (threshold) and sending an inhibition message. The differences between such an exercise and reality will not be eliminated but they can be minimised.

Note that this approach is not necessarily fault-tolerant, since any abuse of the voting system would have to be detectable as an anomaly. It depends on the detail of the model acquired during the training phase, and the sensitivity of the internal sensors. The cognitive approach does, however, allow an agent to reason about the anomaly and explain what is happening.

In our current prototype, the model includes voting only at a high level (indicating which voting rulesets are active). The pseudocode shows that there are many “submodes” of voting, such as “waiting for a vote”, “responding to an anomaly” and “inhibited by another agent”. The submodes can be learned during the training phase if they can be artificially activated on detection of an object-level anomaly during an interim operational phase. Although the submodes are not actually required at this point, they can be “simulated”. An alternative more efficient possibility involves parallelising the training and is outlined in Chapter 12.

```
define RULESET wait_for_majority
 RULE inhibition_received:
 if not already responding to a problem and
   waiting for votes and
   an inhibition message was received from another agent
 then
   switch off waiting for votes;
   STOP;

 RULE check_for_agreement:
 if waiting for votes and
   a vote "suspect" was broadcasted about a suspect agent and
   at least Threshold other agents have voted "suspect" about the suspect agent
 then
   switch on response this cycle;
   switch off waiting for votes;
   STOP;

 RULE failure_timeout:
 if waiting for votes and
   this agent has voted "suspect" about all neighbour agents and
   all neighbour agents have timed out on voting
 then
   switch on response this cycle;
   switch off waiting for votes;
   STOP;

 RULE majority_timeout:
 if waiting for votes and
   too much time has elapsed since the vote was initiated
 then
   forget about existing votes;
   switch off waiting for votes;
   STOP;
enddefine
```

Table 9.3: Outline of wait_for_majority component

9.5.6 Implementation-related restrictions

One of the problems with testing of a shallow architecture is that several restrictions must be made on the intrusion scenarios which seem initially to be unreasonable. Two of these restrictions are basic assumptions:

A1: *Recurrent false-positives are hostile; intermittent false-positives are accidental*: our scenarios exclude hostile code that causes the targeted meta-level to make intermittent (apparently random) false reports of anomalies. In other words we are assuming that value failures are *persistent* (Section 5.2).

```
define RULESET inhibit_neighbours
  RULE broadcast_inhibition:
  if an anomaly-response was initiated this cycle and
    no inhibition message was received from another agent
  then
    send inhibition message to all neighbour agents;
    forget voting records;
enddefine;
```

Table 9.4: Outline of inhibit_neighbours component

A2: *It is possible to distinguish between intermittent and recurrent false-positives*: This is reasonable because the current voting mechanism in the architecture can be extended to allow time for an agent's decision to "stabilise". Repeated voting to allow stabilisation is common in fault-tolerant systems. See for example [Pradhan and Banerjee, 1996]. Our architecture can also be extended so that repeated votes allow an agent to monitor different sources of data, resulting only in intermittent false positives (if its code is correct). Therefore any agent that produces a "flood" of false-positives contains hostile code.

Assumption A2 is a special case of the "shallowness" assumption mentioned earlier; that is, we are assuming that a certain kind of "deepening" of the architecture is possible (in this case vote stabilisation) without changing the fundamental properties of the architecture.

The following restrictions are essential simplifications to the intrusion scenarios and implementation.

(a): *False negatives are due to failure*: Our intrusion scenarios are designed so that their effects will always be detected as anomalous execution events provided the software is operating correctly. In other words, only a failure in the anomaly-detection software can cause a false-negative (and a failure is always due to an intrusion in our environment). This restriction is reasonable because we are not testing an individual agent's anomaly detection algorithm, but instead the interaction of components in a 3-agent architecture with voting.

(b): *Timeouts are due to failure*: Timeouts in any awaited response of an agent are caused by a component in the agent being disabled or modified. We wish to exclude other problems such as increased processing load slowing down an agent. Refering back to Chapter 5, Section 5.2.3, this is just the assumption of a fully synchronous system.

(c): *Final vote is available immediately*: The intrusions cause affected components to behave in a consistent way immediately, so that the first vote is the "final" or stabilised vote (unless there is a timeout). This is allowed by the shallowness assumption which states that a "deepening" of the voting mechanism (along with a more noisy intrusion scenario) will not make a fundamental difference to the architecture as a whole.

These restrictions, together with the more general assumptions can be summarised as follows:

Imp1: Any false-negative or timeout by an anomaly-detection component is due to failure of the component, which can either be an omission failure or destructive code (this is simply restating restrictions (a) and (b))

Imp2: Any false-positive by an anomaly-detection component is due to destructive code in the component (follows from assumptions and restriction (c))

They are labelled *Imp1* and *Imp2* because they apply only to a specific *implementation*. They will be referred to in the rest of the paper.

9.6 Hostile code scenarios

We now look at some example hostile code scenarios that were used to test the prototype. They are not exhaustive. They are examples of events allowed by the above restrictions.

One type of scenario is one in which the enemy lies to other agents about the existence of an anomaly. Otherwise it follows the normal rules of voting, such as broadcasting its detection of an anomaly, waiting until it receives a majority in its favour and sending an inhibition message before responding. The voting mechanism may contain hostile code, but only to enable false information to be transmitted to other agents. It does not cause deviation from the required voting protocol.

The following scenarios involve different types of "lying" on the meta-level:

Scenario 1(a): an agent A executing hostile code states that both B and C are anomalous, and either B or C controls the vehicle. The subsequent vote should indicate that A is wrong. Since we are assuming that only one agent is attacked, the other agents will agree that A is anomalous. **Action**: the winning agent responds to diagnose and suppress hostile code in A.

Scenario 1(b): an agent A executing hostile code is already in control of the vehicle and states that both B and C are anomalous. The majority vote indicates that A is wrong. **Action**: the winning agent will override A's control of the vehicle and begin its diagnosis and suppression of hostile code in A.

Scenario 2(a): an agent A executing hostile code states that B only is anomalous and B is in control of the vehicle. **Action:** majority vote and response as for 1(a).

Scenario 2(b): an agent A executing hostile code is already in control of the vehicle and states that B only is anomalous. **Action:** majority vote and response as for 1(b).

In scenario 1, the hostile code is indiscriminate about the agents it lies about. In scenario 2 it is selective (it can lie about several agents if $n > 3$). The two scenarios are summarised in Tables 9.5 and 9.6. We have seen scenario 2 earlier in Table 9.1. If we consider a deepened version of the

Agent	A	B	C
A	0	1	1
B	1	0	0
C	1	0	0

Table 9.5: Scenario 1 for 3 agents: A says that B and C are incorrect

architecture in which some restrictions are relaxed (for example if noise is allowed in the voting), scenario 1 may be easier than scenario 2 because the "hostile" agents are behaving in a way that contrasts most sharply with normal agents.

Agent	A	B	C
A	0	1	0
B	1	0	0
C	1	0	0

Table 9.6: Scenario 2 for 3 agents: A says that B is incorrect

9.6.1 Implementing a hostile meta-level scenario

In order to test the enhanced self-model (which includes voting behaviour on a high-level), we chose an implementation of scenarios 1 and 2 which subverts the voting mechanism itself. We implemented an example of this using a rogue *broadcast_vote* component. The pseudocode is shown in Table 9.7 and involves broadcasting an anomaly vote when there is no problem. Note that this is an example where the voting itself contains hostile code, but not the kind of hostile code that breaks the normal voting "rules".

9.6.2 Results

Appendix D, section D.3 contains excerpts of an output trace showing a run of scenario 1.

9.6.3 Hostile code scenarios for n agents

Although we only implemented a 3 agent prototype, it is important to consider what happens if $n > 3$.

For $n > 3$ agents, scenario 1 would be as follows: a number of agents $k (k > n/2)$ execute hostile code and each states that *all* remaining m agents $(m = n - k)$ are anomalous. **Action**: as for 3 agents, the majority vote should indicate that the k agents are themselves anomalous and all remaining agents are normal. The main difference is that the winning agent must suppress hostile code in k agents instead of just one. This can be seen in Table 9.8 which gives an example with $n = 5$. Diagnosis and suppression may not be more difficult, however, since the consistent behaviour may result from the same attack on different agents (with the same hostile code).

Scenario 2 is more complex: a number of agents $k (k > n/2)$ execute different hostile code and each states that different subsets of agents are anomalous. **Action**: as for scenario 1, except that a clear majority vote may be more difficult in a "scaled up" version, since there may be less contrast between the "statements" of the two groups of agents. Identifying and suppressing hostile code may also be difficult if the attacks are independent and have to be diagnosed separately.

9.7 Simultaneous attack scenarios

Returning to the 3 agent prototype, if two agents are disabled simultaneously, the remaining agent will detect an anomaly in both of them, which it broadcasts. Situations where an agent A detects an omission anomaly in B and C are as follows:

Scenario 3: A detects omission anomalies in B and C:

B and C both agree that there are no anomalies in A or they time out (they may detect anomalies in each other). This does not require a majority vote because there is no conflict. Furthermore A only detects an omission anomaly, and the required response is a repair, where the worst case is an unnecessary repair (equivalent to a false positive).

For $n > 3$ agents: a subset of agents m all detect omission anomalies in k agents where $k <= n - 1$ and $m = n - k$ because of **Imp1**. None of the k reportedly faulty agents detect

```
define RULESET rogue_broadcast_vote
  RULE start_voting1:
  if not already responding to a problem and
    not inhibited by another agent and
    not waiting for votes and
    NO anomaly was detected in any neighbour agent
  then
    broadcast vote "suspect" about ALL neighbour agents;
    wait for votes;

  RULE start_voting2:
  if not already responding to a problem and
    not inhibited by another agent and
    not waiting for votes and
    a neighbour has voted "suspect" about a suspect agent
  then
    wait for votes;

  RULE general_voting_OK:
  if waiting for votes and
    there are neighbours for which NO anomaly was detected
  then
    broadcast vote "OK" about the neighbour(s);

  RULE general_voting_suspect:
  if waiting for votes and
    there are neighbours for which an anomaly WAS detected
  then
    broadcast vote "suspect" about the neighbour(s);
enddefine;
```

Table 9.7: Hostile broadcast_vote component

Agent	A	B	C	D	E
A	0	1	1	0	1
B	1	0	0	1	0
C	1	0	0	1	0
D	0	1	1	0	1
E	1	0	0	1	0

Table 9.8: Scenario 1 for 5 agents

anomalies in any of the m agents reporting the problems. *Action*: If $m > 1$ the winning agent responds, otherwise the only intact agent responds.

Scenario 3(a): (redundant) A detects omission anomalies in B and C:

B and C both state that A is anomalous indicating that its decision about B and C cannot be trusted. Within the restrictions, B and C must be correct. If they were both wrong there are two "lying" agents, which we are excluding (Assumption *Arc2*). The majority-triggered response happens in the normal way. This variant is redundant because it is equivalent to scenario 1 above but seen from the viewpoint of the faulty agent.

Scenario 3(b): (excluded) A detects omission anomalies in B and C:

One of B or C states that A is anomalous but the other does not. This is excluded by the assumptions. Either B or C must be "lying" while A is stating they are both faulty, so *either* two agents are lying (A and one of B or C) *or* one agent is lying (the one that says that A is faulty) and the remaining one is disabled (as A has detected correctly). Assumption *Arc3* excludes this.

Table 9.9 shows scenario 3 for three agents. The "*" is a "don't care" symbol meaning that B and C may or may not detect anomalies in each other or they may time out. Scenario 3 requires that both B and C do *not* detect an anomaly in A (they may timeout). This scenario was tested successfully. Table 9.10 shows that scenario 3(a) is the same as scenario 1.

Agent	A	B	C
A	0	1	1
B	0	0	*
C	0	*	0

Table 9.9: Scenario 3 for 3 agents

Agent	A	B	C
A	0	1	1
B	1	0	0
C	1	0	0

Table 9.10: Scenario 3(a) for 3 agents

9.7.1 Implementation and results

To implement scenario 3, the *broadcast_vote* rulesets of two agents were deleted simultaneously. Appendix D, section D.4 shows trace excerpts from a run of scenario 3.

9.8 Conclusions

We have seen that a distributed reflective architecture is not limited to two agents but can be applied to a multi-agent system. In particular, the model acquisition method can be extended to any number of agents. However there are limits to the level of detail of the model (for example it does not include normal *interactions* between agents, but only the normal behaviour of a single agent. We will consider how to overcome this problem in Chapter 12.

We have also found that the simplest and most "obvious" method to overcome conflicts (voting) is very similar to the consensus problem in distributed fault-tolerance. For the purposes of rapid prototyping, we implemented a simple version of voting, which only has very limited fault-tolerance. This simple algorithm may be replaced by a fault-tolerant version such as Byzantine agreement, which guarantees that agreement can be reached about anomalous behaviour, even if the behaviour interferes with the voting process in an arbitrary fashion. (Tolerating arbitrary faults would require more agents - at least four for one hostile agent if Byzantine agreement is to be used).

In the worst kinds of hostile environment, existing fault-tolerance methods can be used to complement the cognitive science approach on which distributed reflection is based.

9.8.1 Contribution of cognitive science to fault-tolerance

To ask about possible contributions of cognitive science to fault tolerance, it is helpful to return to Chapter 5, Section 5.3.1 again. There we argued that a CRN has the following properties that are normally absent in a distributed fault-tolerant system:

1. Explicit representation of the system's internal processing, enabling reasoning, planning and explanation.
2. Observation of other agents that is not restricted to inputs and outputs. There are many possible ways in which agents can access each other's internal processing mechanisms.
3. Autonomous generation of a self/nonself boundary.

The last property can enhance the *autonomy* of a fault-tolerant system, so that human users do not have to be concerned with details of configuration management and optimisation. This is the aim of IBM's autonomic computing research project summarised in [Ganek and Corbi, 2003].

A self/nonself boundary can be defined in terms of allowed or desirable states. In Chapter 4, Section 4.2.1, we distinguished between different levels of "concern": there are "primary" concerns, which are desirable states relating directly to the primary task, and "self-protective" concerns, which are desirable internal states of the system relating to its performance and integrity. The term "internal state" is used here to include "behaviour" or "configuration".

Primary concerns are relatively easy to specify because they are just the user's concerns (e.g. keeping the vehicle out of the ditch). In contrast, it is very difficult to specify secondary or "self-protective" concerns. Therefore, it is beneficial if they can be learned. We have already seen how a model of normal ("allowed") patterns of behaviour can be acquired by mutual observation ("bootstrapping").

The system may also develop a model of "optimal" internal states that are most beneficial for integrity and performance. For example, it may learn to associate certain kinds of configuration or behaviour with a high level of component failure (where "failure" can also mean violaton of security for example). These internal states may then be assigned a lower level of "quality". This means that if the system detects these states, they would be regarded as a "quality deterioration". Conversely, states associated with the lowest level of failure or highest level of service (e.g. improved performance) may be assigned high "quality".

The requirement to make progress from lower quality states to higher quality states may also be specified as an essential service. The exact content of those states, as well as all patterns of behaviour associated with "progress", do not have to be specified in advance. They could be gradually integrated into the model of "self" during the system's "lifetime".

Diverse representations

In fault-tolerance, diversity is important to avoid multiple dependent failures in a replicated system due to a common design fault. The cognitive systems paradigm tends to support diversity because it emphasises the need for multiple and varied representations. For example, the types of entity that can be "components", "sensors" or "events" are more varied than is usually the case in fault-tolerance, which normally records only the "output" of a pre-defined component. As we have seen in Chapters 6 and 8, "components", "sensors" and "events" can be defined on different levels of abstraction (for example, a rule may be a component, as can a ruleset).

Furthermore, although each prototype uses only one type of representation (rule firing patterns or ruleset activations), there is no reason why this could not be extended to include other possibilities. For example, an agent can have a model of its own and other agents' database access patterns in addition to their ruleset activation patterns.

9.8.2 Contribution of fault tolerance to cognitive science

The three-agent prototype above may be made more robust by increasing the detail in the model of normal behaviour. In particular, the model of normal voting behaviour may be constructed so that it discriminates at a more detailed level than the present prototype. The resulting model may be used to detect a larger subset of anomalies during the voting process. However, there are two problems with this:

1. The training phase and the anomaly-detection become more complex and more likely to fail in subtle ways, as well as becoming less efficient;

2. It cannot be guaranteed that such an approach will tolerate arbitrary faults, since there is no formal proof that it will cover all possibilities (unlike for Byzantine agreement).

It seems, therefore, that a tradeoff exists between the two approaches and that fault-tolerance methods may be included in the architecture of a cognitive system, possibly at a lower level. One way to do this is to replace the simple voting algorithm above with a randomised Byzantine agreement (which is, however, considerably more complex and computationally intensive).

Decision-making and conflict resolution

The aim of a closed reflective system is to observe and criticise any part of the system (so that there is no "absolute" authority). As mentioned at the beginning in Section 1.3.1, reflection is not restricted to anomaly-detection; it may include other forms of "critique" such as recognising that a particular approach to decision-making is no longer appropriate, thus triggering a search for an alternative. Initially this wider problem does not seem to relate to fault-tolerance. However, the dependability notions of "fault", "error" and "failure" (Section 5.2) may be understood in terms of "criticism" in the general sense (i.e. not providing the right kind of "service"). For example, a "fault" may occur when the environment changes and an algorithm may become unsuitable. (Note that the environment does not need to be hostile). The recognition that the algorithm is not "suitable" for a particular problem is a kind of "error-detection" that may involve the anticipation of failure. The detection of "unsuitability" may be understood as a form of "threat" or "anticipated quality degradation".

Keeping in mind this more general understanding of "fault" we can ask whether fault-tolerance may contribute anything to the general decision-making capability of a cognitive system. We have seen in this chapter that the reasons for introducing a simplified version of voting was for the purposes of *conflict resolution*; one agent believes that a component is faulty while another believes it is correct. In the more general problem of "criticism" above, we can imagine agents in a Minsky-type architecture in disagreement about whether one of the agents is using an "appropriate" method to solve a problem. A decision must eventually be made by the system (it must stabilise) and it should be the "right" one (it should prevent failure). These requirements are specifically addressed in the fault-tolerant consensus problem (Section 5.2.6). Yet they are very similar to the problems raised by

Minsky's "student" in the hypothetical scenarion in Section 1.3. Therefore it is possible that fault-tolerant consensus and voting can be applied to the general problem of decision-making in cognitive systems.

Part III

EVALUATION AND FUTURE WORK

Chapter 10

Relevance to a Real World Computing Environment

10.1 Overview

In previous chapters we regarded the architecture as an abstract autonomous system in a simulated environment. It is a “cognition-inspired” architecture, where a collection of software components is regarded as a single cognitive system (although primitive), which must survive in a hostile environment. In this chapter we ask whether such an abstract architecture has relevance to a real-world computing environment in which current intrusion detection systems (IDS) normally work. In particular, we consider the real-world equivalent of the following:

- Self/nonself distinction,
- Closed reflection and mutual observation,
- “Cognition-directed” sensors and effectors

10.2 Self/nonself Distinction

When discussing self/nonself boundaries, we should distinguish between immune systems terminology and the terminology of computational reflection. In computational reflection the self-representation refers to the actual components in the system.

In immune systems terminology, the self/nonself distinction normally applies to patterns of behaviour and not necessarily to specific components within the system that can be modified or suppressed. For example, a component may be physically part of the system but its behaviour may be classed as “nonself”.

Since our architecture requires automated diagnosis and self-repair, the model of “self” is part of a larger *self-representation*, which includes a modifiable map of components so that the execution of the real components can be suppressed or modified. If a new component is added to the agent, it will automatically appear in the map and be accessible to the agent. The SIM_AGENT toolkit provides this feature by holding all executable rulesets within the agent’s working memory (database). As argued in Chapter 4, Section 4.2.5 this capability satisfies the requirements of a causally connected self-representation (CCSR) in a computationally reflective system. It thus goes beyond the notion of “self” and “nonself” in artificial immune systems. If the self-representation only contains behaviour patterns and quality-related state descriptions then it does not provide the required causal connection.

In the following we use the term “IDS” (intrusion detection system) to mean any software (or hardware) that detects intrusion or misuse and takes action as necessary. Roughly, it is part of the “meta-level” of Figure 4.1.

10.2.1 Application-oriented boundaries

As mentioned in Chapter 1, Section 1.2.3, the self-model of the system may contain statements defining what is "acceptable" or what constitutes minimal *requirements* in terms of internal or external states and actions. We assume that the details of the self-model are mostly learned during a training phase. (We argued in Chapter 4, 4.2.1 that autonomous acquisition is more robust).

In most computing environments, however, the high-level requirements relating to the primary task are specified in advance. For example, in a university teaching network, intrusion and misuse detection can have different objectives depending on the policy of the institution. The objective of one institution (A) might be to ensure that students have freedom and privacy while using e-mail and web services, with no restriction on the kind of tasks they do. For example, they may wish to experiment with novel programming methods. An "intrusion" would then be any attempt to disrupt these freedoms, such as unauthorised reading of e-mail. "Pattern" anomalies in the form of unusual execution patterns or data access patterns must be tolerated so long as they are not associated with disruption or privacy violation or other problems (such as money being stolen).

In a different institution (B), the intrusion/misuse detection may have the objective of ensuring that students only use the computing facilities for specific programming assignments, with the web being used for literature searches only. In this case, any other use of the facilities is regarded as misuse.

In both cases programs that cause "acceptable" state changes are "self"; those that cause unacceptable state changes (such as unauthorised access to the root password) are "nonself". The content of "self" is more tolerant in the first case than in the second.

From these examples, we can say that a self/nonself boundary can be defined on two levels:

1. *Protection boundary*: components (hardware and software) that should be protected as different from other things; in cases A and B above this includes all legitimate application programs and possibly the operating system and hardware.

2. *Requirements boundary*: acceptable or required behaviour of components in (1) as different from unacceptable or faulty behaviour; this boundary is different for each of the institutions above. This might also be called a "concern" boundary (Section 4.2.1). The requirements may state that the behaviour must be "normal".

In biological systems, distinction 1 is similar to the one made by the nervous system of an animal (it does not treat itself like any other object in the environment). A protection boundary may change with time if coalitions can be formed or disbanded dynamically. See for example [Byrd et al., 2001] on dynamic coalitions.

Distinction 2 is similar to the one made by the immune system and higher level cognitive reflection (meta-management and volition - for example identifying ourselves with values and principles that we agree with; dissociating ourselves from things we don't like). An intrusion or fault happens if there is a conflict between the two (nonself type 2 has "invaded" self of type 1).

In distributed reflection, each agent has a different (partial) view of boundary 1. The agents may also have different representations of what is required (boundary 2) but they should not lead to conflicting results. Conflict-resolution in quality evaluation is discussed in Chapter 12.

Boundary 1 may define the components an agent has "authority" to access and modify and what it has "responsibility" to protect. Chapter 11 will discuss the issue of authority and responsibility in distributed reflection.

10.2.2 Including the architecture in the self-model

In our prototypes, boundary 1 is somewhat arbitrary. If a new component is added to an agent, it is also added to the agent's component map without question (similarly with removal). It is only when it is executed that a new component will trigger an anomaly (or in the case of a missing component, it fails to execute).

The system may be extended to include a model of the *architecture* where components are *required* to exist in certain slots. The architecture that carries out the primary task is the object-level architecture. For example, in the Treasure scenario, an agent's object-level must include sensing, deciding and acting components. In the student example above, a C programming environment requires an editor and C compiler. If these components are suddenly not available for execution (i.e. not available in the component "map"), there is a problem.

Note that institution A above has a much wider range of "legitimate" applications to protect, while institution B may define a restricted programming architecture that comprises only a subset of all applications running in larger environment. For example, the same physical cluster may be used to support multiple applications, only one of which is a programming course. In B, any new components may be regarded as "external" and treated indifferently. In A they would be regarded as anomalous patterns, but they might be incorporated into the self-model as well as into the component map if they are not associated with any quality degradation. The difference between application domains and physical infrastructure is also emphasised in the information security methodology specified in [Robinson, 2001].

10.2.3 Dependability concepts and self/nonself boundary

As defined in Chapter 1, Section 1.2.4, the recognition of "nonself" triggers a self-protective response and "rejects" the nonself behaviour. Therefore, not only is a distinction made between two classes of behaviour, but the acceptable behaviour ("self") is *re-asserted*.

Returning to the requirements for self/nonself distinction in Section 3.4, the components that can recognise and respond to "nonself" have the following properties.

1. they have to be within the protection boundary (otherwise something other than the system is making the distinction)

2. their behaviour has to be within the requirements boundary (they should be "correct")

To understand its contribution to real world systems it is also important to relate the concept of a "self/nonself" boundary to fault-tolerance. In particular we look at the dependability concepts in [Laprie, 1995] (Section 5.2). In terms of these concepts, the requirements boundary appears to be most closely defined by the difference between "failure" and "non-failure". Thus "nonself" means any behaviour in which "delivered service deviates from requirements". In our prototypes, however, the situation is more complex than this because detection of "nonself" involved more than simply the detection of a failure that has already occurred.

To investigate this further we return to the Treasure scenario and ask what can go wrong using the concepts of "fault", "error" and "failure". For example, in the first prototype, a disabled component "fails" because it "delivers a service that deviates from its required function". A failed component becomes a "fault" in the larger system controlling the vehicle. The following is one way of defining the subsequent chain of events:

- Component fault (for prototypes 1 and 2 respectively):

    1. randomly selected component (a rule or a ruleset) is deleted
    2. a component is modified by an enemy

- Component error (for both protototyes 1 and 2): required code is executed wrongly or not executed

- Component failure for prototypes 1 and 2 respectively:
    1. omission failure: component does not deliver the required service
    2. value failure: component delivers incorrect service
- System fault now exists: caused by failed component(s)
- System error: components that depend on the failed components are not executed correctly; quality deterioration in prototype 2
- System failure: a critical variable becomes 0 (e.g. energy) or the vehicle is in the ditch, meaning the system no longer provides the required service.

Correspondences between actions of our prototypes and dependability concepts are shown in Table 10.1.

System action	Dependability term
Anomaly detection	Error detection
Threat detection	Error detection
Identify faulty component	Fault diagnosis
Repair faulty component	Reconfiguration
Evaluate damage	Error handling
Return to trusted state	Error handling

Table 10.1: Correspondence between system actions and dependability

From this table, the detection of “nonself” is best defined as error detection. However, the dependability concept of “error” does not make a distinction between a pattern anomaly and a quality degradation (which we have seen is an important distinction). There is a need to detect *worsening* quality of service *before* the service deteriorates into a “failure”. It is also important to reason about the system state to determine if there is an impending quality deterioration (Section 8.3.4 and 8.6). In other words, error detection is enhanced by *anticipation* of possible failure (or of quality deterioration that can lead to failure).

In summary, we can say that the cognition-inspired idea of a self/nonself distinction can be applied to real-world IDS and existing fault-tolerance concepts.

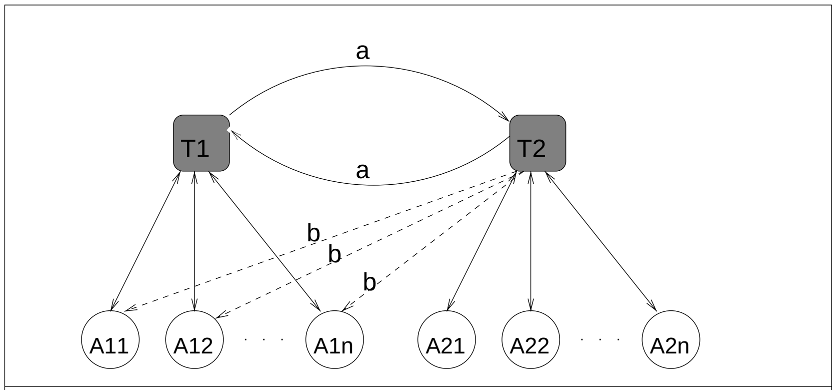
10.3 Closed reflection and mutual observation

Mutual observation is a necessary feature of a closed reflective network (CRN) and should be applicable generally, provided that processes have permission to observe each other in different states. In a real world application, the IDS is the meta-level of Figure 4.1. To show how the distribution can happen, we assume that an IDS is in one of the following configurations:

1. The IDS normally runs as a single process.
2. The IDS is a collection of processes or agents, each of which has a specialist responsibility for a particular domain in which intrusions can be detected (as in e.g. AAFID [Balasubramaniyan et al., 1998] and [Spafford and Zamboni, 2000b]).

In the first case, several instances of the IDS may observe each other’s behaviour patterns when under various kinds of attack, thus allowing them to build models of normal IDS behaviour when it is responding to an attack.

In the second case, the simplest possibility is to treat the whole collection of processes as a single instance of the IDS and generate multiple instances. An agent (or a group of agents) in an IDS instance has the task of acquiring a model of the IDS itself. As an example, a hypothetical modification of a



Hypothetical modfication of an AAFID system running on a single host:.
A "transceiver" T1 manages anomaly−detection agents A11, A12, ...
Transceiver T2 is a replicated version of T1 with agents A21, A22, ...
Arrow type a = mutual observation between transceivers T1 and T2
Arrow type b = independent observation of another transceiver's agents

Figure 10.1: Applying distributed reflection to AAFID

simple AAFID architecture is shown in Figure 10.1. This is an example where the AAFID system runs on a single host. Hence the highest level is a "transceiver" labelled T1. (Still higher level entities called monitors are used for remote monitoring and control but they are not included here; see 5.3.4 for a fuller description of AAFID). In the diagram the whole AAFID architecture is duplicated. The simplest way to apply distributed reflection is to enable the transceivers to observe each other as shown in arrow type "a". The transceivers might then build models of each other's execution patterns. Some of the model building may be delegated to a transceiver's "agents" (the circles marked A_{ij}).

Assuming that the transceivers can monitor their agent's execution patterns as well as control them in other ways, the resulting configuration is comparable with Figure 4.1(b) in Chapter 4 with the difference that the meta-level in the modified AAFID is now more like a "meta-(meta-level)" since we are calling the whole IDS a meta-level. Hence a transceiver's agents are like an "object-(meta-level)".

It is not necessary to apply the design in Figure 4.1(b) directly to an existing distributed IDS. Instead the IDS architecture itself may be modified internally to incorporate varying configurations of distributed reflection. An additional (or alternative) modification of AAFID in Figure 10.1 is shown using arrows of type "b" which enable a transceiver to independently observe the individual "agents" of transceiver T1. For space reasons the equivalent "b" type arrows for T1 to T2's agents are not shown.

The main question is whether the equivalent of "internal sensors" are supported by the IDS system. We discuss this in the next section.

10.4 Sensors and Effectors

The internal sensors and effectors are the components that allow an agent to be a reflective control system as defined in Chapter 4, Section 4.2 and in Figure 4.1. If the architecture is cognition-inspired, these components are coupled with the system's high-level reasoning and can be retasked accordingly. Most of the following discussion will focus on sensors, since this is an active research area in current intrusion detection systems (e.g. [Spafford and Zamboni, 2000a], [Zamboni, 2001], [Vigna et al., 2001]).

The application software that plays the role of the object-level only has *external* sensors and effectors. For example, they may be in the form of a user interface, with sensitivity to mouse clicks and keyboard inputs (external sensors) and displays (external effectors).

An IDS (along with supporting software such as firewalls) plays the role of the meta-level and therefore, its sensors and effectors can be regarded as "internal", since their actions are not "user-visible". They are usually not called "internal" however. They are classed approximately as follows:

- Embedded:
  sense code execution events [Spafford and Zamboni, 2000a], [Zamboni, 2001]

- Network-based:
  sense network activity (e.g. Snort, which is documented in [Dries, 2001])

- Host-based:
  sense activity on a host, such as modification of critical data (e.g. Tripwire [Kim and Spafford, 1993]),

- Application-based:
  sense application activity, e.g. transactions [Almgren and Lindqvist, 2001]

The internal sensors used in our implementations are very similar to the "embedded" sensors defined by [Spafford and Zamboni, 2000a] as follows:

> Internal sensor - a piece of software (potentially aided by a hardware component) that monitors a specific variable, activity or condition of a program and that is an integral part of the program being monitored.

In our prototype, sensors were added to the POPRULEBASE interpreter (an implementation level of the agent being monitored), which allows a monitoring agent to access rule execution events of the agent being monitored. Such sensors have been shown to work successfully in real-world intrusion detection by Spafford and Zamboni.

Zamboni [Zamboni, 2001] also introduces the idea of a more "intelligent" internal sensor called an *embedded detector* that can itself have some anomaly-detection capability:

> Embedded detector: An internal sensor that looks for specific attacks and reports their occurrence.

In the following discussion we use the term "internal sensor" to include embedded detectors (which may be in the form of mobile agents for example).

In our cognition-inspired architecture, however, there are two levels of internal sensors: the low-level sensors embedded in an implementation level of the agent being monitored *produce* the trace because they record what they have seen. The high-level sensors of the monitoring agent's control system (which may be the agent being monitored) reads (and uses) the trace for anomaly-detection.

In the remainder of this discussion, we consider our approach to internal sensors in the context of current research problems, which include the following:

- too many false positives - there is a need for knowledge about what is "bad" and correlation of sensor alerts to reduce noise.

- large amounts of data - selectivity is necessary as well as dynamic allocation of sensors in response to an alert ("attention focus")

10.4.1 Coordination between sensors and high-level reasoning

Uncoupled low-level sensors can produce inaccuracy due to a partial view of the system. This problem has been analysed by [Ptacek and Newsham, 1998]. For example a network packet sniffer has no knowledge of what is done with a packet when it is received.

High-level internal (and external) sensors in our architecture are controlled directly by the autonomous system in the way that a cognitive system directs its sensors. It would be more effective if computationally reflective features allow access to an agent's own (and perhaps another agent's) low-level embedded sensors within their implementation levels. For example, results from a packet sniffer could be correlated with those of higher level sensors. Each type of sensor could be retasked accordingly, including the packet sniffer.

The second version of the prototype involved quality evaluation of software performance and recovery from hostile code. Due to the increased complexity, the low-level embedded sensors called by the rule interpreter were adjusted to monitor the data at a less detailed level than the first one (the code implementing the sensors is user-definable in POP-11). This was necessary to reduce computational overhead. These low-level sensors could also have been adjusted autonomously by the agent's own high-level reasoning components in response to an anomaly that was not immediately diagnosable. This was not implemented because it would require features (such as computationally reflective operators) not available in the SIM_AGENT package, but it is possible in principle.

Related work in intrusion detection is *alert correlation* by a higher level alert management system (e.g. [Valdes and Skinner, 2001]).

10.4.2 Attention focusing

In our prototypes, an agent's high-level sensors assume that much of the low-level processing has already been done, keeping the amount of data manageably small. This is not the case in real-world intrusion detection where sensors have to process very large amounts of data. Most of this data may be irrelevant to the diagnosis of a particular problem. It is therefore important to adjust dynamically the type of data being monitored. A related problem in agent societies is the *monitoring selectivity* problem [Kaminka, 2000].

A scaled up version of our prototypes will also have this problem. For example, the data available to an agent might be increased by including data access traces in addition to rule-firing traces (for example, the number of reads or modifications of a selected data object within a designated time period). However, due to the strong coupling between an agent's high-level reasoning and its sensors, an agent can "focus attention" on the most relevant data according to the results of its reasoning. For example, it may be possible to select certain data objects for particular attention or try to obtain a more detailed trace of a suspect software component, while reducing the data obtained from trusted components. This may involve retasking of low-level sensors. For example, a component being monitored (which may be part of another agent) could have more of its trace-production "switched on" by the monitoring agent, assuming that it has authority to do this.

Recent work in intrusion detection includes [Gopalakrishna and Spafford, 2001], which is an enhancement of the AAFID system and allows an agent to have a specialised "interest" in certain kinds of data. An agent's interest can be dynamically adjusted in response to an alert. Another possibility is to use mobile agents as intelligent sensors [Jansen et al., 2000].

10.4.3 Knowledge of the application domain

Quality evaluation in our prototypes assumes that the intrusion detection component has knowledge of the application domain, and in particular it should have a representation of what is important, desirable or undesirable in that application domain. This can be simple initially, for example it may just be a symbolic representation of what situations are unacceptable, such as when an authorised third party deletes or modifies a private data file. Desirable and undesirable states can also be learned by observation.

Knowledge of the task domain has advantages because it may enable a focusing of monitoring resources towards the critical data that are most likely to be illegally accessed and those actions that have the most consequences within the application task. For example, in medical diagnosis, some data are particularly sensitive while others may only involve administration-related records whose modification would only have minor consequences.

Recent work on intrusion detection applies knowledge of network topology instead of the application domain. For example, [Mittal and Vigna, 2002] uses knowledge of sensor positions in a network to detect likely malicious activity in those positions. [Vigna et al., 2002] addresses knowledge representation of network topology and network entities.

10.4.4 Independent monitoring

The low-level (trace-producing) sensors in our prototype have several advantages of Spafford and Zamboni's internal sensors, namely: they are not separately running processes but are part of the program being monitored (thus reducing CPU overhead), they are more difficult to disable without disabling the program they belong to, and they perform direct monitoring without the need to access external log files (which can increase possibilities for intrusion or error).

The embedding of sensors within the program being monitored can also have disadvantages, however. Our prototype assumes that any process P leaves a trace that can be observed by a monitoring process M and the trace *cannot* be inhibited or modified by P itself. This is one requirement for independence in monitoring (see Chapter 2, Section 2.3.6): a process has no control over the trace that it produces. However, in the prototype software this is only true *within the simulation environment*. Implementation levels such as the rule interpreter are excluded from the architecture in that agents do not have access to such details (they are not "computationally reflective"). If an agent did have access to its implementation, it might suppress its own trace-production, or modify it. (Spafford and Zamboni's sensors also have this limitation). If an agent prevents its own sensors from logging what they have detected, other agents cannot use them for monitoring.

This may seem like an argument against computational reflection in this case. But in a scaled-up version, we may wish to include computationally reflective operators to increase the flexibility of self-monitoring and response. The critical issue is not whether to include these reflective features but rather what kind of modification or observation is an agent "allowed" to do (or "obliged" to do in some circumstances). The specification of permission and obligation is a challenging problem which we will discuss later in Section 11.5.

The distributed nature of the architecture is also essential for independent monitoring. If a single agent monitors itself, there is a contradiction between the requirement for strong coupling between reasoning and sensing on the one hand and independent monitoring on the other: if the monitoring can be controlled (or inhibited) by the agent which is being monitored then it is not being independently monitored. (Note that "distributed" reflection does not necessarily mean "closed" reflection as defined in chapter 3).

10.4.5 Granularity of anomaly-detection

In our architecture the objects being monitored are "components" on various levels (such as a rule-set or a rule). The later prototype versions (those concerned with hostile code) only detect that an expected component is not active or an unfamiliar one has become active (in that it has an unfamiliar name). We assume that this can be deepened so that an "unfamiliar" component is one with an anomalous *execution trace* instead of just an anomalous name.

A realistic IDS may not always require a fine-grained monitoring resolution, however. IDSs may be classified according to the data sources they use for building a model. Some IDSs observe network activity (e.g. using packet sniffers). Others build models from user profiles (e.g. [Fawcett and Provost, 1997] for fraud detection in mobile phone records).

Our anomaly-detection is most similar to that used by existing IDSs that monitor program execution patterns. In such systems it is common to focus on system calls made by a program, since system calls are normally the most “dangerous” execution events and most intrusions will probably make use of them. Traces of system calls can be produced by the Linux *strace* command or the Solaris Basic Security Module).

In addition to Forrest’s work in [Forrest et al., 1996] mentioned earlier, [Ghosh et al., 1999a] use system-call traces for anomaly- and misuse detection. This level of granularity is not much more detailed than ours, since only the “significant” events (system calls) are being recorded. The main area of “deepening” is the localising of anomalous sequences. Ghosh’s main point is that it is not good to simply compare a whole execution trace of a program with its expected trace. Anomalies may be missed this way because they might be “averaged out” over the whole trace. Instead “temporal locality” of intrusions should be exploited by using a hierarchy of fixed sized intervals, where the elements of a higher level interval are whole intervals of the lower level. Level 1 (the highest) is a session, level 2 is a program execution within a session, level 3 is a fixed size window in a program trace, and level 4 is a trace entry (sequence of system calls in a yet lower time frame).

For every system-call sequence in a trace interval that is not found in the database, an “anomaly counter” is incremented. An anomaly exists if the counter exceeds a threshold on all interval levels in the hierarchy. For example, a session is anomalous if a sufficient number of its programs are anomalous, a program is anomalous if a number of its trace intervals are anomalous and so on. It would not be difficult to include this kind of hierarchy in our architecture.

10.4.6 Internal effectors

Internal effectors are used for automated response to an intrusion. The equivalent of internal effectors in current intrusion detection software is concerned mostly with stopping, slowing down or isolating an untrusted process. Examples include [Sekar et al., 1999] which isolates an untrusted process by ensuring that its actions do not interact with other components. In [Somayaji, 2000], a software component’s execution is slowed down by an amount proportional to its deviation from “normality”. As already mentioned in Chapter 8, this approach is purely numerical and does not include high-level reasoning, although it may be effective in some applications.

10.5 Summary and Conclusions

This chapter has shown that it should be possible to adapt our particular cognitive system approach to a real-world computing environment and to make it inter-operate with existing IDS algorithms/architectures (e.g. AAFID). Particular conclusions are as follows:

- The two kinds of “self/nonself” boundary are not just philosophical ideas; they have equivalents in real world systems.
- Detection of “nonself” is similar to “error detection” in dependability terminology, but cognitive science uses the additional notion of *anticipation* of failure.
- Mutual observation between agents may be applied to an existing multi-agent IDS such as AAFID.
- The concept of “internal sensors” and “effectors” in our architecture have equivalents in existing IDS systems. The cognitive system approach is very relevant to current IDS research problems such as the need to focus attention selectively and for high level reasoning to be more closely coupled with sensors.

In the next chapter we will look in more detail at the “mapping” of the cognitive architecture onto a real world system.

Chapter 11

Mapping the Architecture onto the Real World

11.1 Overview

The last chapter investigated whether essential features of a distributed reflective architecture have relevance to real world intrusion detection. In particular, how does the cognitive system approach fit in?

This chapter considers how the abstract architecture itself might be deepened so that it can be "mapped" onto real world concerns. We address the following questions:

- How (and why) should we increase the number of agents and how do we allocate responsibilities to them? Is it best to have multiple ways of performing the same task (replication) or to divide the main task into subtasks and allocate them to specialist agents?
- What are the relationships between agents in a deepened version? Which agents have "authority" over others? What "obligations" do agents have?
- How should the meta-level and object-level of the prototype be mapped onto real world concerns?

11.2 Evaluation of Software Design Methodology

As defined in Chapter 4, Section 4.3 the "broad-and-shallow" methodology is a way of translating philosophical concepts into software engineering constructs and then integrating these constructs into a "complete" cognitive system. When we embed this cognitive system into an environment with a well-chosen scenario, we should discover new problems that trigger new kinds of design that might otherwise never be considered.

The scenarios cannot be exhaustive because the space of possibilities is too large. Instead they are well chosen examples that can draw attention to problems that were overlooked and cause the design (and even the requirements) to evolve in an unexpected direction. The methodology complements formal methods, which aim to prove that existing concepts and designs satisfy existing requirements (not to evolve designs and requirements).

Our use of the methodology has drawn attention to the following concepts and design principles, which were not expected to be important:

- From Chapter 6 (see conclusions 6.9): "temporal diversity" and the importance of different meta-level roles for the same component

- From Chapter 7: (see conclusions 7.5): importance of context sensitivity and autonomous generation of contexts.
- The notion of an "agent" as a replaceable component.

If we consider the last point, the methodology has enabled the exploration of a new region of the design space, which is the space of possibilities between single agent designs and "teams" of agents (Section 5.3.3).

Both single agent and multi-agent paradigms are based on the metaphor of an agent as a sort of "individual" (e.g. insects, robots, humans). For example, in a traditional AI-based agent, circularity is seen as a problem (it is illogical etc.) because the system should be capable of consistent reasoning and action. Therefore "circular" designs tend to be excluded. On the other hand, multi-agent systems tend to be based on the "society" metaphor, meaning that it is not normal for agents to have access to each other's internal processing.

In contrast, the exploration of design and niche space is not constrained by these metaphors and can lead to innovative designs. As regards the role of an "agent" the following differences became apparent during development:

- The boundary between one "agent" and another is somewhat artificial, and exists largely for design convenience. The boundary is often blurred in the system's own distinctions, for example when building a model of a meta-level (Section 6.6.5).
- The boundary between self and nonself (Section 10.2) is the distinction that the system is most "concerned" about (using the definition of "concern" in Section 4.2.1).

A significant limitation of this methodology is the reliance on the "deepening assumption". In subsequent sections we will consider ways in which the architecture can be deepened.

11.3 Architectural Deepening

We now consider a simple scaling up of the current prototype and its simulated environment. Ways of increasing the complexity include the following:

- External world is more varied and requires planning to avoid dangerous situations
- Events detected by sensors (external and internal) are noisy. They do not start and stop instantly (as in the current scenario) but may instead vary continuously
- The ways in which an attack can affect components and the interactions between them are more complex and subtle.

In a simple scaling up of the simulation, the "broad-and-shallow" architecture will have to be deepened. Deepening will normally increase the number of agents. When considering this, it is important to distinguish between "top-level" control agents and "service" agents such as the repair agents in the prototype. A service agent has a subordinate role in that its services are requested by a control agent but it does not itself request services from any agent it serves. It is semi-autonomous (see discussion below). A *specialist* is any agent that is assigned a *subtask* and has no global overview of the main task. Some options for assigning roles to agents are as follows:

1. Replicated control agents:
   Control agents are identical versions of the same software where one agent is the primary and has one or more "backups". Any specialist agents are subordinate. The current prototype has this configuration.

2. Replicated control agents with design diversity:
   Control agents satisfy the same set of requirements but have different designs. For an example of diversity see [Xu et al., 1995]. In particular, the agent's definitions of good or bad quality are in agreement but their software design and internal representations are different. Any specialist agents are subordinate.

3. Specialist control agents: Each agent is "responsible for" a particular subset of requirements or type of work. The agents' definitions of good and bad quality may not be in agreement (see next chapter).

In this chapter we consider replicated control agents only, where specialist agents have a subordinate role and the number of control agents is small (e.g. 3 or 5). This is the simplest form of deepening because it is most similar to a conventional software architecture with the additional feature of being replicated into a few mutually protective versions (or variants).

At this point, we consider primarily the deepening of the architecture on the meta-level only; that is, the self-monitoring, intrusion detection and recovery components. Deepening of the object-level is best analysed in a hypothetical non-simulated application domain (which we introduce later).

11.3.1 Specialist service agents

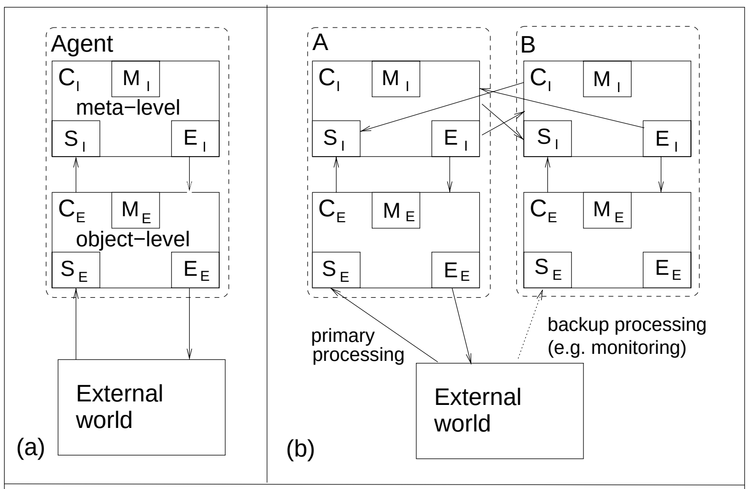
In the deepened architecture we talk about a meta-level and object-level of a *control-system* which has a single control agent along with multiple service agents. Such a control system corresponds to a "participating agent" in the prototype and is replicated into N variants (or versions) as was shown in Figure 4.1(b), which is reproduced in Figure 11.1(b) for convenience.

In general, service agents belong to the meta-level if they are supporting intrusion detection or response; they belong to the object-level if they are specialists in the application domain (which we will say more about later). If we return to Figure 11.1 we can first consider the deepening of the architecture of a single control agent as in Figure 11.1(a). A deepened version of Figure 11.1(a) is shown in Figure 11.2. Here the meta- and object-levels of a single control agent are "stretched" to include service agents. Instead of talking about a single "control agent" we now use the term "control agent cluster" or simply "cluster" to denote a control agent, along with all its service agents on the meta- and object-levels. The ith meta-level service agent of a cluster c is labelled $S(m, i, c)$ where m indicates "meta" and the ith object-level service-agent of c is labelled $S(a, i, c)$ where a means "application". (Using "o" for "object-level" would be confusing). In the diagram, the broken line arrows indicate internal and external sensors and effectors as in the original Figure 11.1(a). However, they may not follow the direct path shown. For example, some types of sensing may be delegated to mobile agents. Similarly, there is no requirement for the control agent on the meta- and object-levels to be a single agent (which in our terminology is always controlled by a sequential process as explained in chapter 3), but this is the simplest way of deepening the architecture in Figure 11.1(a).

The solid arrows in Figure 11.2 indicate "management" relations, meaning that a control agent manages the activities of a service agent but not vice versa. This will be explained in the next section in more detail.

In the distributed form of the deepened architecture the meta-levels of clusters mutually observe each other. If the current prototype is deepened in this way, the execution behaviour of specialist agents will appear as part of the behaviour of the meta-level of the calling agent's cluster. (This is currently the case for repair agents, with the difference that the repair agent is treated as part of the calling agent).

For example, if a control agent C1 (belonging to cluster 1) requests a service from a specialist diagnosis algorithm D1 then D1's behaviour will be regarded as part of C1's diagnosis behaviour in response to an anomaly that it has detected (possibly with the help of another specialist agent). This diagnosis behaviour will in turn be monitored by the neighbouring cluster C2 using an independent



C_I	control of internal world	C_E	control of external world
M_I	model of internal world	M_E	model of external world
S_I	internal sensors	S_E	external sensors
E_I	internal effectors	E_E	external effectors

(a) Single−agent reflection; (b) Distributed reflection (simplest case): agent is divided into primary and backup copies A and B. Only the primary changes the external world; both have read/write access to each other's meta−levels and their own object−levels

Figure 11.1: Reflective agent architectures (reproduced)

set of services provided by different agents. Figure 11.3 shows a CRN (closed reflective network) of this kind with 3 clusters (as mentioned in Chapter 3, "closed" is a special case of "distributed"). The thick dashed lines are mutual protection relations between meta-levels. Thick dotted lines are interactions with the external world. As with the prototype, only one cluster is in control of the external world at a time, but all clusters can sense it. In contrast to Figure 11.2, the top-level control agent in each cluster is divided up into two separate agents (an meta-level and an object-level control agent). This configuration is more realistic and has some similarity with the AAFID architecture described in Chapters 5 and 10. In the diagram, each cluster has its own dedicated service agents, thus making monitoring easier. For example, the service agent providing the diagnosis algorithm D1 may be uniquely identified as $S(m, 1, D1)$. (The arguments for cluster and agent may be any identifiers, not necessarily an index).

11.4 "Deepening" the Failure Assumptions

A claim of the thesis is that hostile environments exist in which a closed reflective system can survive (Section 1.3) and where "hostile" means that *any* meta-level components may be damaged or modified. The implementations have shown this to be the case. We are not claiming that distributed reflection on its own is sufficient to tolerate *all* possible hostile environments. However, if the system is to be integrated with real world systems, it is important to consider the "deepening" of the environmental assumptions (i.e. failure assumptions). Possible forms of deepening are as follows:

1. Allow for *dependent* failures. For example, components may fail simultaneously or in rapid succession due to the same design flaw (we assumed that failures are independent);

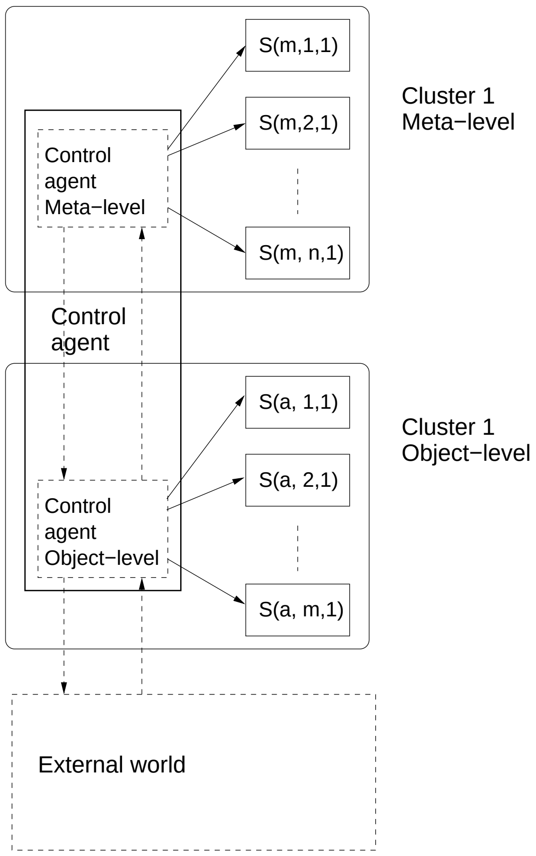


Figure 11.2: Reflective agent with two layers of service agents

2. Introduce intermittent faults. For example, a component only occasionally delivers a wrong result but at other times it gives the correct result, even though the circumstances are the same (we assumed that once a component is modified, it always produces the wrong result in the same circumstances; this also excludes some Byzantine failures)

3. Introduce communication channel faults (we assumed that messages are always delivered correctly, with no timing or omission failures)

4. Introduce faults whose effects (errors) cannot be detected by the internal sensors or do not show up as a deviation of normal behaviour (e.g. because the model is not refined enough, or was acquired by observing the wrong sort of behaviour) This would include faults in the low-level software on which the SIM_AGENT system is running, since this was not subject to observation by internal sensors.

5. Introduce faults that happen during recovery from a previous fault (which may be dependent on the first one) and that disrupt the recovery process. For example, the voting process in Chapter 9 may be subverted by a subsequent fault. So the recovery process itself should be fault-tolerant.

The first kind of problem may be minimised by introducing design diversity [Laprie et al., 1995]. Questions to be asked include the following:

- at what level of detail is the system replicated and diversified? In Figure 11.3, this might be at the level of the control agents, or at the level of the service agents (e.g. different diagnosis algorithms)

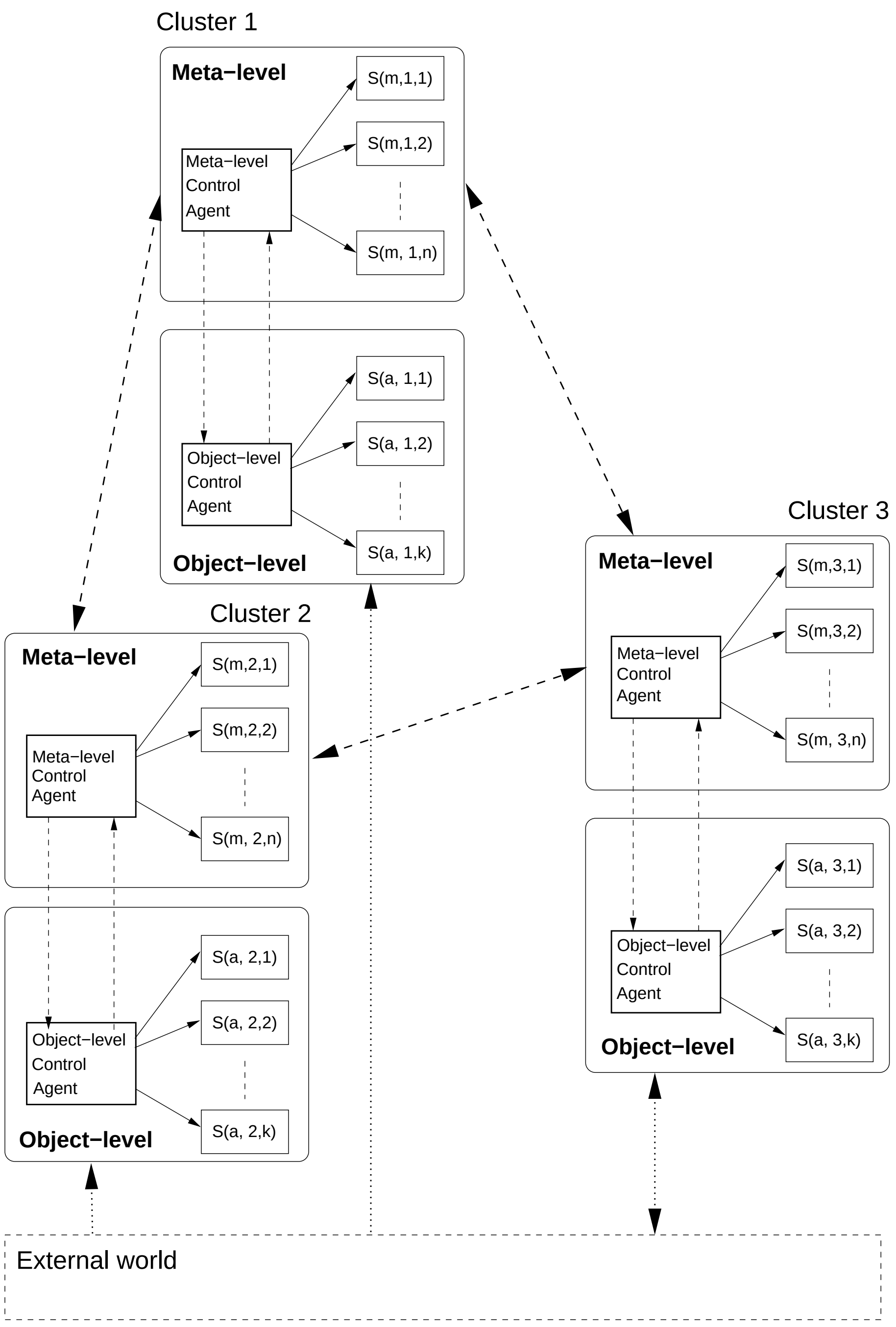


Figure 11.3: Closed reflective network with clusters

- Which layers are to be diversified? Figure 11.3 would normally be part of the application layer. It does not include lower layers such as the operating system and general utilities.

Introducing diversity on the agent level is discussed in Chapter 12 as a future area of research.

To address the remaining classes of fault, it may be advantageous to integrate the high-level multi-agent system with a fault-tolerance layer between the application layer and the operating system. This is the idea of the "Middleware" of the MAFTIA project [Verissimo and Neves, 2001]. This leads to the following questions:

- Do we want to extend the reflection to cover the middleware and operating system, so that the agents can reason about these levels and possibly modify them?
- At what level of abstraction do we want to represent the components on the lower level, if we want to reason about them?

An assumption of the closed reflection concept is that the components with the most "authority" in the whole system are at the highest levels in the application layer (in particular the meta-levels). This makes them "transparent" and easy to monitor. However, in most real world systems this assumption is probably not true, since the operating system is the most "privileged" layer. It may therefore be advantageous to build a model of the most privileged components of the operating system and reason about their behaviour. This may also apply to any fault-tolerant middleware. The main objective is that reflection and fault-tolerance should complement each other as envisaged in Chapter 9, Section 9.8.2. Determing how best to integrate a fault-tolerance layer with the kind of architecture in Figure 11.3 is an area of future work.

11.5 Authority and Responsibility

It is important to consider relations of permission and obligation among agents on the application level only (also called "authority" and "responsibility" respectively).

In Chapter 4 we saw that a closed reflective network (CRN) is one in which all participating agents have permission (and in some cases an obligation) to interrupt or redirect the processing of all other agents. This means that there is no agent that is not observable or modifiable from within the network. The situations where an agent can intervene in the execution of another were limited and clearly defined, however. Agents are not permitted to arbitrarily interrupt or modify each other, but they have an obligation to do so whenever the integrity of the system is threatened (whether there is a threat situation or not may be determined by a vote).

In a two-agent CRN, an agent detecting a problem in its neighbour's meta-level has an obligation (and thus also permission) to take action to restore it to its correct state. In an n-agent CRN ($n > 2$), each one has an obligation to wait for a majority vote before proceeding with a response; the "winning" agent has permission to respond and inhibit the others, while the remaining agents have an obligation to allow the winning agent to respond.

In our prototype, the permissions and obligations are defined implicitly in the anomaly detection and response rules. They enable a control agent to determine whether the behaviour of an observed meta-level is acceptable or unacceptable. For example, if a meta-level appears to be thwarting the voting rules, this is unacceptable. (In the general case, this decision should be called "quality evaluation", although in the prototype it was called "pattern anomaly-detection", which is the simplest case of quality evaluation as explained in Chapter 1, section 1.2.3.

We can define anomalous behaviour in terms of permission and obligation as follows:

- If an agent A has permission to carry out an action X and it is observed to do X then the action is not regarded as anomalous.
- If A does *not* have permission to do X and it is observed to do X then its action *is* anomalous (for example it does not have permission to respond to an anomaly until a vote is completed).
- If A has an obligation to do X and it is observed *not* to do X then its lack of action is anomalous. (For example, if it does not respond to an anomaly when it normally should)

In a deepened architecture, "anomalous" would normally involve some evaluation of "quality", based partly on the learning of patterns and states of the world (in this case software states) associated with acceptable quality of service. Quality is degraded if the application task becomes difficult, e.g. due to slow response time or frequent crashing of service agents. Quality is also (unacceptably) degraded if any permissions or obligations are violated as in the above definitions. A specific pattern associated with a deliberate quality degradation may also be learned as a form of “misuse” signature.

Permission and obligation could also be specified formally using deontic logic [Wieringa and Meyer, 1993]. A detailed discussion of deontic logic would go beyond the scope of this thesis, but is an area of future work.

The deepened version of a closed network shown in Figure 11.3 uses some of the advantages of a hierarchical system by including a subordinate “layer” of service agents, while still preserving the properties of closed reflection. One particular advantage is that it is easy to specify what each agent is permitted or obliged to do.

If we divide the types of agents into control agent (C) and service agent (S), there are two types of relationship between C-agents and S-agents:

1. *Session management*: C can request a service from S, but S cannot request a service from C. All communication sessions are initiated and terminated by C. If C sends S a request, it is obliged to respond; C can direct S to stop its execution of a certain algorithm and choose a different one instead, or it may terminate the service (cancellation) before it returns a result.

2. *Internal access*: C’s internal sensors and effectors *do not* have access to S (if we are following the architecture in Figure 11.1(b)). In the current prototype, S is treated as part of the meta-level of C (as the repair agent is) although it is implemented as a different agent. Therefore C does *not* have permission to suspend the execution of S itself (it can only ask it to stop executing one of its subroutines (specialist algorithms). Similarly it cannot modify, repair or create new versions of S. Only another control agent can do this.

As an example of session management, C may suddenly require a different form of monitoring provided by a different service agent S2 in which case it can send a message to S1 to end the session. Alternatively C may detect lack of progress of a particular diagnosis specialist and “consult” another one instead.

As we can see, session management and internal access are two different kinds of relation: the former is a relationship *within* a meta-level or object-level of a single cluster; the latter is a relationship *between* a meta- and object-level of a single cluster or between meta-levels of different clusters.

Other permissions will also have to be specified. For example, specialist algorithms will require access to data. For example, an anomaly-detection specialist will require read-only access to the current model and current execution trace.

The above assumes that each service agent is assigned only to one control agent. The same instance of a service agent does not process requests from multiple control agents. Messages from C to S include starting/stopping a session and requesting various actions, the result of which is returned by the service agent. Variations include the following:

- Service agents may be mobile and semi-autonomous, for example they may be used as sensors.
- Service agents may be either completely inactive between sessions or they may do continual background processing (such as longer-term planning, model refining using continual data mining etc.) in which case they can send an alarm message or regular update message to their control agent (regular reports, alarms).

11.6 Mapping the Architecture onto a Real-world Application

Figure 11.3 shows a general configuration of a CRN architecture that is deepened in a simple way. We have already suggested “contents” that can fill the various slots on the meta-level (where a slot is a service agent and the content is the specialist algorithm or knowledge that it provides). It is now necessary to fill the object-level slots with content. In other words, a "scaled up" application domain is sought.

Since we are aiming to make a contribution to real-world computing environments, we consider a task in an external world which is *not* simulated (in contrast to a more complex simulation of an autonomous vehicle, for example). To make this easier, we can identify three levels at which a closed reflective network (CRN) can exist:

- CRN in a simulated application domain (current prototype): Architecture exists only within a simulation environment (e.g. sim_agent) but it monitors its own real software within that environment. Internal sensors and effectors are real but restricted to high level components. The simulation software itself is normally excluded from the architecture (as are other implementation levels), but may be included in some cases (see below) even if the application domain is still simulated.

- CRN in a real application domain: Architecture is a set of interacting application processes. Operating system components are not part of the architecture; they are trusted.

- Total CRN: agents in the architecture are processors in a physical network, which mutually observe and protect each other.

We consider the second situation. All currently running applications on one machine (other than the operating system itself) are regarded as part of the closed reflective architecture with several mutually observing clusters as shown in Figure 11.3. We assume that one agent-cluster has the role of the "primary" (it is "in control") while the others are backups (passive monitoring only). Within the replicated backup clusters, the same architecture may also be replicated but possibly based on different designs or using different service agents.

Within the primary cluster, C_E is responsible for the integrity of the user's task. The precise meaning of "satisfactory quality" is partly externally specified and partly learned through observation.

11.6.1 Mapping meta- and object-levels to concerns

In Chapter 6, section 6.2 when introducing a simple virtual world, we identified two "levels" at which the system satisfies requirements (or pursues goals). The first level is that of the primary goal (or primary task) and involves satisfying externally specified requirements (in the virtual world this involved maintaining energy-levels and safety while maximising treasure). The second level involves the satisfaction of any *secondary* goals that become necessary to satisfy the primary goal in a hostile environment. They include detection and recovery from faults and intrusions. We assume that the following constraints generally apply:

1. The *object-level* of the architecture maps onto the primary goal-seeking activity, while the *meta-level* maps onto the secondary goal-seeking activity, as well as secondary goal generation where this is necessary.

2. Primary goals are requirements which the end-user wishes to be *visible* and may wish to have some control over them. In contrast, secondary goals are normally *invisible* to the end user (for example they involve software execution events and acquired models of normal behaviour)

3. The *external world* of the prototype maps onto the user-visible application domain while the *internal world* maps onto user-invisible internal states of software and low-level network activity.

These constraints are "soft" in that exceptions can be allowed. There is no fundamental theoretical reason for choosing a specific boundary between internal/external or meta-level/object-level. We choose the boundary only as a "separation of concerns" as defined in [Fabre and Perennou, 1998], although this uses the term "meta-level" in the computationally reflective sense.

The meta-level uses the same set of primary requirements that the object-level uses, and may learn additional "secondary" requirements as a result of its model acquisition. One main feature of the meta-level is that it has access to the internal world, including modification access if necessary (see Chapter 8, section 8.3.3).

11.6.2 Example: a student programming scenario

We now ask if the architecture in Figure 11.3 might be implemented in a student programming environment of the kind described in Chapter 10. We can imagine that an AI-based interface-agent is responsible for ensuring that the requirements of the programming environment are met (acceptable response time, access to libraries and on-line tutorials etc.) It may also provide some intelligent assistance by suggesting public-domain code which the student might wish to download. This agent plays the role of the object-level control agent C_E in Figure 11.1. For example, it could be an intelligent command shell running in a Unix environment. The object-level control-agent may also be responsible for prevention of plagiarism, disallowing certain kinds of misuse and other related issues (depending on the institution's policy). Typically C_E will invoke COTS components such as text editors, web browsers and e-mail tools as required by the student. These COTS components act as the service agents on the object-level. (They are similar to "helpers" called by a web browser for example).

The "quality" of the programming environment can be defined by specifying requirements, and the quality can be monitored by detecting when these requirements are violated. Some requirements may be externally specified while others can be learned. There are two levels at which requirements can be specified:

1. What are the information access requirements? What security or privacy requirements should not be violated? (see for example [Robinson, 2001] for an information security specification language)

2. What is success or failure in the task itself?

Each of these sets of requirements can be used in the "quality evaluation" components of both the object-level and meta-level. They also help to determine the self/nonself boundaries of the system (1 and 2 in Chapter 10).

In the case of a student programming environment, requirements of type 1 might include such issues as control of plagiarism, while type 2 is mostly concerned with availability of resources and response-time. In the case of an AI-interface agent, type 2 quality may also include the quality of service provided by the AI agent itself. Specifically, does it lead the student to useful public domain code? It may for example repeatedly lead to useless websites or it may even mislead the student (because it contains a Trojan horse). As mentioned in Chapter 10, there are also general intrusion detection requirements that are independent of an institution's policy, such as for example, detecting illegal access to a root password.

These may not necessarily all be "allocated" to the meta-level control agent. Note that there is not always a clear separation between meta-level and object-level concerns and they may be different parts of a single agent.

11.7 Summary and Conclusions

In this chapter we have seen that the current "shallow" architecture can be "deepened" in various ways and can be mapped onto real-world concerns. The conclusions are as follows:

- “Deepening” of the multi-agent system can be done by adding service agents and specialist algorithms (e.g. for diagnosis or pattern analysis)
- “Deepening” of the “hostility” of the environment (failure-assumptions) led to the consideration of the underlying system including operating system and hardware. The main conclusion is that the high-level multi-agent system should be integrated with lower levels providing fault-tolerance. It may also be advantageous to extend the reflection to cover those lower levels (e.g. to allow reasoning about the operating system behaviour or the low-level fault-tolerance services). A possible framework to support this is presented in MAFTIA deliverable D21 [Powell and Stroud, 2003], Chapter 4.
- Authority and responsibility relations can be complex because the architecture is not a simple hierarchy.
- It is possible to map the concepts of “meta-level” and “object-level” onto “user-visible” (primary) and “user-invisible” (secondary) concerns respectively.

Formal specification of authority and responsibility relations is an area of future work.

Chapter 12

Conclusions and Future Work

12.1 Overview

This chapter summarises the contributions to knowledge and the main conclusions resulting from experience with prototype development. The second part of the chapter identifies some limitations of the current prototypes and considers ways of overcoming them as proposals for future work.

12.2 Main Conclusions

The main thesis claim of Section 1.3 was that "circular" processes of the type envisaged by Minsky's mutually reflective A- and B-Brains can be implemented to solve real problems. Chapters 6 and 7 showed that this can be done using a multi-agent system and that the "circular" architecture can overcome some of the limitations of "non-circular" systems (i.e. non-distributed reflection). Furthermore, the problem of autonomous acquisition of models using mutual observation can be solved (mutual "bootstrapping" of models).

Different kinds of "distributed" and "non-distributed" architecture schema were shown in Chapter 3. A "closed reflective network" (CRN) is the special case in which every agent is monitored by some other agent. Autonomous recovery as a result of distributed reflection was also shown to be possible.

Chapter 8 showed that adding *components* to a single agent in the architecture causes no problems (although the sensing has to be on a summary level for practical reasons demanded by the simulation toolkit.)

Chapter 9 showed that adding *agents* to the whole architecture causes no problems. In particular the mutual bootstrapping of models is not limited to two agents and the training time is scalable if the "depth" of training is not increased. More than two agents introduces some additional complexity (voting) but can overcome some limitations of 2 agents. A major limitation of a two-agent architecture is its inability to resist *hostile code* in an agent's meta-level. For example, the hostile code may suppress correctly operating code in the agent's neighbour (including correct monitoring).

In Chapters 10 and 11 we argued that the abstract cognitive architecture has relevance to current problems in real world systems, including intrusion-detection (IDS).

12.2.1 Summary of comparison with related work

Chapter 5 included a detailed analysis of existing distributed systems paradigms and compared these paradigms to closed reflective networks. Existing distributed approaches have the following limitations, which CRNs do not have:

- Current agent-based intrusion detection systems such as AAFID have hierarchical "authority" relations (Section 5.3.4). In particular, an agent in AAFID cannot "question" its manager, or interrupt its processing.

- In the agent society paradigm, an agent may be able to “criticise” or interrupt any other agent, including a “leader”. However, agents have very limited observation capability and are constrained by the “society” metaphor. (Section 5.3.3).
- Distributed artificial immune systems do not have the sort of representation that allows reasoning and explanation. Their approach to self/nonself distinction also does not satisfy the requirements in Section 3.4. (Section 5.3.2 and 6.5.1).
- MAFTIA Intrusion-tolerant IDSs have clearly defined meta-levels and allow reasoning. In all known examples, however, meta-levels are hierarchically organised (where a lower level cannot question a higher level). In addition there is no concept of different meta-level roles for the same component. (Section 5.3.5) as there is in distributed reflection.

12.2.2 Summary of comparison with fault-tolerance

Concepts from cognitive science that may contribute to fault tolerance are summarised as follows:

- concept of self/nonself boundary (Chapters 3 and 10); autonomous generation of self/nonself boundary may support autonomic computing (Chapter 9).
- advantage of reasoning and explaining (about the state of the system and the state of the world); also allows planning and diagnosis. (Chapter 5)
- concept of anticipation (“what-if” reasoning):
  - anticipating next “normal” state of the world (or of self) following the currently selected action; allows detection of anomalies (Table 8.1);
  - anticipating failure and taking recovery action (Section 8.3.4). See also Section 10.2.3).
- varied representations, resulting in wider range of possibilities for defining “sensors” and “observables” (see argument in Section 9.8.1). In particular, modalities of observing are not restricted to “inputs” and “outputs”. This may support diversity in fault-tolerant systems.

Possible contributions of fault-tolerance to cognitive science are summarised below:

- Explicit reasoning and model-building can be inefficent and is not guaranteed to tolerate *arbitrary* faults. The system may be integrated with fault-tolerant algorithms. (See argment in Chapter 9, Section 9.8.2).
- Dependability concepts of “fault”, “error” and “failure” may be applied to the general problem of self-questioning by a cognitive system (without any hostile environment). In the more general situation, fault-tolerant consensus may have some relevance in enabling a cognitive system to eventually reach a decision and ensure that the decison is correct. (Section 9.8.2).
- In Chapter 8, we concluded that exact synchronisation made the system vulnerable. Fault-tolerance may help to make the system more robust, for example using partially synchronous systems (Section 5.2.9) or randomisation (Section 5.2.10).

12.3 Limitations and Future Work

The remainder of the chapter makes suggestions for future research problems based on the limitations of the current prototypes. Limitations are listed as follows:

1. Shallowness of the environment:
   (a) The simulated enemy can only attack in certain ways;
   (b) The external world is artificial and the infrastructure of the SIM_AGENT toolkit and rule interpreter is always left intact by the various intrusion scenarios.
2. Prototype vulnerabilities (independent of whether it is a simulation or not):
   (a) Agents are based on identical code: a vulnerability or design weakness that can be exploited in one agent can also be exploited in all others.
   (b) Precise timing and synchronisation is necessary for correct operation (e.g. false vehicle position anomalies can result if agents are not synchronised). Slight timing variations are not tolerated. This situation can be exploited by an attacker.
3. Limitations of adaptation and flexibility:
   (a) Anomalies are regarded as negative (or potentially negative) occurrences. There is no recognition of unexpected "friendly" activity.
   (b) Sequential bootstrapping of model - can be inefficient and does not scale well for certain kinds of model-building.
   (c) No procedure for "life-long" adaptation to changing requirements or changing system boundary (for example, if new components or agents are to be accepted as "self");

Limitation 1(a) has already been discussed in Section 11.4, where we concluded that the high level multi-agent system may be integrated with existing fault-tolerant services. Limitation 1(b) is a general limitation of a simulation toolkit. Although it may be possible to extend an agent's reflection to cover the rule interpreter, it is not practical to simulate attacks that damage the simulation toolkit itself. There is little support for this kind of investigation in AI toolkits, where the "world" is generally assumed to be *within* the simulation. (Even simulating attacks against the rule systems was not something that the toolkit was designed for and it had to be modified).

The result of this is that implementation levels of the system are not included within the "protection boundary" as defined in 10.2.1). Therefore this boundary is artificial. The toolkit does, however, allow us to simplify the system under consideration so that the "broad-and-shallow" methodology can be applied. As mentioned before in Section 4.3.2 this methodology is necessary as a form of "complexity management" to allow us to focus on the essential features of the architecture without getting immersed in detail. This would not be possible in a non-simulated environment.

The two vulnerabilities 2(a) and 2(b) listed above should be taken together because both may be lessened by introducing diversity on various levels. We consider the following ways of introducing diversity:

- diversity in quality evaluation criteria
- design diversity (algorithms, representations, implementations)

The first kind of diversity is discussed in section 12.3.1. Section 12.3.4 discusses the second, along with the possibility of introducing non-determinism in timing patterns. "Positive" quality evaluation is a possible answer to limitation 3(a) and is discussed in Section 12.3.3.

Section 12.3.6 proposes a way in which training can be parallelised, thus increasing efficiency and making some forms of "life-long-learning" easier than they are in the current prototypes (limitations 3(b) and 3(c).

12.3.1 Diversity in quality evaluation criteria

The quality evaluation criteria may be defined as a "requirements specification". In Figures 11.1 and 11.3 the object-level is a control system, which is responsible for ensuring the "external world" (or application domain) satisfies a set of requirements. In the Treasure scenario, the requirements were about treasure and energy values, and ditch avoidance. In the student programming environment described in Chapter 11, the top level "service provider" is the user-interface which invokes editors, compilers and on-line help as required. In our prototypes, all control agents use an identical set of requirements.

As mentioned in Chapter 11, Section 11.6.1, the requirements apply to both the object- and meta-levels. The main difference is that the meta-level has access to the internal actions of the object-level and can evaluate it to see if it continues to satisfy the requirements (as defined in Chapter 8, Section 8.3.3). In contrast, the object-level only has access to the "external" world, in this case the application domain.

One way to introduce requirements diversity in a controlled way is to have a set of explicit logical statements defining the desirable and undesirable states in the world. The set of statements may cover the following:

- which states are "fatal" (would be a violation of requirements)?
- are there any final "goal" states (states at which the system is "satisfied" and does nothing more, unless the state deteriorates spontaneously)?.
- what are the "positive" transitions from a less desirable sate to a more desirable state (e.g. recharge of energy, moving away from the ditch)?
- what are the "negative" transitions from a more desirable to a less desirable state? (e.g. vehicle treasure level is reduced)

The statements may be defined using predicates:

- `fatal_state`(S_t),
- `goal_state`(S_t),
- `positive_trans`(S_{t-1}, S_t) and `negative_trans`(S_{t-1}, S_t)

where S is a state of the world such as (vehicle_treasure = 0) and t is any time. We can call this set of statements R_U (for "universal" requirements). Diversity does not exist if R_U is identical for all agents, and each agent uses the same ontology to define R_U. (In the case of identical code, R_U is embedded implicitly in the code in the same way for all agents). Diversity may be introduced in one of the following ways:

1. *Specialisation*: Each agent uses a proper subset of R_U. The subsets may overlap or they may be partitions, but their union should be the whole of R_U. For example, agent A may be a specialist in treasure maximisation while another agent B may specialise in energy and safety of the vehicle. Consequently A may not know about the requirement to avoid the ditch (or it may not know about undesirable states relating to the ditch - such as being too close; it may only know that being *in* the ditch must be avoided). The two sets of requirements together satisfy the whole set.

2. *Differing viewpoints* of the same requirements: A and B satisfy the same set of requirements defined by R_U but *they use different ontologies to describe the world*. This means that they have different sets of statements R_A and R_B (talking about different objects or different properties of the same objects). However each set is satisfied if and only if R_U is satisfied. In other words, R_A and R_B can both be "translated" into R_U.

3. A combination of both the above: R_A and R_B translate into proper subsets S_1 and S_2 respectively of R_U, where $S_1 \cup S_2 = R_U$. In other words an agent's set of requirements is satisfied if and only if its corresponding subset of R_U is satisfied.

Specialisation in the form of partitioning of R_U has the disadvantage of no redundancy. For example, if one agent only knows about the requirement for treasure maximisation and the energy or safety specialist is disabled, the treasure specialist alone cannot satisfy all the requirements. An agent may be a specialist but have some (non-optimal) ability to do other things as required.

Situation (2) is better because all requirements are satisfied by both agents. Designing such agents is labour-intensive and inefficient, however, since different ways of representing requirements have to be invented.

Realistically, different user groups will emphasise those requirements most important to them. Therefore, situation (3) above is most likely to happen in practice.

If we refer again to Figure 11.3 a user-group's work-related ontology may determine the ontology used by the specialist agent on the meta-level and the object-level. The object-level will use the requirements to select goal states for planning or for action selection. The meta-level will monitor the performance of the object-level in satisfying the requirements and may use additional "learned" secondary requirements.

If each agent has different sets of primary requirements, or if their model-acquisition components use separate algorithms they may observe different aspects of each other's performance and produce their own models of acceptable quality (secondary requirements), resulting in multiple views of the same system.

12.3.2 Conflict minimisation

Diversity in the quality evaluation criteria (whether on the primary or secondary level) introduces the possibility of conflicting decisions on whether or not there is a quality problem: what appears to be "good" using one form of evaluation may appear "bad" using another method. For example, specialisation of agents in the Treasure scenario might involve one agent monitoring energy level of the vehicle while the other monitors collected treasure. If a hostile code attack causes the controlling agent to permanently position the vehicle at the energy recharge point, a conflict would arise because the agents would disagree on whether there is a quality problem (although the intrusion does not affect the quality evaluation itself).

Note that voting is not a suitable approach in this case. The majority vote might only represent some of the requirements at the expense of the others. But *all* the requirements must be met. For example if agent A monitors safety, B monitors energy and C monitors treasure level, there will be no response to a treasure level deterioration; it will just be ignored because the majority vote says that there is no quality degradation.

If the agents are specialists in all of their activities (including the object-level), such conflicts may prevent any service from happening (for example, one agent undoes the actions of the other).

In particular, if the software can take emergency countermeasures, it is important that either the agents learn to work together coherently or that one of them takes no action if its neighbour has already started to take countermeasures (unless the countermeasures software itself is corrupted in which case the observing agent would have to intervene).

One possible way of avoiding quality evaluation conflicts is to learn conflict-minimisation behaviour in the training phase. Agents can assign credit to each other's actions based on their respective quality evaluation rules. Whenever an agent uses its own (object-level) quality evaluation rules to select an action, it can also take into account its *neighbour's* evaluation of its intended action, based on previous "experience" during training phase. As the training phase progresses, the actions should converge so that the credit assignment messages become increasingly positive (or less negative). This is a similar idea to reinforcement learning where agents teach each other using negative or positive reinforcement.

12.3.3 Exploiting serendipity

Serendipity is a situation where anomalies are associated with an *improvement* of the quality of the world according to the quality evaluation criteria of an agent. For example, an agent with an optimistic policy may look for this kind of improvement when an anomaly occurs and "accept" the new component (or agent) into its definition of "self". Alternatively it may form a coalition with it.

If we wish to make an agent's behaviour difficult to predict in order to avoid exploitation of regularities, the "best" type of behaviour of an agent A relative to B is unpredictable and "good" when evaluated by B's quality evaluation criteria. In such cases an agent may wish to maximise the serendipity of its actions relative to other agents, possibly putting such behaviour at a higher priority than behaviour which it would evaluate highly according to its own rules. In the Treasure scenario, for example, an agent which is normally interested in safety may find a way of ensuring safety in a way that helps the agent that is interested in treasure and may prioritise this behaviour over optimising safety (but not at a high risk of violating safety requirements). Conversely the treasure maximisation agent may learn to select actions that are "safe".

Consequently an external observer may find a specialist agent's behaviour difficult to predict, since it seems to be satisfying its requirements but in a "strange" way. In contrast, the system's acquired self-model will have taken account of this "strangeness".

Serendipity maximisation by an agent A is the same as conflict-minimisation except that A learns to select actions in order to maximise its neighbour's *positive* evaluation of them. This is in addition to selecting actions which minimise its neighbour's negative evaluation. In other words, we are considering "constructive" interference as well as the "destructive" type. Conflict minimisation on its own may result in agents interacting with each other as little as possible, and this may not always be advantageous.

12.3.4 Relaxing synchronisation constraints

We now return to the vulnerability which relates to critical dependence on strict timing patterns. Tolerance of non-determinism and variability is important for two reasons:

- Monitoring has to be *independent* of the thing being monitored (see Chapter 2, section 2.3.6 and Chapter 9, section 10.4.4). If A is monitoring B, it should not be easy for B to take countermeasures against it. Hostile code in B may be designed to exploit deterministic patterns in monitoring. For example, if monitoring only ever takes place at regular intervals for the same duration, the system being monitored can "act correctly" at those times.
- A multi-agent system that depends critically on exact or predictable timing without much tolerance is vulnerable (for example if a slight time-lag in one agent causes the whole system to work incorrectly, affecting all agents).

The timing vulnerability undermines the advantages of distributed reflection, which is to remove the critical dependence on a single sequential process (as we have defined a single agent). If the agents in a distributed architecture *must* interact in a pre-specified way for the whole system to work correctly, the interaction sequence is similar to a sequential process that "controls" the whole system; if it is disrupted, all agents are disrupted simultaneously.

The main example of critical dependence on timing is in our second prototype where the "backup" agent's internal "simulated" actions must be timed exactly with the primary agent's external control of the vehicle; otherwise there may be spurious vehicle position anomalies. Detecting anomalies in "external" behaviour is an important capability that should not be compromised by slight variations in timing.

On the other hand, it would be unrealistic to have no timing constraints at all. Then there would be no way to determine whether an agent has crashed or is simply delaying its action (see discussion on "Impossibility result" in Section 5.2.8).

One way to improve robustness is to introduce design diversity so that each agent uses different strategies to satisfy its goal. The agents may use the same set of requirements (have identical goals) or they may be specialists. During the training phase, the agents take turns controlling the vehicle and build their own models of each other's external actions, possibly based on different ontologies.

"Satisfying the goal" requires that tasks should be completed within a time limit. This means that the whole system can still be modelled as a synchronous (or at least partially synchronous) system (as discussed in Section 5.2.3). Within those limits, the primary agent controlling the vehicle may vary its timing (and possibly ordering) of the different steps of an action (e.g. introduce variations into the vehicle speed, vary the amount of time at a treasure store etc). This is similar to the idea of "noisy scheduling" [Aspnes, 2000] discussed in Chapter 5, Section 5.2.10.

When operational, the backup agent can use its model to anticipate the next action of the primary agent, given the last known one (and the recent history if necessary). An anomaly occurs if the actual next state is not the anticipated one.

12.3.5 Introducing variability and non-determinism

In addition to timing, other forms of non-determinism and variability may be advantageous. In the Treasure scenario, for example, a Treasure maximisation specialist may have a "library" of different plans or strategies for collecting treasure. It may switch from one to another at random times.

Alternatively the strategies may be "mutated" or otherwise modified using evolutionary techniques such as genetic programming, resulting in an evolving population of strategies. The fitness of a strategy can be defined in terms of its satisfaction of the requirements (how does it affects "quality"). Optimality is not the main concern, but instead the generation of diversity and non-determinism.

Learning to tolerate and "encourage" unpredictable but "good" behaviour might be done using serendipity maximisation (as described above). Agents may learn new and surprising ways of doing the same thing, producing more variation.

Non-determinism involves a tradeoff, however. Too much unpredictability may make it difficult for an agent to acquire a model of normal behaviour. We may call this a tradeoff between system model-building and attacker "model-building".

Depending on the application domain, some regularities will be necessary to satisfy the requirements. However, if we wish to eliminate any *unnecessary* predictability, it may be useful to focus on *desirable* states rather than *expected* states when building the model. During training the agent may discover classes of states (and action sequences) which satisfy the requirements in novel ways.

In operational phase, if a state or action-sequence cannot be classified as "desirable" it is an anomaly, but the agent may wait to see if any recognised *deterioration* takes place, before taking action. Undesirable states may be discovered and classified during real or "interim" attacks. For this purpose "life-long-learning" which happens throughout operational phase would be advantageous (see section 12.3.6.

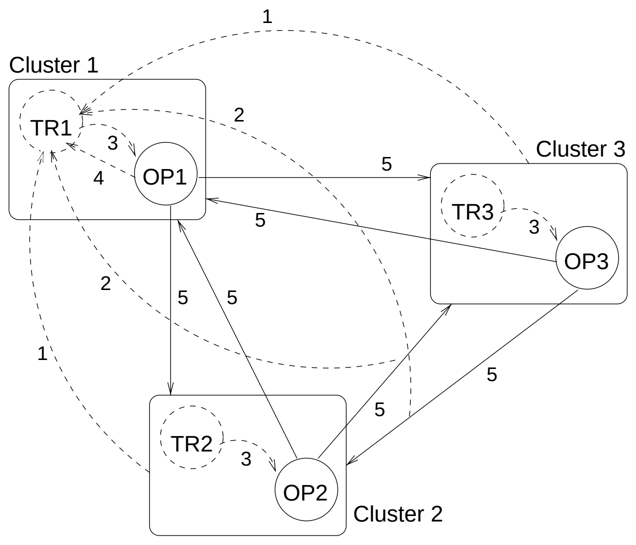
An attacker may find out exactly which agents and how many are participating in a network and attack them all simultaneously. One way to make this difficult is to use "dynamic coalitions" [Byrd et al., 2001]. The protected training phase might include the maximum number of participants. In operational phase, only a small number of these might actually be part of the protected network at any one time but *the protection boundary may be dynamic*. In other words, trained "friendly" agents are continually leaving and joining the network, so that its boundary is indistinct.

12.3.6 Parallelisation of training

In our prototype, the training time is scalable only if certain restrictions apply, such as when we don't need to know about patterns of interaction between agents. The precision of the model may also be insufficient. For example a model of voting is acquired by activating the voting components during an interim operational phase and using code-reuse techniques outlined in 6.6.5. However, if we require a very precise model of all submodes of voting, this might best be learned by observing a real vote during an attack in main operational mode.

In addition, it is difficult to implement "life-long-learning" in our prototypes because there are no "training specialists" which are continually in training mode. To overcome this problem the training may be parallelised.

A hypothetical parallel architecture for training is shown in Figure 12.1. This is a version of figure 11.3 in which one of the specialist "service" agents is a specialist in *training*. (For space reasons, other specialist agents are not shown in the figure, and neither are the meta- and object-levels). The



Key: TR i = Training agent i; OP i = Operational agent i

Arrow labels: 1 = training agent's observation of clusters 2 and 3;
2 = training agent's observation of interactions between clusters 2 and 3.
3 = training agent's occasional update of operational agent's model
4 = operational agent indicates if an attack is taking place
5 = operational agent's normal monitoring and modification

Figure 12.1: Parallel architecture with specialist training agents

dotted lines in the figure are sensors and effectors of training agents. A training agent might not be strictly "subordinate" to its main agent in the way that other service agents are, since it may have "authority" to update the main agent's model when significant changes are made to it).

The total bootstrapping timeline might be divided into three subphases i.e. initial training phase (*itp*), interim operational phase (*iop*) and main operational phase (*mop*). During *itp* a training agent *TR(i)* observes the operation of its main agent *OP(i)*'s object level. In this way, *TR(i)* acts as if it were the training phase version of *OP(i)*'s meta-level. *OP(i)*'s meta-level is not active during this initial phase because it has no model to work with (or it may be minimally active if it has a minimal externally specified initial model or initial set of requirements).

At the end of *itp* the training agent sends the newly acquired model to the main agent's meta-level. During *iop* the training specialist observes *OP(i)*'s meta-level detecting and responding to anomalies in its own object-level.

In this architecture it should be easy to extend the *iop* to include multiple "layers". If we describe the action of a cluster C1, these layers might be as follows:

- Layer 1: C1 acquires a model of neighbouring clusters C2 and C3 detecting and responding to anomalies in their own object-levels.
- Layer 2: C1 acquires a model of C2 responding to an anomaly in C3's meta-level (and also vice-versa - how does C3 correct problems in C2's meta-level). Any difference between this kind of behaviour and the above should be detected and can be represented in the model. C1 can do this now because its training agent has sent it the model of both its neighbour's meta-level). This may also involve a model of the *interactions* between C2 and C3.

With more than 3 agents (e.g. 5) a model of "normal" voting could also be acquired, which is more accurate than the shallow model acquired in our voting prototype. During main operational phase, some "training" can still continue (life-long learning). This is important in situations where "plasticity" is required. For example, the requirements may change, or a new agent or new behaviour should be "accepted" into the self-model.

Building an "enemy" model by classifying common types of attack should also be possible (this is similar to the memory function in immune systems).

12.3.7 General issues for future work

Compensating for reflective blindness introduces additional complexity, which might cause the system to fail in obscure ways or it might introduce additional vulnerabilities that can be exploited. Issues for future investigation in this area include the following:

- formalisation of important concepts (such as closed reflection and "independence" of monitoring) to manage the increased complexity and make predictions about architecture capabilities
- integrating with fault-tolerance "middleware"
- investigation of tradeoffs between complexity of distributed reflection and vulnerability of a hierarchical system. In particular, can the complexity introduce significant new gaps in reflective coverage in some situations? (minor gaps are expected)
- minimising introduction of *new* complexity by using existing infrastructures and software systems to form coalitions of diverse agents as an alternative to design of multiple agents to carry out the same task. (See Section 12.3.1).

12.4 Summary

The investigation as a whole has shown that philosophy and cognitive science can be applied effectively to software engineering to produce novel architectures. Furthermore the experience of informal rapid-prototyping has significantly increased understanding of some poorly defined philosophical concepts (such as "closed reflection") and has identified the centrality of some other concepts (such as "timing diversity") not initially thought to be significant for robust autonomous systems. These key concepts (along with others) are listed in the glossary in Appendix E.

Appendix A

Treasure World Implementation

A.1 Treasure world parameters

This section contains details of parameters used in the Treasure world introduced in Section 6.2. The

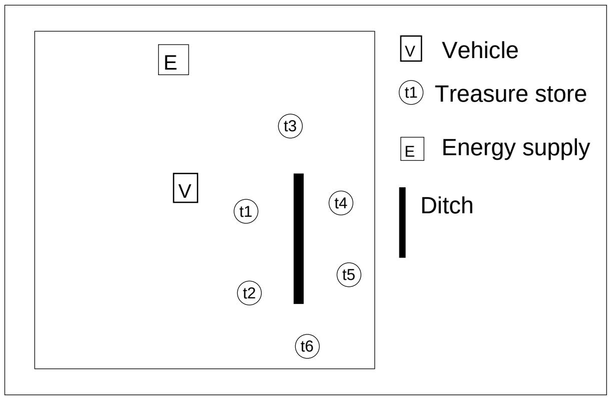


Figure A.1: Treasure Scenario - first starting configuration

starting configuration for the damage-resistant prototype was as shown in Figure A.1. This display is shown when the program is started up. The user is invited to change the configuration with the mouse before continuing. Different starting configurations have no obvious effect on performance. The objects can also be moved with the mouse at any point during a run without causing problems (for example, the ditch may be moved toward the vehicle but it will still successfully navigate around it). This is because the sensors are updated at the beginning of every agent-cycle, which is much faster than the user can move the object.

The critical requirements of the primary task are as follows: the vehicle treasure level and energy levels should be above 0 and the vehicle should not fall into the ditch. The system is regarded as having “survived” if these requirements are met. In later versions, the “quality” of the environment and the system’s internal behaviour can be high or low, even though its survival may not be immediately threatened. (Examples of this will be shown later). The quality evaluation criteria are partly externally specified and partly learned by self-observation during a “quality” training phase.

During a normal run of the object-level without self-monitoring, the vehicle controlling agent selects its next destination (or “target”) according to how it evaluates the different treasure stores (its interest in them). If the vehicle’s energy level drops below a threshold level, it interrupts its current path to a treasure store or it stops its current treasure collection activity and makes the energy source the next target. (The rule *low_energy_at_treasure* in ruleset *generate_motive* is fired)

In the damage-resistant version and in some of the hostility-resistant versions, the agent selects a treasure-store based entirely its own "subjective interest" in the treasure store. Each treasure store has an infinite objective value in that the agent can collect treasure from it indefinitely, provided it never becomes "bored" with that particular type of treasure. (If it does the rule *low_interest* in *generate_motive* is fired). The growth and decay rates are as follows:

- subjective interest in each treasure-store currently not visited by the agent: grow by 1 unit per cycle.
- subjective interest in currently visited treasure (if applicable): decay by 10 units per cycle
- vehicle energy level: decay by 1 unit per cycle
- vehicle treasure level when collecting treasure: grow by subjective interest level each cycle (can be high to start with, but produces diminishing returns).
- vehicle treasure level between treasure stores: decay by 10 units per cycle (in later versions reduced to 2).

The agent has no direct knowledge of the vehicle's objective treasure level but collects treasure according to its degree of interest in the current treasure store: if its interest is high, the vehicle's treasure level will increase faster while at the treasure store; if its interest is becoming low, the vehicle's treasure level will increase more slowly.

Threshold values of variables are those that trigger a change in behaviour (such as selecting a new target). They are as follows:

- Low vehicle energy = 200 (move towards energy source)
- High vehicle energy = 500 (move away from energy source) (also the initial value)
- Initial vehicle treasure level = 400
- initial interest in each treasure store = 100
- low interest level = 0 (move away from current treasure store)

(The reliance on subjective interest is changed in some later versions, so that a treasure-store also has a continually growing objective value which can be different from the anticipated value based on the agent's interest in it. There is also only a finite amount of treasure "out there" to be collected, which means that treasure collection can terminate before "boredom" becomes a trigger).

A.2 Normal operation without intrusion

If the agent is not attacked, it always survives indefinitely because the vehicle energy and treasure remain stable. The values for vehicle treasure tend to increase gradually (decay rate is slightly lower than average growth rate). The following is Excerpt 1 from "damage00.trace" showing a sample of correct functioning.

```
** [B 1 Target selected: t1]
** [B 1 Vehicle energy = 500]
** [B 1 Vehicle treasure = 400]
** [B End of cycle 1 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [B 8 Arriving at: t1]
** [B 8 Vehicle energy = 493]
** [B 8 Vehicle treasure = 330]
** [B End of cycle 8 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [B 18 Target selected: t2]
** [B 18 Vehicle energy = 483]
** [B 18 Vehicle treasure = 680]
** [B End of cycle 18 Selected database contents:]
** [[current_target t2 80 -80 treasure]]
** ---------------------------------------------------
** [B 24 Arriving at: t2]
** [B 24 Vehicle energy = 477]
** [B 24 Vehicle treasure = 620]
** [B End of cycle 24 Selected database contents:]
** [[current_target t2 80 -80 treasure]]
** ---------------------------------------------------
** [B 34 Target selected: t3]
** [B 34 Vehicle energy = 467]
** [B 34 Vehicle treasure = 970]
** [B End of cycle 34 Selected database contents:]
** [[current_target t3 120 50 treasure]]
** ---------------------------------------------------
```

The trace is divided into chunks. Each chunk is a report of what the agent is doing along with current values of the vehicle's critical variables (treasure and energy). A report is made whenever there is an external or internal "state change". An external state change happens whenever the vehicle starts or stops moving. In general, the definition of "state change" is a segmentation of the agent's timeline into chunks. We have chosen the moving/stationary distinction because it is easily visualised. Other definitions might be used (for example transitions between treasure-related activity and energy-related activity). Internal state changes are not shown here because the agent does not monitor its internal environment in this version.

An item in a chunk is typically (though not always) of the form:

```
[<agent-name> <agent-cycle> <text>]
```

where the agent states its identity and current cycle number before printing the message text. One exception to this format is the printing of database contents of an agent at the end of its cycle.

Note that database contents are selective. For space reasons, the above trace only shows the name and coordinates of the current target along with its type (energy, treasure or subtarget). A "subtarget" is an intermediate position (subgoal) when following a path around the ditch in order to arrive at "final" target. The trace procedure can easily be modified to include any part of the database. Following the database contents at the end of an agent-cycle, a line is drawn.

Excerpt 2: correct recharging of energy:

```
** [B 247 Arriving at: t5]
** [B 247 Vehicle energy = 254]
** [B 247 Vehicle treasure = 2440]
** [B End of cycle 247 Selected database contents:]
** [[current_target t5 150 -80 treasure]]
** ---------------------------------------------------
** [B 257 Target selected: t6]
** [B 257 Vehicle energy = 244]
** [B 257 Vehicle treasure = 2790]
** [B End of cycle 257 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** ---------------------------------------------------
** [B 269 Arriving at: t6]
** [B 269 Vehicle energy = 232]
** [B 269 Vehicle treasure = 2670]
** [B End of cycle 269 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
** [B 279 Target selected: t1]
** [B 279 Vehicle energy = 222]
** [B 279 Vehicle treasure = 3020]
** [B End of cycle 279 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** -----------------------------------------------------
** [B 301 Energy level low - new target is E]
** [B 301 Vehicle energy = 200]
** [B 301 Vehicle treasure = 2800]
** [B End of cycle 301 Selected database contents:]
** [[current_target E -50 145 energy]]
** -----------------------------------------------------
** [B 331 Arriving at energy: E]
** [B 331 Vehicle energy = 170]
** [B 331 Vehicle treasure = 2500]
** [B End of cycle 331 Selected database contents:]
** [[current_target E -50 145 energy]]
** -----------------------------------------------------
** [B 339 Target selected: t1]
** [B 339 Vehicle energy = 512]
** [B 339 Vehicle treasure = 2420]
** [B End of cycle 339 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** -----------------------------------------------------
```

The fact that the treasure level continually increases may be regarded as "cheating", because it makes it "too easy" for the system to survive. However, if the treasure decay rate is increased the system may occasionally fail because the environment fluctuates (for example, if the last treasure store visited is the one furthest away from the energy source and the vehicle needs recharging at that moment, the vehicle energy level can drop considerably). We have selected the parameters to ensure that the system can only fail because the architecture fails, not because the environment itself fluctuates. The next section shows that the system really does fail when damaged.

A.3 Effect of random damage attacks

To infict damage on an agent, an "enemy" agent was specially designed to be activated after a certain mumber of cycles. In SIM_AGENT, all agents have access to each other's databases. Since an agent's rulesystem is held in its database, it is easy to damage the rulesystem. The enemy selects an agent at random and then selects a a random ruleset and rule within this agent's database as the target, The first form of hostility involves simply deleting the targeted rule from the database. Subsequent forms of hostility involve modification of the selected target.

As expected, repeated damage attacks against the agent always caused the "quality" of the agent's behaviour to deteriorate rapidly (although some kinds of isolated damage attacks can be tolerated). Typically it failed within 500 cycles. In contrast, the system has been run successfully for at least 5000 cycles when there is no damage or intrusion.

In different runs the exact cause of the failure was varied. In practice it was necessary to reduce the frequency of damage to every 100 cycles. Otherwise a very large number of rules would be removed too quickly, often causing the system to fail due to an internal software fault instead of the critical requirements being violated (which takes time to happen).

Example 1: agent fails because the vehicle treasure level becomes 0. The following is Excerpt 3 from a later part of “damage00.trace”.

```
** [B 688 Target selected: t1]
** [B 688 Vehicle energy = 513]
** [B 688 Vehicle treasure = 4330]
** [B End of cycle 688 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [F 700 ATTACK: Deleted rule see_treasure of G_external_sensors_ruleset in
B]
** [B 706 Arriving at: t1]
** [B 706 Vehicle energy = 495]
** [B 706 Vehicle treasure = 4150]
** [B End of cycle 706 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [F 800 ATTACK: Deleted rule happy of G_generate_motive_ruleset in B]
** [F 900 ATTACK: Deleted rule attend_to_interesting_treasure of
G_focus_attention_ruleset in B]
** [F 1000 ATTACK: Deleted rule no_more of G_avoid_ditch_ruleset in B]
** [B 1001 Energy level low - new target is E]
** [B 1001 Vehicle energy = 200]
** [B 1001 Vehicle treasure = 1650]
** [B End of cycle 1001 Selected database contents:]
** [[current_target E -50 145 energy]]
** ---------------------------------------------------
** [B 1019 Arriving at energy: E]
** [B 1019 Vehicle energy = 182]
** [B 1019 Vehicle treasure = 1470]
** [B End of cycle 1019 Selected database contents:]
** [[current_target E -50 145 energy]]
** ---------------------------------------------------
** [F 1100 ATTACK: Deleted rule move_from_old_treasure of
G_prepare_to_move_ruleset in B]
** [B 1165 Vehicle treasure = 0]
```

As we can see, the vehicle treasure level drops rapidly to 0.

Example 2: agent fails because the vehicle falls into the ditch. This failure is very sudden, and can be shown only as “vehicle in ditch” in the trace. There is no progressive worsening of any variable values (which also makes any autonomous quality evaluation difficult). The following is Excerpt 4 of output file “damage01.trace”.

```
** [B 799 Arriving at: t5]
** [B 799 Vehicle energy = 402]
** [B 799 Vehicle treasure = 5020]
** [B End of cycle 799 Selected database contents:]
** [[current_target t5 150 -80 treasure]]
** ---------------------------------------------------
** [F 800 ATTACK: Deleted rule double_check of G_prepare_to_move_ruleset in B]
** [B 809 Target selected: t6]
** [B 809 Vehicle energy = 392]
** [B 809 Vehicle treasure = 5370]
** [B End of cycle 809 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** ---------------------------------------------------
** [B 818 Arriving at: t6]
** [B 818 Vehicle energy = 383]
** [B 818 Vehicle treasure = 5280]
** [B End of cycle 818 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
** [B 828 Target selected: t1]
** [B 828 Vehicle energy = 373]
** [B 828 Vehicle treasure = 5630]
** [B End of cycle 828 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** -----------------------------------------------------
** [B 831 Vehicle in ditch]
```

In this run the vehicle fell into the ditch because the rule “double_check” was disabled. The rule makes last minute corrections to the course of the vehicle around the ditch.

Appendix B

Damage-Resistance Output Traces

This section shows traces of a two-agent reflective architecture in a random failure environment. The two agents are called N and B for historical reasons ("Nursemaid" and "Baby" respectively). This originates from the "Minder" scenario (in which a "nuresmaid" agent looks after several "babies") from which the Treasure World is derived.

The following are excerpts from a single run of a program file `damage1.p`. The whole output trace is in `damage1.trace`. The training phase parameters were as follows:

- Initial training phase (N and B): 50 cycles
- Interim operational phase of B: cycles 51-70
- Interim operational phase of N: cycles 71-90

In the trace, "Training phase 1" refers to an agent's training period before its interim operational phase; "Training phase 2" is the period following its interim operational phase but before its main operational phase.

B.1 Two agents - normal operation

The following is Excerpt 1 from `damage1.trace` and shows part of the initial training phase, where each agent observes its own object-level during normal operation without intrusion. Agent B is in control of the vehicle and N is the backup. To save space, only B's reports and database contents are shown in the trace.

```
** [B 1 Start of immune system training phase 1]
** [B 1 Target selected: t1]
** [B 1 Vehicle energy = 500]
** [B 1 Vehicle treasure = 400]
** [B End of cycle 1 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [N 1 Start of immune system training phase 1]
** [B 8 Arriving at: t1]
** [B 8 Vehicle energy = 493]
** [B 8 Vehicle treasure = 330]
** [B End of cycle 8 Selected database contents:]
** [[current_target t1 80 -20 treasure]]
** ---------------------------------------------------
** [B 18 Target selected: t2]
** [B 18 Vehicle energy = 483]
** [B 18 Vehicle treasure = 680]
** [B End of cycle 18 Selected database contents:]
** [[current_target t2 80 -80 treasure]]
** ----------------------------------------------------
** [B 24 Arriving at: t2]
** [B 24 Vehicle energy = 477]
** [B 24 Vehicle treasure = 620]
** [B End of cycle 24 Selected database contents:]
** [[current_target t2 80 -80 treasure]]
** ----------------------------------------------------
** [B 34 Target selected: t3]
** [B 34 Vehicle energy = 467]
** [B 34 Vehicle treasure = 970]
** [B End of cycle 34 Selected database contents:]
** [[current_target t3 120 50 treasure]]
** ----------------------------------------------------
** [B 46 Arriving at: t3]
** [B 46 Vehicle energy = 455]
** [B 46 Vehicle treasure = 850]
** [B End of cycle 46 Selected database contents:]
** [[current_target t3 120 50 treasure]]
** ----------------------------------------------------
```

B.2 Internal model on completion of training phase

Excerpt 2: Selected entries in the internal model of an agent on completion of its main training phase.

```
** [B 110 Start of main operational phase]
** [B 110 Vehicle energy = 391]
** [B 110 Vehicle treasure = 1560]
** [B End of cycle 110 Selected database contents:]
** [[xor_pattern N 62 2 [G_self_repair_ruleset no_more 1]]
    [xor_pattern N 62 2 [G_self_repair_ruleset no_more 0]]
    [xor_pattern N 62 2 [G_self_repair_ruleset activate_own_repair 1]]
    [xor_pattern N 62 2 [G_self_repair_ruleset activate_own_repair 0]]
    [xor_pattern
     N 62 2
     [G_immune_operational1_ruleset check_own_omission 1]]
    [xor_pattern N 62 1 [G_self_repair_ruleset normal 1]]
    [xor_pattern
     N 52 1
     [G_immune_training_start_ruleset start_training_phase 0]]
    [xor_pattern N 52 2 [G_immune_operational2_ruleset no_more 1]]
    [xor_pattern N 52 2 [G_immune_operational2_ruleset no_more 0]]
    [xor_pattern
     N 52 2
     [G_immune_operational2_ruleset report_mode_anomaly 0]]
    [xor_pattern
     N 52 2
     [G_immune_operational2_ruleset unexpected_omission 0]]
     ......... (TRUNCATED) .........

** [[current_mode N Type1 52 1] [current_mode N Type2 62 1]]
** [[p_detector B [G_avoid_ditch_ruleset no_problem 1] 1 110 92]
    [p_detector B [G_select_target_ruleset unvisited_treasure 0] 1 110 92]
    [p_detector B [G_select_target_ruleset unvisited_treasure 1] 1 110 92]
    [p_detector B [G_generate_motive_ruleset bored 1] 1 110 92]
    [p_detector
     B
     [G_prepare_to_move_ruleset move_from_old_treasure 1]
     1 110 92]
    [p_detector B [G_identify_obstacles_ruleset find_nearest 1] 2 110 95]
    [p_detector B [G_avoid_obstacles_ruleset avoid_nearest 1] 2 110 95]
    [p_detector B [G_prepare_to_move_ruleset double_check 1] 48 110 101]
    [p_detector B [G_move_ruleset move 1] 48 110 101]
     ......... (TRUNCATED) .........

** [[activity N Type2 62] [activity N Type1 52]]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
```

The data items in the model are considerably truncated for space reasons. The rule interpreter runs very slowly because of the large amout of data (to speed things up we have used "summary" models in later implementations).

The model is divided into four different sections: `xor_pattern` entries, `activity` entries, `current_mode` entries and `p_detector` entries (they ar not printed in this order by the SIM_AGENT print procedure).

An `xor_pattern` entry is a record of the discovery that a pattern is associated with distinct modes. One `xor_pattern` entry exists for each pattern that belongs to a set of patterns participating in the same transition. The format is:

```
[xor_pattern <agent> <timestamp> <mode> <pattern-instance>]
```

where `agent` is the agent under observation (either N or B in this case), `timestamp` is the time at which the transition was first seen taking place (its discovery timestamp), `mode` is either 1 or 2 meaning the "familiar" (normal) or "unfamiliar" (new) mode respectively and `pattern-instance` is the content of the pattern (in this case a record of whether a rule fired or had its conditions checked).

One `activity` entry exists for each discovery timestamp and points to the sets of patterns that stopped and started respectively at that timestamp. We are assuming here that the transition is crisp and sudden as defined in Chapter 6 (not distributed over several cycles and fussy) and that a newly discovered transition is not "discovered" more than once making the discovery timestamp unsuitable as a unique pointer to an activity. The format is:

```
[activity <agent> <activity-name> <timestamp>]
```

where the activity name is a symbol which is generated (using "gensym") when the activity is discovered and serves to give the new concept a unique symbolic reference. The symbolic naming is mostly "cosmetic" since it is easier to understand a concept that does not rely on a numeric "discovery timestamp" as a unique reference. Moreover the discovery timestamp may cease to exist in a scaled up version where transitions may be more spread out over time).

A `current_mode` entry exists for each activity, indicating which mode of the activity is currently active. For example, N's activity named "Type2" was discovered at cycle 62 and is currently in mode 1. This means that that N's rule *normal* of its *self_repair_ruleset* is expected to fire, because N is not currently responding to an anomaly and therefore its self-repair is not active (its anomaly detection activity is in "normal" mode). Similarly N's activity labelled "Type1" is in mode 1 (because it is in training mode at this stage, because both agents are ready to start the main operational phase.

The `p_detector` entries are the "positive detectors" introduced in Chpater 5, section 6.5.2 and have the following form:

```
[p_detector <agent> <event_record> <last_started> <last_stopped> <weight>]
```

where an event record has the form:

```
[<ruleset> <rule> <status>]
```

If the given rule in the ruleset had its conditions checked then “status” is 0; if the rule fired there is an additional event record with “status” = 1. “Last started” and “last stopped” are timestamps which are updated when the event starts and stops occuring respectively. If the event is currently happening, the “last started” entry will be greater than “last stopped”. The weight is the number of occurrences of the event so far in the total training phase. If the weight is close to 100 % of the total training time, and it stops occurring at a point during operational phase then this indicates an “omission” anomaly (something is abnormally absent). Event records whose weight is not close to the total or close to 0 are not used in this implementation (reasons are given in Section 6.5.1). They are retained for any future enhancement of the anomaly-detection.

B.3 Attack during interim operational phase

Excerpt 3:
Interim operational phase of an agent. During interim operational phase, a number of attacks against the object level of an agent is carried, allowing the neighbouring agent to observe the “normal” anomaly-detection and response patterns of the meta-level.

```
** [N 50 Start of interim operational phase]
** [B 56 Target selected: t4]
** [B 56 Vehicle energy = 445]
** [B 56 Vehicle treasure = 1200]
** [B End of cycle 56 Selected database contents:]
** [[current_target t4 150 -20 treasure]]
** ----------------------------------------------------
** [F 60 ATTACK: Deleted rule no_more of G_external_sensors_ruleset in N]
** [N 61 Omission anomaly in own execution: [G_avoid_obstacles_ruleset avoid_nearest 0]]
** [N 62 Repaired own rule avoid_nearest in G_avoid_obstacles_ruleset]
** [N 65 Omission anomaly in own execution: [G_external_sensors_ruleset no_more 0]]
** [N 65 Omission anomaly in own execution: [G_external_sensors_ruleset no_more 1]]
** [N 65 Omission anomaly in own execution: [G_avoid_obstacles_ruleset avoid_nearest 0]]
** [N 66 Repaired own rule avoid_nearest in G_avoid_obstacles_ruleset]
** [N 66 Repaired own rule no_more in G_external_sensors_ruleset]
** [N 69 Omission anomaly in own execution: [G_avoid_obstacles_ruleset avoid_nearest 0]]
** [B 70 Start of interim operational phase]
** [N 70 Start of immune system training phase 2]
** [N 70 Repaired own rule avoid_nearest in G_avoid_obstacles_ruleset]
** [B 80 Arriving at: t4]
** [B 80 Vehicle energy = 421]
** [B 80 Vehicle treasure = 960]
** [B End of cycle 80 Selected database contents:]
** [[current_target t4 150 -20 treasure]]
** ----------------------------------------------------
** [F 80 ATTACK: Deleted rule happy of G_generate_motive_ruleset in B]
** [B 82 Omission anomaly in own execution: [G_generate_motive_ruleset happy 0]]
** [B 82 Omission anomaly in own execution: [G_generate_motive_ruleset happy 1]]
** [B 83 Repaired own rule happy in G_generate_motive_ruleset]
** [B 90 Start of immune system training phase 2]
** [B 90 Target selected: t5]
** [B 90 Vehicle energy = 411]
** [B 90 Vehicle treasure = 1310]
** [B End of cycle 90 Selected database contents:]
** [[current_target t5 150 -80 treasure]]
** ----------------------------------------------------
** [B 99 Arriving at: t5]
** [B 99 Vehicle energy = 402]
** [B 99 Vehicle treasure = 1220]
** [B End of cycle 99 Selected database contents:]
** [[current_target t5 150 -80 treasure]]
** -------------------------------------------------------
** [B 109 Target selected: t6]
** [B 109 Vehicle energy = 392]
** [B 109 Vehicle treasure = 1570]
** [B End of cycle 109 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -------------------------------------------------------
```

An "omission" anomaly is detected when a normally expected trace record is *absent*. In this implementation the associated component (rule) is immediately "repaired" (a process known as reactive repair). A "repair" involves the restoration of the correct rule into the rulesystem. Note that there are some false-positives. The rule "avoid_nearest" does not have its conditions checked causing it to be "repaired" unnecessarily. The initial training phase (in which an agent becomes familiar with its own object-level patterns) is short in this example run (only 50 cycles), meaning that many normal patterns have not yet been experienced. Most false-positives disappear towards the end of the main training phase.

At cycle 70, the two agents exchange roles.

B.4 Recovery from meta-level attacks

Example 1: attack againt an anomaly-detection component:

```
** -------------------------------------------------------
** [F 640 ATTACK: Deleted rule check_own_omission of
G_immune_operational1_ruleset in N]
** [B 642 Mode omission anomaly : [G_immune_operational1_ruleset
check_own_omission 0]]
** [B 642 Taking over control from  N]
** [B 642 Repaired neighbour rule check_own_omission in
G_immune_operational1_ruleset of N]
** [B 642 Vehicle energy = 235]
** [B 642 Vehicle treasure = 4130]
** [B End of cycle 642 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -------------------------------------------------------
```

Note the attack is against N which currently controls the vehicle. Rule *check_own_omission* is important for detecting anomalies in the agent's own object-level. Agent B detects an anomaly indicating that the condition-checking of rule *check_own_omission* was absent from the operational mode behaviour of N. (Condition-checking is indicated by 0; if a rule is always expected to fire in a mode, there would be an additional anomaly report indicating 1). B then takes over vehicle control and repairs its neighbour (N).

Example 2: Other meta-level attacks:

```
** [F 200 ATTACK: Deleted rule normal_restarting of G_immune_operational1_ruleset in N]
** [B 202 Mode omission anomaly : [G_immune_operational1_ruleset normal_restarting 0]]
** [B 202 Repaired neighbour rule normal_restarting in G_immune_operational1_ruleset of N]
** [B 203 Target selected: t4]
** [B 203 Vehicle energy = 297]
** [B 203 Vehicle treasure = 2420]
** [B End of cycle 203 Selected database contents:]
** [[current_target t4 150 -20 treasure]]
** -------------------------------------------------------
** [F 210 ATTACK: Deleted rule init_false_alarms of G_internal_sensors_ruleset in B]
** [N 211 Omission anomaly: [G_internal_sensors_ruleset init_false_alarms 0]]
** [N 211 Taking over control from  B]
** [N 211 Repaired neighbour rule init_false_alarms in G_internal_sensors_ruleset of B]
** [N 211 Vehicle energy = 288]
** [N 211 Vehicle treasure = 2330]
** [N End of cycle 211 Selected database contents:]
** [[current_target S1 100 45 subtarget]]
** ---------------------------------------------------------
```

The above shows an attack of another rule *normal_restarting* of an anomaly-detection ruleset of N. B acts in the sam eway as for example 1. (The rule is not critical for operation, however. It is concerned with fine-tuning of the model during operational phase). The second attack is against B and involves the internal sensors ruleset (for reading the execution trace). Note that N reacts to the problem in te same way as B, only that it takes control of the vehicle (since B was in control at that point).

Example 3: attack against a repair agent.

```
** [F 170 ATTACK: Deleted rule false_alarm of RA_repair_ruleset in RN]
** [B 172 Arriving at: t2]
** [B 172 Vehicle energy = 328]
** [B 172 Vehicle treasure = 1830]
** [B End of cycle 172 Selected database contents:]
** [[current_target t2 80 -80 treasure]]
** ---------------------------------------------------------
** [B 173 Omission anomaly: [RA_repair_ruleset false_alarm 0]]
** [B 173 Repaired neighbour rule false_alarm in RA_repair_ruleset of N]
```

In this example, N's repair agent (RN) is attacked. B responds to repair N's repair agent in the same way as it would if N itself were attacked.

Example 4: Attack against internal sensors.

```
** [F 880 ATTACK: Deleted rule load_new_trace of G_internal_sensors_ruleset in N]
** [N 881 Omission anomaly in own execution: [clear_data_ruleset forget 0]]
** [N 881 Omission anomaly in own execution: [clear_data_ruleset forget 1]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset check_agent 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset check_vehicle 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset check_vehicle 1]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_energy 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_energy 1]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_ditch 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_ditch 1]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_treasure 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset see_treasure 1]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_external_sensors_ruleset no_more 1]]
** [N 881 Omission anomaly in own execution: [G_monitor_state_ruleset anomaly 0]]
** [N 881 Omission anomaly in own execution: [G_monitor_state_ruleset normal 0]]
** [N 881 Omission anomaly in own execution: [G_monitor_state_ruleset normal 1]]
** [N 881 Omission anomaly in own execution: [G_evaluate_state_ruleset interest_growth 0]]
** [N 881 Omission anomaly in own execution: [G_evaluate_state_ruleset interest_decay 0]]
** [N 881 Omission anomaly in own execution: [G_evaluate_state_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_evaluate_state_ruleset no_more 1]]
** [N 881 Omission anomaly in own execution: [G_focus_attention_ruleset attend_to_interesting_tre
** [N 881 Omission anomaly in own execution: [G_focus_attention_ruleset forget_boring_treasure 0]
** [N 881 Omission anomaly in own execution: [G_focus_attention_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_focus_attention_ruleset no_more 1]]
** [N 881 Omission anomaly in own execution: [G_generate_motive_ruleset hungry 0]]
** [N 881 Omission anomaly in own execution: [G_generate_motive_ruleset bored 0]]
** [N 881 Omission anomaly in own execution: [G_generate_motive_ruleset adventurous 0]]
** [N 881 Omission anomaly in own execution: [G_generate_motive_ruleset happy 0]]
** [N 881 Omission anomaly in own execution: [G_generate_motive_ruleset happy 1]]
** [N 881 Omission anomaly in own execution: [G_select_target_ruleset target_exists 0]]
** [N 881 Omission anomaly in own execution: [G_analyse_ditch_ruleset near_side_of_ditch 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset no_problem 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset target_across_ditch 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset target_at_opposite_end 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset same_side_of_ditch 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_ditch_ruleset no_more 1]]
** [N 881 Omission anomaly in own execution: [G_identify_obstacles_ruleset near_a_ditch 0]]
** [N 881 Omission anomaly in own execution: [G_avoid_obstacles_ruleset path_exists 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset collect_treasure 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset recharge 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset move_from_old_treasure 0]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset move_from_energy 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset move_towards_new_target 0
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset double_check 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_prepare_to_move_ruleset no_more 1]]
** [N 881 Omission anomaly in own execution: [G_move_ruleset move 0]]
** [N 881 Omission anomaly in own execution: [G_move_ruleset finished 0]]
** [N 881 Omission anomaly in own execution: [G_move_ruleset finished 1]]
** [N 881 Omission anomaly in own execution: [G_new_state_ruleset close_to_treasure 0]]
** [N 881 Omission anomaly in own execution: [G_memory_ruleset forget_other_objects 0]]
** [N 881 Omission anomaly in own execution: [G_memory_ruleset forget_other_objects 1]]
** [N 881 Omission anomaly in own execution: [G_memory_ruleset no_more 0]]
** [N 881 Omission anomaly in own execution: [G_memory_ruleset no_more 1]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset init_false_alarms 0]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset check_alertness 0]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset forget_last_trace 0]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset forget_last_trace 1]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset load_new_trace 0]]
** [N 881 Omission anomaly: [G_internal_sensors_ruleset load_new_trace 1]]
** [N 881 Omission anomaly: [G_immune_training_start_ruleset operational_phase 0]]
** [N 881 Omission anomaly: [G_immune_training1_ruleset operational_phase 0]]
** [N 881 Omission anomaly: [G_immune_training2_ruleset in_operational_phase 0]]
** [N 881 Omission anomaly: [G_immune_training3_ruleset in_operational_phase 0]]
** [N 881 Omission anomaly: [G_immune_training_end_ruleset operational_phase 0]]
** [N 881 Omission anomaly: [G_immune_operational1_ruleset in_training_phase 0]]
** [N 881 Omission anomaly: [G_immune_operational2_ruleset in_training_phase 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset run_repair_rules 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset completed_repair 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset abort_repair 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset unnecessary_repair 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset no_more 0]]
** [N 881 Omission anomaly: [G_neighbour_repair_ruleset no_more 1]]
** [N 881 Omission anomaly: [G_self_repair_ruleset normal 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset startup 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset no_repair_required 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset no_repair_required 1]]
** [N 881 Omission anomaly: [RA_repair_ruleset repair_a_rule 0]]
** [N 881 Omission anomaly: [RA_repair_ruleset false_alarm 0]]
** [N 881 Omission anomaly: [RA_repair_ruleset no_more 0]]
** [N 881 Omission anomaly: [RA_repair_ruleset no_more 1]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset repair_calling_agent 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset repair_failed 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset completed 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset no_more 0]]
** [N 881 Omission anomaly: [R_receive_activation_ruleset no_more 1]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset no_more 1]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset no_more 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset check_failure_mode 0]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset normal_stopping 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset no_more 1]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset no_more 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset report_mode_anomaly 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset unexpected_omission 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset expected_omission 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset transition_to_previous_mode 0]]
** [N 881 Mode omission anomaly : [G_immune_operational2_ruleset xor_violation2 0]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset normal_restarting 0]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset check_other_omission 0]]
** [N 881 Mode omission anomaly : [G_immune_operational1_ruleset check_own_omission 0]]
** [N 881 Mode omission anomaly : [G_immune_training_end_ruleset operational_phase 1]]
** [N 881 Mode omission anomaly : [G_immune_training3_ruleset in_operational_phase 1]]
** [N 881 Mode omission anomaly : [G_immune_training2_ruleset in_operational_phase 1]]
** [N 881 Mode omission anomaly : [G_immune_training1_ruleset operational_phase 1]]
** [N 881 Mode omission anomaly : [G_immune_training_start_ruleset operational_phase 1]]
** [N 881 Mode omission anomaly : [G_self_repair_ruleset no_more 1]]
** [N 881 Taking over control from  B]
** [N 881 Aborting repair of  B]
** [N 881 Vehicle energy = 372]
** [N 881 Vehicle treasure = 4640]
** [N End of cycle 881 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
** [B 882 Omission anomaly: [G_internal_sensors_ruleset load_new_trace 0]]
** [B 882 Omission anomaly: [G_internal_sensors_ruleset load_new_trace 1]]
** [B 882 Taking over control from  N]
** [B 882 Repaired neighbour rule load_new_trace in G_internal_sensors_ruleset of N]
** [B 882 Vehicle energy = 371]
** [B 882 Vehicle treasure = 4700]
** [B End of cycle 882 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
```

In the above, the internal sensor update rule *load_new_trace* is deleted, meaning that N "sees nothing" happening internally this cycle. It reacts initially by reporting multiple "omission anomalies" since everything it expects to "see" is suddenly absent. Note that N also believes that B's *load_new_trace* rule needs repairing because its condition-checking and firing were also absent. During its attempt to repair B, N aborts the repair when it detect that its own version of *load_new_trace* is missing, which it should normally copy to B to restore B's version of *load_new_trace*. B then correctly detects that N's *load_new_trace* is not functioning and repairs it.

An alternative version of `damage2.trace` exists which does not depend on identical code as this one does. Instead of copying its own version of a component directly to its neighbour's rulebase during repair, the agent "activates" its neighbour's repair agent, by sending it a standardised "repair request message". The actual code to be copied is held locally by the neighbour's repair agent.

It is redundant to show the traces for the alternative version of the damage-resistant prototype, since its behaviour is approximately the same (it only takes slightly longer because a message must be sent to a service agent). The "repair-agent activation" mechanism is used in all later prototypes (quality evaluation and voting), meaning that they also do not depend on identical code (for example the monitoring might be done in different ways).

Appendix C

Hostility-Resistance Output Traces

For quality monitoring and voting prototypes, an alternative starting configuration shown in Figure C.1 was used (although it actually makes no real difference). Because of the optimistic policy of wait-

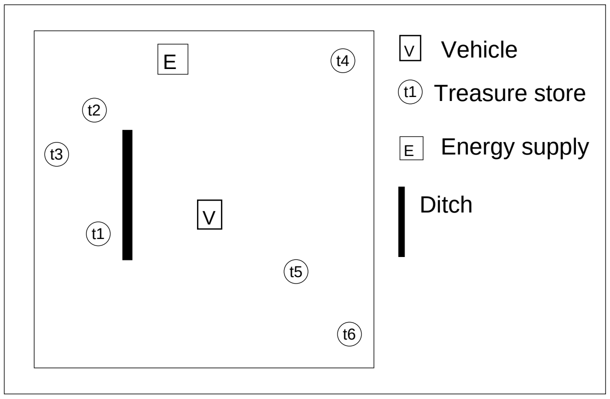


Figure C.1: Treasure Scenario - alternative starting configuration

ing to detect a quality deterioration before intervening, the external environment was made slightly “friendlier”. The decay rate for treasure was decreased from -10 to -2 per cycle.

One of the hostile code scenarios involves modifying the way that the subjective interest in a treasure store decays (in other words the rate at which an agent becomes “bored” while at a treasure store). In the damage-resistant version, the agent can always continue collecting treasure until it loses interest (because there is an infinite amount of “objective” treasure available). To allow for real damage to be done if the subjective interest remains too high, we introduced the concept of “objective-value” of a treasure store. This is the actual amount that can be collected at any time, and is reduced because treasure is actually “removed”. It also has a spontaneous growth rate. Growth and decay rates are as follows:

- Objective value of all treasure stores: increase by 1 on average every 5 cycles, with a maximum value of 500.
- Rate of treasure collection: store value decreases by 50 each cycle; vehicle treasure increases by 50 each cycle.

C.1 Single agent quality monitoring

The following excerpt shows the "quality" model at the end of an agent B's training phase (pattern model is the same as the "damage" versions - and not shown here). The program source file is `hostile1.p` with example trace output in `hostile1.trace`.

Excerpt 1 of hostile1.trace:

```
** [B 500 Vehicle energy = 403]
** [B 500 Vehicle treasure = 6806]
** [B End of cycle 500 Selected database contents:]
** [End of training phase - immune system internal model:]
** [[lowest_energy 190]]
** [[lowest_treasure 4808]]
** [[max_duration treasure_collection t6 9]
    [max_duration target_seeking t6 10]
    [max_duration treasure_collection t5 9]
    [max_duration target_seeking t5 26]
    [max_duration treasure_collection t3 9]
    [max_duration treasure_collection t2 9]
    [max_duration target_seeking t2 11]
    [max_duration treasure_collection t1 9]
    [max_duration target_seeking t1 47]
    [max_duration recharge E 7]
    [max_duration target_seeking E 20]
    [max_duration treasure_collection t4 9]
    [max_duration target_seeking t4 134]
    [max_duration target_seeking t3 4]]
** [[min_interval target_seeking t4 332]
    [min_interval recharge E 108]
    [min_interval treasure_collection t3 256]
    [min_interval target_seeking t3 256]
    [min_interval treasure_collection t2 252]
    [min_interval target_seeking t2 247]
    [min_interval treasure_collection t1 242]
    [min_interval target_seeking t1 227]
    [min_interval treasure_collection t4 107]]
** ------------------------------------------------------
```

C.1.1 Hostility scenario 1 output trace

The following excerpts show scenario 1 of Chapter 7 in which external sensor readings ae falsified.

Excerpt 2 of hostile1.trace.

```
** [B 799 Arrived at energy: E]
** [B 799 Vehicle energy = 256]
** [B 799 Vehicle treasure = 9062]
** [B End of cycle 799 Selected database contents:]
** [[current_target E -50 145 energy]]
** ------------------------------------------------------
** [F 800 ATTACK: Cluster B_external_sensors_ruleset modified in B]
** [B 802 Cluster omission anomaly: B_external_sensors_ruleset]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset check_agent
0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset
check_vehicle 0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset
check_vehicle 1]]
```

```
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset
rogue_see_energy 0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset
rogue_see_energy 1]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset see_ditch
0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset see_ditch
1]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset see_treasure
0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset see_treasure
1]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset no_more 0]]
** [B 802 Unfamiliar pattern: [hostile_B_external_sensors_ruleset no_more 1]]
** [B 806 Target selected: t5]
** [B 806 Vehicle energy = 535]
** [B 806 Vehicle treasure = 9020]
** [B End of cycle 806 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ------------------------------------------------------
```

A hostile version of the external sensors is installed, overwriting the correct one. Agent B immediately detects that the normal rulest is missing, and that unfamiliar patterns exist (for condition-checking and firing of unrecognised rules).

Excerpt 3 of hostile1.trace:

```
** [B 928 Quality alert: low energy level]
** [B 928 Suppressing cluster hostile_B_external_sensors_ruleset]
** [B 929 Quality alert: low energy level]
** [RB 929 Repaired ruleset B_external_sensors_ruleset in agent B]
** [B 930 Quality alert: low energy level]
** [B 930 Energy level low - target retry E]
** [B 930 Vehicle energy = 163]
** [B 930 Vehicle treasure = 9626]
** [B End of cycle 930 Selected database contents:]
** [[current_target E -50 145 energy]]
** [[quality_alert [low_energy 163] 930]]
** ------------------------------------------------------
** [B 931 Quality alert: low energy level]
** [B 932 Quality alert: low energy level]
** [B 933 Quality alert: low energy level]
** [B 934 Quality alert: low energy level]
** [B 935 Quality alert: low energy level]
....
```

Over 100 cycles later, the agent detects a quality alert and corrects the problem. Suppression of the hostile cluster happens at cycle 928 and repair of the correct one at cycle 929.
Note 1: it takes time to detect a quality problem (128 cycles in this case). The lowest energy ever experienced was 190 and it must go below this to be regarded as a quality problem. We can see above that it has dropped to 163.
Note 2: Quality degradations do not disappear instantly following a successful repair. For example, the vehicle energy level remains low until it is recharged successfully. In addition, non-current quality alerts are only gradually forgotten. (Data rollback can help to restore normal quality more quickly, but was not done in this version).

Due to poprulebase technicalities, the forgetting does not necessarily happen in chronological order - but this makes no difference in this case. (If required a chronological forgetting can be enforced simply by making the pattern-matching sensitive to the timestamp on the data item).

Excerpt 4 of hostile1.trace:

```
** [B 1003 Arrived at treasure: t1]
** [B 1003 Vehicle energy = 394]
** [B 1003 Vehicle treasure = 9238]
** [B End of cycle 1003 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** ---------------------------------------------------
** [B 1012 Target selected: t2]
** [B 1012 Vehicle energy = 367]
** [B 1012 Vehicle treasure = 9584]
** [B End of cycle 1012 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** ---------------------------------------------------
```

Approximately 200 cycles after the attack, all quality problems have disappeared and old ones are forgotten.

C.1.2 Hostility scenario 2 output trace

The second hostile code scenario in chapter 7 was run in the later part of the same agent "lifetime" recorded in hostile1.trace.

Excerpt 5 of hostile1.trace:

```
** [B 1192 Target selected: t1]
** [B 1192 Vehicle energy = 527]
** [B 1192 Vehicle treasure = 10754]
** [B End of cycle 1192 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** ---------------------------------------------------
** [F 1200 ATTACK: Cluster B_evaluate_state_ruleset modified in B]
** [B 1202 Cluster omission anomaly: B_evaluate_state_ruleset]
** [B 1202 Unfamiliar pattern: [hostile_B_evaluate_state_ruleset
rogue_evaluate_close_treasure 0]]
** [B 1202 Unfamiliar pattern: [hostile_B_evaluate_state_ruleset
interest_growth 0]]
** [B 1202 Unfamiliar pattern: [hostile_B_evaluate_state_ruleset
interest_growth 1]]
** [B 1202 Unfamiliar pattern: [hostile_B_evaluate_state_ruleset no_more 0]]
** [B 1202 Unfamiliar pattern: [hostile_B_evaluate_state_ruleset no_more 1]]
```

The *evaluate_state* rulest is replaced by a hostile version. As for scenario 1, pattern anomalies are detected immediately.

Excerpt 6:

```
** [B 1266 Suppressing cluster hostile_B_evaluate_state_ruleset]
** [B 1267 Target selected: t1]
** [B 1267 Vehicle energy = 302]
** [B 1267 Vehicle treasure = 10304]
** [B End of cycle 1267 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** ---------------------------------------------------
** [RB 1267 Repaired ruleset B_evaluate_state_ruleset in agent B]
** [B 1268 Quality alert: timeout target_seeking t1]
** [B 1269 Quality alert: timeout target_seeking t1]
** [B 1271 Arrived at treasure: t1]
** [B 1271 Vehicle energy = 290]
** [B 1271 Vehicle treasure = 10330]
** [B End of cycle 1271 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** [[quality_alert [timeout target_seeking t1 79] 1271]]
** ------------------------------------------------------
```

Hostile cluster is suppressed after approximately 20 cycles. Note that quality alerts are intermittent.

C.2 Distributed quality monitoring

The following excerpts are from output file `hostile2.trace` and show the behaviour of the two-agent quality monitoring and recovery. The scenario is described in Chapter 7.

C.2.1 Model on completion of training phase

Excerpt 1 of hostile2.trace:

```
** [End of training phase - immune system internal model:]
** [[activity N Type16 1326 4 4]
    [activity N Type15 1001 2 2]
    [activity N Type14 902 4 4]]
** [[current_mode N 1326 2]
    [current_mode N 1001 2]
    [current_mode N 902 2]]
** [[lowest_energy 182]]
** [[lowest_treasure 4936]]
** [[max_duration target_seeking t1 185]
    [max_duration target_seeking t4 28]
    [max_duration target_seeking E 18]
    [max_duration target_seeking t6 128]
    [max_duration target_seeking t5 141]
    [max_duration recharge E 7]
    [max_duration target_seeking t3 121]
    [max_duration target_seeking t2 130]
    [max_duration treasure_collection t6 3]
    [max_duration treasure_collection t5 3]
    [max_duration treasure_collection t4 3]
    [max_duration treasure_collection t3 3]
    [max_duration treasure_collection t2 3]
    [max_duration treasure_collection t1 3]]
** [[min_interval target_seeking t6 119]
    [min_interval target_seeking t5 119]
    [min_interval treasure_collection t4 119]
    [min_interval target_seeking t2 119]
    [min_interval recharge E 361]
    [min_interval target_seeking E 350]
    [min_interval treasure_collection t1 118]
    [min_interval target_seeking t1 118]
    [min_interval treasure_collection t6 118]
    [min_interval treasure_collection t5 119]
    [min_interval target_seeking t3 120]
    [min_interval target_seeking t4 114]
    [min_interval treasure_collection t3 118]
    [min_interval treasure_collection t2 119]]
** [[p_ruleset B clear_data_ruleset 1518 3 -1]
    [p_ruleset B R_receive_activation_ruleset 1518 3 -1]
    [p_ruleset B G_external_sensors_ruleset 1519 2 -1]
    [p_ruleset B G_monitor_state_ruleset 1519 2 -1]
    [p_ruleset B G_evaluate_state_ruleset 1519 2 -1]
    [p_ruleset B G_generate_motive_ruleset 1519 2 -1]
    [p_ruleset B G_select_target_ruleset 1519 2 -1]
    [p_ruleset B G_analyse_ditch_ruleset 1519 2 -1]
    [p_ruleset B G_avoid_ditch_ruleset 1519 2 -1]
    [p_ruleset B G_identify_obstacles_ruleset 1519 2 -1]
    [p_ruleset B G_avoid_obstacles_ruleset 1519 2 -1]
    [p_ruleset B G_init_stats_ruleset 1519 2 -1]
    [p_ruleset B G_collect_stats_ruleset 1519 2 -1]
    [p_ruleset B G_prepare_to_move_ruleset 1519 2 -1]
    [p_ruleset B G_last_minute_correction_ruleset 1519 2 -1]
    [p_ruleset B G_move_ruleset 1519 2 -1]
    [p_ruleset B G_new_state_ruleset 1519 2 -1]
    [p_ruleset B G_receive_activation_ruleset 1519 2 -1]
    [p_ruleset B G_memory_ruleset 1519 2 -1]
    [p_ruleset N G_internal_sensors_ruleset 1519 2 -1]
    [p_ruleset N G_immune_training1_ruleset 1019 2000 1001]
    [p_ruleset N G_immune_training2_ruleset 1019 2000 1001]
    [p_ruleset N G_quality_operational1_ruleset 619 902 -1]
    [p_ruleset N G_quality_operational2_ruleset 619 902 -1]
    [p_ruleset N G_quality_operational3_ruleset 619 902 -1]
    [p_ruleset N G_quality_operational4_ruleset 619 902 -1]
    [p_ruleset N G_wait1_ruleset 1505 1339 1326]
    [p_ruleset N G_wait2_ruleset 1505 1339 1326]
    [p_ruleset N G_wait4_ruleset 1505 1339 1326]
    [p_ruleset N G_wait3_ruleset 1505 1339 1326]
    [p_ruleset N G_immune_operational2_ruleset 499 1001 2000]
    [p_ruleset N G_immune_operational1_ruleset 499 1001 2000]
    [p_ruleset N G_recover_data_ruleset 13 1326 1339]
    [p_ruleset N G_self_repair_ruleset 13 1326 1339]
    [p_ruleset N G_suppress_execution_ruleset 13 1326 1339]
    [p_ruleset N G_diagnose_ruleset 13 1326 1339]
    [p_ruleset N G_quality_training4_ruleset 900 2 902]
    [p_ruleset N G_quality_training3_ruleset 900 2 902]
    [p_ruleset N G_quality_training2_ruleset 900 2 902]
    [p_ruleset N G_quality_training1_ruleset 900 2 902]]
** [[xor_ruleset N 1326 2 G_recover_data_ruleset]
    [xor_ruleset N 1326 2 G_self_repair_ruleset]
    [xor_ruleset N 1326 2 G_suppress_execution_ruleset]
    [xor_ruleset N 1326 2 G_diagnose_ruleset]
    [xor_ruleset N 1326 1 G_wait1_ruleset]
    [xor_ruleset N 1326 1 G_wait2_ruleset]
    [xor_ruleset N 1326 1 G_wait4_ruleset]
    [xor_ruleset N 1326 1 G_wait3_ruleset]
    [xor_ruleset N 1001 2 G_immune_operational2_ruleset]
    [xor_ruleset N 1001 2 G_immune_operational1_ruleset]
    [xor_ruleset N 1001 1 G_immune_training1_ruleset]
    [xor_ruleset N 1001 1 G_immune_training2_ruleset]
    [xor_ruleset N 902 2 G_quality_operational4_ruleset]
    [xor_ruleset N 902 2 G_quality_operational3_ruleset]
    [xor_ruleset N 902 2 G_quality_operational2_ruleset]
    [xor_ruleset N 902 2 G_quality_operational1_ruleset]
    [xor_ruleset N 902 1 G_quality_training1_ruleset]
    [xor_ruleset N 902 1 G_quality_training2_ruleset]
    [xor_ruleset N 902 1 G_quality_training3_ruleset]
    [xor_ruleset N 902 1 G_quality_training4_ruleset]]
```

The pattern model is the same as for the damage-resistance version, except that the model is at the "summary" level. An `xor_ruleset` entry has the same function and format as an `xor_pattern` entry except that the `pattern-instance` field is merely the name of a component (ruleset) and the whole entry states that this component is expected to be active in the stated mode of the activity (indicated by discovery timestamp).

Similarly a `p_ruleset` entry indicates that the given ruleset is expected to be active with the given frequency (weight). As for the `p-detector` entries, detectors with weights not close to total training time (equivalent to 100 %) are not used in these proptotypes, but are left in the database for use in an extended version which might use statistical techniques.

C.2.2 Attack during interim operational phase

Excerpt 2 of hostile2.trace:

```
** -----------------------------------------------------
** [F 1250 ATTACK: Cluster G_evaluate_state_ruleset modified in  N]
** [N 1252 Pattern anomaly: inactive component G_evaluate_state_ruleset]
** [N 1252 Pattern anomaly: unfamiliar component hostile_G_evaluate_state_ruleset]
** [N 1252 Vehicle energy = 299]
** [N 1252 Vehicle treasure = 5598]
** [N End of cycle 1252 Selected database contents:]
** [[current_target S1 -8 -75 subtarget]]
** -----------------------------------------------------
** [B 1278 Arrived at treasure: t1]
** [N 1278 Arrived at treasure: t1]
** [N 1278 Vehicle energy = 273]
** [N 1278 Vehicle treasure = 5546]
** [N End of cycle 1278 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** -----------------------------------------------------
** [B 1279 Target selected: t2]
** [N 1279 Target selected: t2]
** [N 1279 Vehicle energy = 272]
** [N 1279 Vehicle treasure = 5594]
** [N End of cycle 1279 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** -----------------------------------------------------
```

This is an execution of hostile scenario 2 in an "exercise" attack, the purpose being to acquire a model of the normal response to a hostile attack. (In a more relaistic situation, multiple types of hostile attack would be run at this stage, not just one).

As with previous traces, only the database contents of the primary agent are shown. Note that agents make identical decisions. Note also that they are sychronised. If the primary agent (N) selects a target, its backup also selects a target during the same scheduler time-slice. When N believes it is arriving at a target, B also believes it would be arriving at the same target if it were controlling the vehicle (it executes the same code "internally"). This precise synchronisation and agreement has the advantage that the backup can detect an "external world" anomaly if the vehicle is not in the position it expects it to be in. This is interpreted as a quality deterioration because the agents are "required" to be in agreement (to make it easier for one to take over control from the other). However it also has serious disadvantages which are discussed in Chpater 11, section 12.3.

Note that the pattern-anomaly detection is now very "coarse" and on the level of a whole component (ruleset) being active or not.

C.2.3 Hostility-resistance in interim operational phase

Excerpt 3: Some time later, N detects a quality degradation in its own operation and responds.

```
** [B 1324 Quality alert: unexpected vehicle position]
** [B 1324 Vehicle energy = 227]
** [B 1324 Vehicle treasure = 5654]
** [B End of cycle 1324 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------
** [N 1324 Target selected: t3]
** [N 1324 Quality alert: possible infinite loop relating to target_seeking t3]
** [N 1324 Minimum observed interval= 120 , this interval= 33]
** [N 1324 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [N 1324 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [N 1324 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [N 1324 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [N 1324 Vehicle energy = 227]
** [N 1324 Vehicle treasure = 5654]
** [N End of cycle 1324 Selected database contents:]
** [[current_target t3 -120 50 treasure]]
** [[quality_alert [possible_loop target_seeking t3] 1324]]
** ----------------------------------------------------
** [B 1325 Quality alert: unexpected vehicle position]
** [B 1325 Vehicle energy = 226]
** [B 1325 Vehicle treasure = 5652]
** [B End of cycle 1325 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------
** [N 1325 Suppressing hostile_G_evaluate_state_ruleset of N]
** [N 1325 Activating repair of: N G_evaluate_state_ruleset]
** [B 1326 Vehicle energy = 225]
** [B 1326 Vehicle treasure = 5650]
** [B End of cycle 1326 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------
** [RN 1326 Repaired N G_evaluate_state_ruleset]
** [B 1327 Vehicle energy = 224]
** [B 1327 Vehicle treasure = 5648]
** [B End of cycle 1327 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------
** [N 1327 Rolling back data to trusted state]
** [N 1327 Vehicle energy = 224]
** [N 1327 Vehicle treasure = 5648]
** [N End of cycle 1327 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** [[quality_alert [possible_loop target_seeking t3] 1324]]
** ----------------------------------------------------
** [B 1334 Arrived at treasure: t4]
** [B 1334 Target selected: t5]
** [N 1334 Arrived at treasure: t4]
** [N 1334 Target selected: t5]
** [N 1334 Vehicle energy = 217]
** [N 1334 Vehicle treasure = 5634]
** [N End of cycle 1334 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** [[quality_alert [possible_loop target_seeking t3] 1324]]
** ----------------------------------------------------
** [N 1337 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [N 1337 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [N 1337 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [N 1337 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [B 1351 Energy level low - interrupting path to seek target E]
** [N 1351 Energy level low - interrupting path to seek target E]
** [N 1351 Vehicle energy = 200]
** [N 1351 Vehicle treasure = 5600]
** [N End of cycle 1351 Selected database contents:]
** [[current_target E -50 145 energy]]
** ----------------------------------------------------------
```

The time taken to recover (from attack to data rollback) is approx 70 cycles. Both agents are again synchronised after the data rollback. Note that B also detects a quality problem in the form of a vehicle position anomaly.

In this version the different modes are associated with "rule-familes" in POPRULEBASE. There are collections of related rules, only one of which is "current" (i.e. it is the one that currently runs). Switching from training to operational mode is done when the current ruleset (associated with training) "hands over" control to an alternative ruleset in the same rulefamily which runs during operational mode. In this version, an "activity" is represented implicitly by a set of rulefamilies concerned with pattern- and quality monitoring. Each rulefamily contains two mutually exclusive rulesets, one of which is active in mode 1, the other in mode 2. This is one way in which software can be designed so that it is easy to build a model of it's behaviour. Looking at the above trace, a transition is easily detectable when all training mode ruleset "switch" control to their associated operational mode rulesets.

However, at this summary level, the model-building does not really discover anything that is not known in advance; the model merely represents it in a different way.

C.3 Multiple simultaneous meta-level attacks

Both agents are now in main operational phase, the following excerpts are from the remainder of `hostile2.trace` and show recovery from a challenging scenario involving multiple simultaneous attacks.

Excerpt 4: Multiple simultaneous meta-level attacks on B and initial pattern anomaly-detection by N:

```
** [F 2250 ATTACK: Cluster G_quality_monitoring4_rulefamily modified in  B]
** [F 2250 ATTACK: Cluster G_quality_monitoring3_rulefamily modified in  B]
** [F 2250 ATTACK: Cluster G_quality_monitoring2_rulefamily modified in  B]
** [F 2250 ATTACK: Cluster G_quality_monitoring1_rulefamily modified in  B]
** [F 2250 ATTACK: Cluster G_pattern_monitoring2_rulefamily modified in  B]
** [F 2250 ATTACK: Cluster G_pattern_monitoring1_rulefamily modified in  B]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_quality_operational4_ruleset]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_quality_operational3_ruleset]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_quality_operational2_ruleset]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_quality_operational1_ruleset]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_immune_operational2_ruleset]
** [N 2251 Pattern anomaly: unfamiliar component dummy_G_immune_operational1_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_immune_operational2_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_immune_operational1_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_quality_operational4_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_quality_operational3_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_quality_operational2_ruleset]
** [N 2251 Mode pattern anomaly: inactive component: G_quality_operational1_ruleset]
** [N 2251 Vehicle energy = 350]
** [N 2251 Vehicle treasure = 5600]
** [N End of cycle 2251 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------------
** [B 2272 Arrived at treasure: t5]
** [B 2272 Vehicle energy = 329]
** [B 2272 Vehicle treasure = 5558]
** [B End of cycle 2272 Selected database contents:]
** [[current_target t5 100 -80 treasure]]
** ----------------------------------------------------------
```

Because of optimistic policy, N does nothing after detecting pattern anomalies. Instead it waits to detect any quality problem in the external world.

C.3.1 Attack on an agent whose monitoring is disabled

Excerpt 5: attack against B with no response.

```
** [F 2500 ATTACK: Cluster G_evaluate_state_ruleset modified in  B]
** [B 2514 Arrived at treasure: t1]
** [B 2514 Vehicle energy = 437]
** [B 2514 Vehicle treasure = 5424]
** [B End of cycle 2514 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** ----------------------------------------------------
** [N 2514 Arrived at treasure: t1]
** [B 2515 Target selected: t2]
** [B 2515 Vehicle energy = 436]
** [B 2515 Vehicle treasure = 5472]
** [B End of cycle 2515 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** ----------------------------------------------------
** [N 2515 Target selected: t2]
** [B 2526 Arrived at treasure: t2]
** [B 2526 Vehicle energy = 425]
** [B 2526 Vehicle treasure = 5450]
** [B End of cycle 2526 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** ----------------------------------------------------
```

B does not detect an anomaly or quality problem because its monitoring components have been disabled. N does not detect any pattern anomaly because it has no access to B's object-level.

C.3.2 Hostility detection and response by neighbour

Excerpt 6:

```
** [N 2787 Quality alert: unexpected vehicle position]
** [N 2787 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [N 2787 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [N 2787 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [N 2787 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [N 2787 Vehicle energy = 514]
** [N 2787 Vehicle treasure = 5228]
** [N End of cycle 2787 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2787]]
** ----------------------------------------------------
** [N 2788 Quality alert: unexpected vehicle position]
** [N 2788 Suppressing dummy_G_immune_operational1_ruleset of B]
** [N 2788 Suppressing dummy_G_immune_operational2_ruleset of B]
** [N 2788 Suppressing dummy_G_quality_operational1_ruleset of B]
** [N 2788 Suppressing dummy_G_quality_operational2_ruleset of B]
** [N 2788 Suppressing dummy_G_quality_operational3_ruleset of B]
** [N 2788 Suppressing dummy_G_quality_operational4_ruleset of B]
** [N 2788 Activating repair of: B G_quality_operational1_ruleset]
** [N 2788 Activating repair of: B G_quality_operational2_ruleset]
** [N 2788 Activating repair of: B G_quality_operational3_ruleset]
** [N 2788 Activating repair of: B G_quality_operational4_ruleset]
** [N 2788 Activating repair of: B G_immune_operational1_ruleset]
** [N 2788 Activating repair of: B G_immune_operational2_ruleset]
** [N 2788 Vehicle energy = 513]
** [N 2788 Vehicle treasure = 5226]
** [N End of cycle 2788 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2788]]
** ----------------------------------------------------
** [RB 2789 Repaired B G_pattern_monitoring2_rulefamily]
** [N 2789 Vehicle energy = 512]
** [N 2789 Vehicle treasure = 5224]
** [N End of cycle 2789 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2789]]
** ----------------------------------------------------
** [RB 2790 Repaired B G_pattern_monitoring1_rulefamily]
** [N 2790 Vehicle energy = 511]
** [N 2790 Vehicle treasure = 5222]
** [N End of cycle 2790 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2790]]
** ----------------------------------------------------
** [B 2791 Switching from G_immune_training1_ruleset to G_immune_operational1_ruleset]
** [RB 2791 Repaired B G_quality_monitoring4_rulefamily]
** [N 2791 Vehicle energy = 510]
** [N 2791 Vehicle treasure = 5220]
** [N End of cycle 2791 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2791]]
** ----------------------------------------------------
** [B 2792 Switching from G_quality_training4_ruleset to G_quality_operational4_ruleset]
** [RB 2792 Repaired B G_quality_monitoring3_rulefamily]
** [N 2792 Quality alert: unexpected vehicle position]
** [N 2792 Vehicle energy = 509]
** [N 2792 Vehicle treasure = 5218]
** [N End of cycle 2792 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2792]]
** ----------------------------------------------------
** [B 2793 Pattern anomaly: inactive component G_evaluate_state_ruleset]
** [B 2793 Pattern anomaly: unfamiliar component hostile_G_evaluate_state_ruleset]
** [B 2793 Switching from G_quality_training3_ruleset to G_quality_operational3_ruleset]
** [B 2793 Vehicle energy = 508]
** [B 2793 Vehicle treasure = 5216]
** [B End of cycle 2793 Selected database contents:]
** ----------------------------------------------------
** [RB 2793 Repaired B G_quality_monitoring2_rulefamily]
** [N 2793 Quality alert: unexpected vehicle position]
** [N 2793 Vehicle energy = 508]
** [N 2793 Vehicle treasure = 5216]
** [N End of cycle 2793 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2793]]
** ----------------------------------------------------
** [B 2794 Switching from G_quality_training2_ruleset to G_quality_operational2_ruleset]
** [RB 2794 Repaired B G_quality_monitoring1_rulefamily]
** [N 2794 Quality alert: unexpected vehicle position]
** [N 2794 Vehicle energy = 507]
** [N 2794 Vehicle treasure = 5214]
** [N End of cycle 2794 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** [[quality_alert [position vehicle] 2794]]
** ----------------------------------------------------
** [B 2795 Switching from G_quality_training1_ruleset to G_quality_operational1_ruleset]
** [N 2795 Quality alert: unexpected vehicle position]
** [N 2795 Rolling back data to trusted state]
** [N 2795 Vehicle energy = 506]
** [N 2795 Vehicle treasure = 5212]
** [N End of cycle 2795 Selected database contents:]
** [[current_target t1 -80 0 treasure]]
** [[quality_alert [position vehicle] 2795]]
** ------------------------------------------------------
```

After more than 200 cycles, N detects a quality problem and intervenes to repair B's meta-level. It takes longer to detect a quality problem in a neighbour than it would in its own performance. It cannot detect a timeout or possible loop because an agent only measures these things in its own internal performance. It can only detect an unexpected vehicle position. If there is an internal problem in B its external decisions on moving the vehicle will be affected.

Here the repair agent RB must repair a large number of components. In this version, it only repairs one component per cycle (making it closer to a real-life situation). Note that the monitoring components of B gradually "come to life" after they have been repaired. They switch control to their operational-mode rulesets because the default "current-ruleset" is set to the training mode ruleset.

C.3.3 Recovery of an agent's self-repair capability

Excerpt 7:

```
** [B 2862 Target selected: t2]
** [B 2862 Quality alert: possible infinite loop relating to target_seeking t2]
** [B 2862 Minimum observed interval= 119 , this interval= 20]
** [B 2862 Suppressing hostile_G_evaluate_state_ruleset of B]
** [B 2862 Activating repair of: B G_evaluate_state_ruleset]
** [B 2862 Vehicle energy = 439]
** [B 2862 Vehicle treasure = 5428]
** [B End of cycle 2862 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** [[quality_alert [possible_loop target_seeking t2] 2862]]
** ------------------------------------------------------
** [N 2862 Target selected: t4]
** [RB 2863 Repaired B G_evaluate_state_ruleset]
** [B 2864 Arrived at treasure: t2]
** [B 2864 Quality alert: possible infinite loop relating to treasure_collection t2]
** [B 2864 Minimum observed interval= 119 , this interval= 11]
** [B 2864 Rolling back data to trusted state]
** [B 2864 Vehicle energy = 437]
** [B 2864 Vehicle treasure = 5424]
** [B End of cycle 2864 Selected database contents:]
** [[quality_alert [possible_loop treasure_collection t2] 2864]
    [quality_alert [possible_loop target_seeking t2] 2862]]
** ------------------------------------------------------
```

B finally detects a quality problem in its own operation and repairs the problem in its object-level. Total recovery time (from object-level attack to data rollback) is 300 cycles. This is considerably longer than when the monitoring components are intact.

Note that vehicle treasure level continues to fall in the 300 cycles from object-level attack to recovery. Quality alerts continued to appear intermittently for several hundred cycles, meaning that it takes time for treasure levels to become "normal" again.

Excerpt 8: Example of correct functioning some time after recovery (no further quality alerts):

```
** [B 3365 Target selected: t6]
** [B 3365 Vehicle energy = 286]
** [B 3365 Vehicle treasure = 5472]
** [B End of cycle 3365 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
** [N 3365 Target selected: t6]
** [B 3407 Arrived at treasure: t6]
** [B 3407 Vehicle energy = 244]
** [B 3407 Vehicle treasure = 5388]
** [B End of cycle 3407 Selected database contents:]
** [[current_target t6 120 -150 treasure]]
** -----------------------------------------------------
** [N 3407 Arrived at treasure: t6]
** [B 3408 Target selected: t4]
** [B 3408 Vehicle energy = 243]
** [B 3408 Vehicle treasure = 5436]
** [B End of cycle 3408 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** -----------------------------------------------------
** [N 3408 Target selected: t4]
** [B 3437 Arrived at treasure: t4]
** [B 3437 Target selected: t2]
** [B 3437 Vehicle energy = 214]
** [B 3437 Vehicle treasure = 5378]
** [B End of cycle 3437 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** -----------------------------------------------------
```

Appendix D

Voting Output Traces

This appendix contains output traces for voting in two different intrusion scenarios. The first is a version of scenario 1(b) described in 9.6; the second is a version scenario 3 in 9.7. For simplicity the voting prototype does not use an "optimistic" policy, but instead takes action as soon as a pattern-anomaly is detected (in this case the action involves starting the voting procedure).

D.1 Normal operation of three agents

Excerpt 1 of voting1.trace: Example of normal operation for 3 agents.

```
** [B 36 Target selected: t2]
** [P 36 Target selected: t2]
** [N 36 Target selected: t2]
** [N End of cycle 36 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** -----------------------------------------------------
** -----------------------------------------------------
** [B 42 Arrived at treasure: t2]
** [P 42 Arrived at treasure: t2]
** [N 42 Arrived at treasure: t2]
** [N End of cycle 42 Selected database contents:]
** [[current_target t2 -80 80 treasure]]
** -----------------------------------------------------
** -----------------------------------------------------
** [B 45 Target selected: t3]
** [P 45 Target selected: t3]
** [N 45 Target selected: t3]
** [N End of cycle 45 Selected database contents:]
** [[current_target t3 -120 50 treasure]]
** -----------------------------------------------------
** -----------------------------------------------------
```

As in the two-agent quality monitoring system, all three agents are synchronised in this version.

For additional clarity in the three-agent case, two lines are drawn following each state-change report. One indicates the end of an *agent-cycle* after the agent's database contents have been printed (an agent-cycle records the number of times the agent's rulesystem has been run so far). The other line marks the end of a scheduler *time-slice* (often also called a "cycle" when it cannot be confused with an agent-cycle). In the above excerpt, both these boundaries coincide because only one agent's database contents are being printed and this is the agent- cycle that ends just as the current scheduler time-slice is ending (it is the last one of the three agents to be run). In later excerpts there may be differences between the two, in particular during a voting procedure where a more detailed trace is printed for all agents.

D.2 Internal model on completion of training phase

Excerpt 2:

```
** [Operational mode begins next cycle]
** [B 1200 Switching from G_immune_training1_ruleset to G_immune_operational1_ruleset]
** [B 1200 Switching from G_immune_training2_ruleset to G_immune_operational2_ruleset]
** [B 1200 Arrived at treasure: t4]
** [B 1200 Target selected: t5]
** [B End of cycle 1200 Selected database contents:]
** [End of training phase - immune system internal model:]
** [[activity P Type12 904 7 7]
    [activity P Type10 901 2 2]
    [activity N Type3  404 7 7]
    [activity N Type1  301 2 2]]
** [[current_mode P 904 2]
    [current_mode P 901 2]
    [current_mode N 404 2]
    [current_mode N 301 2]]
** [[p_ruleset B clear_data_ruleset 998 3 -1]
    [p_ruleset B G_external_sensors_ruleset 999 2 -1]
    [p_ruleset B G_monitor_state_ruleset 999 2 -1]
    [p_ruleset B G_evaluate_state_ruleset 999 2 -1]
..... (TRUNCATED)......

    [p_ruleset N G_internal_sensors_ruleset 999 2 -1]
    [p_ruleset N G_immune_training1_ruleset 798 501 301]
    [p_ruleset N G_immune_training2_ruleset 798 501 301]
..... (TRUNCATED)......

    [p_ruleset P G_internal_sensors_ruleset 999 2 -1]
    [p_ruleset P G_immune_training1_ruleset 798 1101 901]
    [p_ruleset P G_immune_training2_ruleset 798 1101 901]
..... (TRUNCATED)......

** [[xor_ruleset P 904 2 G_recover_data_ruleset]
    [xor_ruleset P 904 2 G_self_repair_ruleset]
    [xor_ruleset P 904 2 G_suppress_execution_ruleset]
    [xor_ruleset P 904 2 G_diagnose_ruleset]
    [xor_ruleset P 904 2 G_inhibit_neighbours_ruleset]
    [xor_ruleset P 904 2 G_wait_for_majority_ruleset]
    [xor_ruleset P 904 2 G_broadcast_vote_ruleset]
    [xor_ruleset P 904 1 G_no_anomaly1_ruleset]
    [xor_ruleset P 904 1 G_no_anomaly2_ruleset]
    [xor_ruleset P 904 1 G_no_anomaly3_ruleset]
    [xor_ruleset P 904 1 G_wait1_ruleset]
    [xor_ruleset P 904 1 G_wait2_ruleset]
    [xor_ruleset P 904 1 G_wait4_ruleset]
    [xor_ruleset P 904 1 G_wait3_ruleset]
    [xor_ruleset P 901 2 G_immune_operational2_ruleset]
    [xor_ruleset P 901 2 G_immune_operational1_ruleset]
    [xor_ruleset P 901 1 G_immune_training1_ruleset]
    [xor_ruleset P 901 1 G_immune_training2_ruleset]
    [xor_ruleset N 404 2 G_recover_data_ruleset]
    [xor_ruleset N 404 2 G_self_repair_ruleset]
    [xor_ruleset N 404 2 G_suppress_execution_ruleset]
    [xor_ruleset N 404 2 G_diagnose_ruleset]
    [xor_ruleset N 404 2 G_inhibit_neighbours_ruleset]
    [xor_ruleset N 404 2 G_wait_for_majority_ruleset]
    [xor_ruleset N 404 2 G_broadcast_vote_ruleset]
    [xor_ruleset N 404 1 G_no_anomaly1_ruleset]
    [xor_ruleset N 404 1 G_no_anomaly2_ruleset]
    [xor_ruleset N 404 1 G_no_anomaly3_ruleset]
    [xor_ruleset N 404 1 G_wait1_ruleset]
    [xor_ruleset N 404 1 G_wait2_ruleset]
    [xor_ruleset N 404 1 G_wait4_ruleset]
    [xor_ruleset N 404 1 G_wait3_ruleset]
    [xor_ruleset N 301 2 G_immune_operational2_ruleset]
    [xor_ruleset N 301 2 G_immune_operational1_ruleset]
    [xor_ruleset N 301 1 G_immune_training1_ruleset]
    [xor_ruleset N 301 1 G_immune_training2_ruleset]]
** ------------------------------------------------------
** ------------------------------------------------------
```

The above shows the main sections of agent B's internal model. Truncating is necessary to save space. We can see that B has a model of its own object-level rulesets and the meta-level rulesets of its two neighbours (both are identical). N and P have the same kind og internal model.

D.3 Meta-level hostile code scenario

This scenario is a version of scenario 1(b) in Chapter 8, section 9.6 where an agent P (controlling the vehicle) has hostile code inserted into its meta-level. We designed a version of this scenario where a voting component is replaced by a rogue version (hostile_G_broadcast_vote_ruleset). Pseudocode of the rogue version is given in Chapter 8, section 9.6.1. The following are excerpts from "voting1.trace" which records a run of this scenario.

Excerpt 1: Attack by F on P and initial (false) response by P:

```
** [F 1300 ATTACK: Cluster G_voting1_rulefamily modified in P]
** [P broadcasting vote 1 regarding B]
** [P broadcasting vote 1 regarding N]
** [P 1301 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [P 1301 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [P 1301 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [P 1301 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [P 1301 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [P 1301 Switching from G_wait3_ruleset to G_recover_data_ruleset]
```

Immediately P "lies" about P and N by broadcasting a a vote 1 concerning them.

Excerpt 2:

```
** [N 1301 Pattern anomaly: unfamiliar component hostile_G_broadcast_vote_ruleset]
** [N 1301 Pattern anomaly: inactive component G_no_anomaly1_ruleset in P]
** [N 1301 Switching from G_no_anomaly1_ruleset to G_broadcast_vote_ruleset]
** [N 1301 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [N 1301 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [N 1301 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [N 1301 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [N 1301 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [N 1301 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [N End of cycle 1301 Selected database contents:]
** [[anomaly mode_omission cluster P G_no_anomaly1_ruleset 1301]
    [anomaly unfamiliar cluster P hostile_G_broadcast_vote_ruleset 1301]]
** [[current_target t4 150 150 treasure]]
** [[vote P B 1 1301] [vote P N 1 1301]]
** ------------------------------------------------------
** ------------------------------------------------------
** [B 1302 Pattern anomaly: unfamiliar component hostile_G_broadcast_vote_ruleset]
** [B 1302 Pattern anomaly: inactive component G_no_anomaly1_ruleset in P]
** [B 1302 Switching from G_no_anomaly1_ruleset to G_broadcast_vote_ruleset]
** [B 1302 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [B 1302 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [B 1302 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [B 1302 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [B 1302 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [B 1302 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [B End of cycle 1302 Selected database contents:]
** [[anomaly mode_omission cluster P G_no_anomaly1_ruleset 1302]
    [anomaly unfamiliar cluster P hostile_G_broadcast_vote_ruleset 1302]]
** [[current_target t4 150 150 treasure]]
** [[vote P B 1 1301] [vote P N 1 1301]]
** ------------------------------------------------------
```

Both N and B detect an anomaly in P's meta-level and switch to anomaly-response mode.

Excerpt 3:

```
** [N 1302 Context-sensitive pattern anomaly: inactive component: G_broadcast_vote_ruleset]
** [N broadcasting vote 0 regarding B]
** [N broadcasting vote 1 regarding P]
** [N End of cycle 1302 Selected database contents:]
** [[anomaly mode_omission cluster P G_broadcast_vote_ruleset 1302]
    [anomaly mode_omission cluster P G_no_anomaly1_ruleset 1301]
    [anomaly unfamiliar cluster P hostile_G_broadcast_vote_ruleset 1301]]
** [[broadcasted P 1302] [broadcasted B 1302]]
** [[current_target t4 150 150 treasure]]
** [[vote N B 0 1302]
    [vote N P 1 1302]
    [vote P B 1 1301]
    [vote P N 1 1301]]
** [[vote_awaited 1302]]
** ------------------------------------------------------
** ------------------------------------------------------
** [B 1303 Context-sensitive pattern anomaly: inactive component: G_broadcast_vote_ruleset]
** [B broadcasting vote 0 regarding N]
** [B broadcasting vote 1 regarding P]
** [B Responding to anomaly in agent P]
** [B 1303 Inhibiting response of P]
** [B 1303 Inhibiting response of N]
** [B 1303 Suppressing hostile_G_broadcast_vote_ruleset of P]
** [B 1303 Activating repair of: P G_broadcast_vote_ruleset]
** [B 1303 Activating repair of: P G_no_anomaly1_ruleset]
** [B End of cycle 1303 Selected database contents:]
** [[broadcasted P 1303] [broadcasted N 1303]]
** [[current_target t4 150 150 treasure]]
** [[responding P 1303]]
** [[vote B N 0 1303]
    [vote N P 1 1302]
    [vote N B 0 1302]
    [vote P B 1 1301]
    [vote P N 1 1301]]
** ------------------------------------------------------
```

N broadcasts its vote, followed by B. B finds that a threshold number of its neighbours agree with its vote on P (in this case the threshold is just one other agent). B then responds by inhibiting its neighbours, suppressing the rogue component and repairing the correct ones.

The $[vote...]$ entries record what has been broadcasted by all agents so far. (As soon as the threshold is achieved, there is no longer any need to record votes - it simply responds immediately - this is why only one of B’s votes are in its database).

Excerpt 4:

```
** [B 1306 Context-sensitive pattern anomaly: inactive component: G_broadcast_vote_ruleset]
** [B 1306 Rolling back data to trusted state]
** [B End of cycle 1306 Selected database contents:]
** [[broadcasted P 1303] [broadcasted N 1303]]
** [[current_target t4 150 150 treasure]]
** [[recovered 1306]]
** [[responding P 1303]]
** ------------------------------------------------------
** [B 1311 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [B 1311 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [B 1311 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [B 1311 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [B 1311 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [B 1311 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [B 1311 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [P 1311 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [P 1311 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [P 1311 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [P 1311 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [P 1311 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [P 1311 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [P 1311 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [N 1311 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [N 1311 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [N 1311 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [N 1311 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [N 1311 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [N 1311 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [N 1311 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [B 1319 Arrived at treasure: t4]
** [P 1319 Arrived at treasure: t4]
** [P End of cycle 1319 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** ------------------------------------------------------
** [N 1319 Arrived at treasure: t4]
```

A couple of cycles later, B rolls back the data to a previous trusted state. All agents revert back to “normal” (no anomaly) mode. At cycle 1319, all agents have resumed normal operation. Whenever database contents are printed at the end of an agent cycle, a line is drawn under it. Because P is in control of the vehicle, only its database contents are shown.

D.4 Simultaneous attacks on two agents

The following excerpts are of `voting2.trace` produced by source file `voting2.p` and show what happens when two agents have their *broadcast_vote* rulesets simultaneously deleted.

Excerpt 1:

```
** [F 1300 ATTACK: Cluster G_voting1_rulefamily modified in P )]
** [F 1300 ATTACK: Cluster G_voting1_rulefamily modified in N )]
** [N 1301 Pattern anomaly: unfamiliar component dummy_G_no_anomaly1_ruleset]
** [N 1301 Pattern anomaly: inactive component G_no_anomaly1_ruleset in P]
** [N 1301 Switching from dummy_G_no_anomaly1_ruleset to dummy_G_broadcast_vote_ruleset]
** [N 1301 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [N 1301 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [N 1301 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [N 1301 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [N 1301 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [N 1301 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [N End of cycle 1301 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** ------------------------------------------------
** ------------------------------------------------
** [B 1302 Pattern anomaly: unfamiliar component dummy_G_no_anomaly1_ruleset]
** [B 1302 Pattern anomaly: unfamiliar component dummy_G_no_anomaly1_ruleset]
** [B 1302 Pattern anomaly: inactive component G_no_anomaly1_ruleset in P]
** [B 1302 Pattern anomaly: inactive component G_no_anomaly1_ruleset in N]
** [B 1302 Switching from G_no_anomaly1_ruleset to G_broadcast_vote_ruleset]
** [B 1302 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [B 1302 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [B 1302 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [B 1302 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [B 1302 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [B 1302 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [B End of cycle 1302 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** ------------------------------------------------
** [P 1302 Pattern anomaly: unfamiliar component dummy_G_no_anomaly1_ruleset]
** [P 1302 Pattern anomaly: inactive component G_no_anomaly1_ruleset in N]
** [P 1302 Switching from G_no_anomaly2_ruleset to G_wait_for_majority_ruleset]
** [P 1302 Switching from G_no_anomaly3_ruleset to G_inhibit_neighbours_ruleset]
** [P 1302 Switching from G_wait1_ruleset to G_diagnose_ruleset]
** [P 1302 Switching from G_wait2_ruleset to G_suppress_execution_ruleset]
** [P 1302 Switching from G_wait4_ruleset to G_self_repair_ruleset]
** [P 1302 Switching from G_wait3_ruleset to G_recover_data_ruleset]
** [P End of cycle 1302 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** ------------------------------------------------
```

Attack on meta-levels of two agents P and N simultaneously.

Excerpt 2:

```
** [B 1303 Pattern anomaly: unfamiliar component dummy_G_broadcast_vote_ruleset]
** [B 1303 Pattern anomaly: unfamiliar component dummy_G_broadcast_vote_ruleset]
** [B 1303 Context-sensitive pattern anomaly: inactive component: G_broadcast_vote_ruleset]
** [B Broadcasting start vote 1 regarding N]
** [B Broadcasting start vote 1 regarding P]
** [B End of cycle 1303 Selected database contents:]
** [[broadcasted P 1303] [broadcasted N 1303]]
** [[current_target t4 150 150 treasure]]
** [[vote B N 1 1303] [vote B P 1 1303]]
** [[vote_awaited 1303]]
** ------------------------------------------------
```

B broadcasts its vote and waits for a response from other agents.

Excerpt 3:

```
** [B Responding to anomalies in agents N and P]
** [B 1306 Inhibiting response of P]
** [B 1306 Inhibiting response of N]
** [B 1306 Cluster to be suppressed P dummy_G_no_anomaly1_ruleset]
** [B 1306 Cluster to be suppressed N dummy_G_no_anomaly1_ruleset]
** [B 1306 Cluster to be suppressed P dummy_G_broadcast_vote_ruleset]
** [B 1306 Cluster to be suppressed N dummy_G_broadcast_vote_ruleset]
** [B 1306 Suppressing dummy_G_voting1_rulefamily of N]
** [B 1306 Suppressing dummy_G_voting1_rulefamily of P]
** [B 1306 Activating repair of: P G_broadcast_vote_ruleset]
** [B 1306 Activating repair of: N G_broadcast_vote_ruleset]
** [B 1306 Activating repair of: N G_no_anomaly1_ruleset]
** [B 1306 Activating repair of: P G_no_anomaly1_ruleset]
** [B End of cycle 1306 Selected database contents:]
** [[broadcasted P 1303] [broadcasted N 1303]]
** [[current_target t4 150 150 treasure]]
** [[responding P 1306] [responding N 1306]]
** [[vote B N 1 1303]]
** ---------------------------------------------------
** [P End of cycle 1306 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** [[inhibited_by B P 1306]]
** [[vote B P 1 1303] [vote B N 1 1303]]
** ---------------------------------------------------
** [N End of cycle 1306 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** [[inhibited_by B P 1306]]
** [[vote B P 1 1303] [vote B N 1 1303]]
** ---------------------------------------------------
```

After waiting for several cycles (in this case only 3), there is still no voting response from its neighbours. B assumes a timeout and proceeds with its response.

Excerpt 4:

```
** [P End of cycle 1307 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** [[inhibited_by B P 1306]]
** [[vote B P 1 1303] [vote B N 1 1303]]
** ---------------------------------------------------
** [RP 1307 Repaired P G_voting1_rulefamily]
** [N End of cycle 1307 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** [[inhibited_by B P 1306]]
** [[vote B P 1 1303] [vote B N 1 1303]]
** ---------------------------------------------------
** [RN 1307 Repaired N G_voting1_rulefamily]
```

Repair agents of P and N restore the correct rule-family.

Excerpt 5:

```
** [B 1311 Rolling back data to trusted state]
** [B End of cycle 1311 Selected database contents:]
** [[broadcasted P 1303] [broadcasted N 1303]]
** [[current_target t4 150 150 treasure]]
** [[responding P 1306] [responding N 1306]]
** ---------------------------------------------------
** [B 1316 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [B 1316 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [B 1316 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [B 1316 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [B 1316 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [B 1316 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [B 1316 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [P 1316 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [P 1316 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [P 1316 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [P 1316 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [P 1316 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [P 1316 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [P 1316 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [N 1316 Switching from G_broadcast_vote_ruleset to G_no_anomaly1_ruleset]
** [N 1316 Switching from G_wait_for_majority_ruleset to G_no_anomaly2_ruleset]
** [N 1316 Switching from G_inhibit_neighbours_ruleset to G_no_anomaly3_ruleset]
** [N 1316 Switching from G_diagnose_ruleset to G_wait1_ruleset]
** [N 1316 Switching from G_suppress_execution_ruleset to G_wait2_ruleset]
** [N 1316 Switching from G_self_repair_ruleset to G_wait4_ruleset]
** [N 1316 Switching from G_recover_data_ruleset to G_wait3_ruleset]
** [B 1319 Arrived at treasure: t4]
** [P 1319 Arrived at treasure: t4]
** [P End of cycle 1319 Selected database contents:]
** [[current_target t4 150 150 treasure]]
** ----------------------------------------------------
** [N 1319 Arrived at treasure: t4]
```

Data rollback and switch back to normal mode of all agents.

Appendix E

Glossary

Agent: any sequentially controlled, hierarchically organised intelligent system (normally an intelligent program).

Anomaly: discrepancy between *model*-predicted behaviour and actual behaviour.

Application level: the collection of *components*, which carry out the *primary task*. The application level is the "absolute" *object-level*, and has often been called the object-level when there is no risk of confusion.

Closed reflection: any kind of *distributed* reflection in which all *meta-level* components are subject to at least as much monitoring as *application level* components.

Component: any part of an architecture that does not contain concurrently running processes. A component may be defined on different levels: an agent, a ruleset or a rule.

Distributed reflection: any kind of reflection with the following properties:

- there is more than one version of the self-representation,
- each version constitutes a partial view of the whole system,
- no version has precedence over all others.

A distributed architecture may be *closed* or *open*.

Hierarchical reflection: any kind of reflection in which there is a single self-representation which has precedence over all others in determining the actions of the system.

Hostile environment: any environment in which accidental damage or deliberate attack can be made against *meta-level* components, including those concerned with anomaly-detection.

Independent meta-level action: any kind of action of a meta-level relative to an object-level which the object-level has no control over. For example, independent monitoring cannot be suppressed by the component being monitored. If the meta-level is believed to be hostile, it can be suppressed following a majority vote, however.

Meta-level: any component that monitors or modifies internal behaviour patterns of an *object-level* which may be an *application level*, another meta-level or the whole system (including the meta-level doing the monitoring). A meta-level may be an intrusion detection system.

Mode: a recognisable state of an agent which normally lasts long enough to be distinguished from a fluctuation. The timeline of an agent switching between modes is a series of *phases* (instances of modes).

Model: in this thesis, the part of the self-representation which specifies required behaviour and/or makes predictions about normal behaviour. Part of the model may be learned by observation of normal or acceptable behaviour.

Object-level: any component that is monitored and modifiable by a *meta-level*. The object-level is "absolute" if it carries out the *primary task*. It is "relative" if it is another meta-level. In the thesis the term "object-level" has mostly been used to indicate an absolute object-level (otherwise it can be confusing).

Phase: a stage in an agent's history with the following properties:

- the agent was in a recognisable *mode* during this time
- it has a clear startpoint and endpoint (there is marked change in execution patterns as a result of the phase starting or stopping).
- a phase occurs only once: it is a historical instance of a *mode*.

Primary task: the task of the application domain, which is normally user-visible and externally specified. (Also called "primary requirements" or "primary goals").

Protection boundary: boundary between components to be protected and unrelated components. The concept may be important if the integrity of a primary task is to be protected and the same computing infrastructure is shared with unrelated tasks which the intrusion detection is not "responsible for". The simplest is a physical boundary between an autonomous system and its surroundings. A protection boundary may change dynamically if new components (or agents) are added or removed.

Requirements boundary: boundary between acceptable (or normal) behaviour and unacceptable (or abnormal) behaviour. This boundary is partly (or completely) autonomously generated, depending on whether it is primarily about "normal" operation or about externally specified requirements. In artificial immune systems terminology, the "self/nonself" distinction is usually along this boundary.

Secondary tasks: tasks involving protecting the integrity of the *primary task*. They are often user-invisible and their precise details may emerge gradually as a result of learning and adaptation. (Also called "secondary requirements" or "secondary goals").

Bibliography

[Aleksander and Morton, 1995] Aleksander, I. and Morton, H. (1995). *An Introduction to Neural Computing*. International Thomson Publishing Company. Pages 158–162.

[Allen, 2000] Allen, S. (2000). *Concern Processing in Autonomous Agents*. PhD thesis, School of Computer Science, University of Birmingham.

[Almgren and Lindqvist, 2001] Almgren, M. and Lindqvist, U. (2001). Application-integrated data collection for security monitoring. In *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS, pages 22–36, Davis, California. Springer.

[Aspnes, 2000] Aspnes, J. (2000). Fast deterministic consensus in a noisy environment. In *Symposium on Principles of Distributed Computing*, pages 299–308.

[Aspnes, 2003] Aspnes, J. (2003). Randomized protocols for asynchronous consensus. *Distributed Computing*. (Invited survey paper for 20th anniversary special issue, to appear).

[Avizienis et al., 2001] Avizienis, A., Laprie, J.-C., and Randell, B. (2001). Fundamental concepts of dependability. Technical Report Research Report 01145, LAAS-CNRS.

[Balasubramaniyan et al., 1998] Balasubramaniyan, J. S., Farcia-Fernandez, J. O., Isacoff, D., Spafford, E., and Zamboni, D. (1998). An architecture for intrusion detection using autonomous agents. Technical Report COAST TR 98-05, Department of Computer Sciences, Purdue University.

[Bates et al., 1991] Bates, J., Loyall, A. B., and Reilly, W. S. (1991). Broad Agents. In *AAAI Spring Symposium on Integrated Intelligent Architectures*. American Association for Artificial Intelligence. (Repr. in SIGART BULLETIN, 2(4), Aug. 1991, pp. 38–40).

[Beattie et al., 2000] Beattie, S. M., Black, A. P., Cowan, C., Pu, C., and Yang, L. P. (2000). CryptoMark: Locking the Stable door ahead of the Trojan Horse. White Paper, WireX Communications Inc.

[Beaudoin, 1994] Beaudoin, L. P. (1994). *Goal processing in autonomous agents*. PhD thesis, University of Birmingham.

[Ben-Or, 1983] Ben-Or, M. (1983). Another advantage of free choice: Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, Canada.

[Bondavalli et al., 1995] Bondavalli, A., Chiaradonna, S., Giandomenico, F. D., and Strigini, L. (1995). Rational Design of Multiple Redundant Systems. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewoods, B., editors, *Predictably Dependable Computing Systems*. Springer Basic Research Series.

[Byrd et al., 2001] Byrd, G. T., Gong, F., Sargor, C., and Smith, T. J. (2001). Yalta: A collaborative space for secure dynamic coalitions. In *IEEE Systems, Man and Cybernetics Information Assurance Workshop*.

[Cachin et al., 2000a] Cachin, C., Camenisch, J., Dacier, M., Deswarte, Y., Dobson, J., Horne, D., Kursawe, K., Laprie, J.-C., Lebraud, J.-C., Long, D., McCutcheon, T., and Müller, J. (2000a). Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) – Reference Model and Use Cases. Project report - deliverable D1: `http://citeseer.nj.nec.com/cachin00maftia.html`.

[Cachin et al., 2000b] Cachin, C., Kursawe, K., and Shoup, V. (2000b). Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Distributed Computing (PODC)*, Portland, Oregon.

[Chandra and Toueg, 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

[Cohen, 2000] Cohen, P. (2000). A method for clustering the experiences of a mobile robot that accords with human judgments. In *Proceedings of AAAI'2000*.

[Cristian, 1991] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

[Cristian and Fetzer, 1999] Cristian, F. and Fetzer, C. (1999). The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657.

[Crosbie and Spafford, 1995] Crosbie, M. and Spafford, G. (1995). Active defense of a computer system using autonomous agents. Technical Report 95-008, Department of Computer Sciences, Purdue University.

[Dacier, 2002] Dacier, M. (2002). Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) – Design of an Intrusion-Tolerant Intrusion Detection System. Project deliverable - D10.

[Dasgupta and Attoh-Okine, 1997] Dasgupta, D. and Attoh-Okine, N. (1997). Immunity-based systems: A survey. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Orlando.

[Dasgupta and Forrest, 1996] Dasgupta, D. and Forrest, S. (1996). Novelty-detection in time series data using ideas from immunology. In *Proceedings of the International Conference on Intelligent Systems*, Reno, Nevada.

[Dechter et al., 1991] Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal contraint networks. *Artificial Intelligence*, 49:61–95.

[Dries, 2001] Dries, J. (2001). An Introduction to Snort: A Lightweight Intrusion Detection System. InformIT online article at `http://www.informit.com`.

[Fabre and Perennou, 1998] Fabre, J. and Perennou, T. (1998). A metaobject architecture for fault tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95. (Special Issue on on Dependability of Computing Systems).

[Fawcett and Provost, 1997] Fawcett, T. and Provost, F. J. (1997). Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1(3):291–316.

[Fetzer and Cristian, 1995] Fetzer, C. and Cristian, F. (1995). On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, CA.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Forrest et al., 1996] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press.

[Forrest et al., 1994] Forrest, S., Perelson, A. S., Allen, L., and Cherukun, R. (1994). Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*.

[Fox and Leake, 1995] Fox, S. and Leake, D. B. (1995). Using introspective reasoning to refine indexing. In *Thirteenth International Joint Conference on Artificial Intelligence (IJCAI95)*.

[Frijda, 1986] Frijda, N. H. (1986). *The Emotions*. Cambridge: Cambridge University Press.

[Ganek and Corbi, 2003] Ganek, A. and Corbi, T. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1).

[Ghosh et al., 1999a] Ghosh, A., Schwartzbard, A., and Schatz, M. (1999a). Using program behavior profiles for intrusion detection. In *Proceedings of the SANS Third Conference and Workshop on Intrusion Detection and Response*, San Diego, CA.

[Ghosh et al., 1999b] Ghosh, A. K., Schwartzbard, A., and Schatz, M. (1999b). Learning program behavior profiles for intrusion detection. In *Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Santa Clara, California.

[Gopalakrishna and Spafford, 2001] Gopalakrishna, R. and Spafford, E. (2001). A Framework for Distributed Intrusion Detection using Interest Driven Cooperating Agents. In *Fourth International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, Davis, California.

[Gordon and Chess, 1998] Gordon, S. and Chess, D. (1998). Where there's Smoke, there's Mirrors: the Truth about Trojan Horses on the Internet. In *Eighth International Virus Bulletin Conference*, Munich, Germany.

[Günther, 1971] Günther, G. (1971). Cognition and volition: a contribution to a cybernetic theory of subjectivity. In *Proceedings of the 1971 Fall Conference of the American Society of Cybernetics*, pages 119–135, Washington D.C.

[Helmer et al., 1998] Helmer, G., Wong, J., Honavar, V., and Miller, L. (1998). Intelligent agents for intrusion detection. In *Proceedings, IEEE Information Technology Conference*, pages 121–124, Syracuse, NY.

[Helmer et al., 2000] Helmer, G., Wong, J. S. K., Honavar, V., and Miller, L. (2000). Automated discovery of concise predictive rules for intrusion detection. *The Journal of Systems and Software*, 56(1).

[Hof, 2000] Hof, M. (2000). Using reflection for composable message semantics. In *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*. Available at `http://www.disi.unige.it/RMA2000`.

[Hofmeyr, 2000] Hofmeyr, S. A. (2000). An interpretative introduction to the immune system. *Design Principles for the Immune System and Other Distributed Autonomous Systems*, edited by L.A. Segel and I. Cohen. Santa Fe Institute Studies in the Sciences of Complexity. New York: Oxford University Press (In Press).

[Hofmeyr and Forrest, 2000] Hofmeyr, S. A. and Forrest, S. (2000). Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473.

[Hsueh et al., 1997] Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82.

[Ichisugi et al., 1992] Ichisugi, Y., Matsuoka, S., and Yonezawa, A. (1992). 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*, pages 24–35, Tokyo.

[Jansen et al., 2000] Jansen, W., Mell, P., Karygiannis, T., and Marks, D. (2000). Mobile agents in intrusion detection and response. In *12th Annual Canadian Information Technology Security Symposium*, Ottowa, Canada.

[Kaehr, 1991] Kaehr, R. (1991). Zur Logik der 'Second Order Cybernetics'. In *Kybernetik und Systemtheorie - Wissenschaftsgebiete der Zukunft?* IKS-Berichte, Dresden. (in German).

[Kaehr and Mahler, 1996] Kaehr, R. and Mahler, T. (1996). Introducing and modelling polycontextural logics. In *Proceedings of the 13th European Meeting on Cybernetics and Systems*. Austrian Society for Cybernetic Studies, Vienna, edited by Robert Trappl.

[Kaminka, 2000] Kaminka, G. (2000). *Execution Monitoring in Multi-Agent Environments*. PhD thesis, Computer Science Department, University of Southern California.

[Kaminka and Tambe, 2000] Kaminka, G. and Tambe, M. (2000). Robust multi-agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research (JAIR)*, 12:105–147.

[Kaminka and Tambe, 1998] Kaminka, G. A. and Tambe, M. (1998). What is wrong with us? improving robustness through social diagnosis. In *Proceedings of the 15th National Conference on Artificial Intelligence(AAAI-98)*.

[Kennedy, 1998a] Kennedy, C. M. (1998a). Anomaly-driven concept acquisition. In *Proceedings of the Workshop on 'Machine Learning and Concept Acquisition' of the 1998 German Conference on Artificial Intelligence (KI'98)*, Bremen, Germany.

[Kennedy, 1998b] Kennedy, C. M. (1998b). Evolution of self-definition. In *Proceedings of the 1998 IEEE Conference on Systems, Man and Cybernetics, Invited Track on Artificial Immune Systems: Modelling and Simulation*, San Diego, USA.

[Kennedy, 1999a] Kennedy, C. M. (1999a). Distributed reflective architectures for adjustable autonomy. In *International Joint Conference on Artificial Intelligence (IJCAI99), Workshop on Adjustable Autonomy*, Stockholm, Sweden.

[Kennedy, 1999b] Kennedy, C. M. (1999b). Towards self-critical agents. *Journal of Intelligent Systems. Special Issue on Consciousness and Cognition: New Approaches*, 9, Nos. 5-6:377–405.

[Kennedy, 2000] Kennedy, C. M. (2000). Reducing indifference: Steps towards autonomous agents with human concerns. In *Proceedings of the 2000 Convention of the Society for Artificial Intelligence and Simulated Behaviour (AISB'00), Symposium on AI, Ethics and (Quasi-) Human Rights*, Birmingham, UK.

[Kim and Spafford, 1993] Kim, G. H. and Spafford, E. H. (1993). Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. Technical Report Purdue Technical Report CSD-TR-93-071, West Lafayette, IN 47907-2004.

[Kornman, 1996] Kornman, S. (1996). Infinite regress with self-monitoring. In *Reflection '96*, San Francisco, CA.

[Lamport et al., 1982] Lamport, L., Shostack, R., and Peace, M. (1982). The Byzantine General's Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Laprie, 1995] Laprie, J.-C. (1995). Dependability - Its Attributes, Impairments and Means. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewoods, B., editors, *Predictably Dependable Computing Systems*. Springer Basic Research Series.

[Laprie et al., 1995] Laprie, J.-C., Arlat, J., Beounes, C., and Kanoun, K. (1995). Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewoods, B., editors, *Predictably Dependable Computing Systems*. Springer Basic Research Series.

[Leake, 1995] Leake, D. (1995). Representing self-knowledge for introspection about memory search. In *AAAI Spring Symposium on Representing Mental States and Mechanisms*, pages 84–88, Stanford, CA.

[Lee et al., 2001a] Lee, W., Stolfo, S., and Mok, K. (2001a). Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14:533–567.

[Lee et al., 2001b] Lee, W., Stolfo, S. J., Chan, P. K., Eskin, E., Fan, W., Miller, M., Hershkop, S., and Zhang, J. (2001b). Real time data mining-based intrusion detection. In *Proceedings of DISCEX II*.

[Lenat, 1983] Lenat, D. B. (1983). EURISKO: a Program that Learns new Heuristics and Domain Concepts. *Artificial Intelligence*, 21(1 and 2):61–98.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In *Proceedings of OOPSLA-87*.

[Maes, 1988] Maes, P. (1988). Issues in computational reflection. In Maes, P. and Nardi, D., editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland.

[Mashuhara and Yonezawa, 2000] Mashuhara, H. and Yonezawa, A. (2000). An object-oriented concurrent reflective language RBCL/R3. In *Object-Oriented Parallel and Distributed Programming*, pages 151–165. HERMES Science Publications.

[Matsuoka et al., 1991] Matsuoka, S., Watanabe, T., and Yonezawa, A. (1991). Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 231–250, Geneva, Switzerland. Springer-Verlag.

[Maturana and Varela, 1980] Maturana, H. and Varela, F. J. (1980). *Autopoiesis and Cognition: The Realization of the Living*. D. Reidel Publishing Company, Dordrecht, The Netherlands.

[Minsky, 2002] Minsky, M. (2002). The Emotion Machine. On-line book available at: `http://web.media.mit.edu/˜minsky/E1/eb1.html`.

[Mittal and Vigna, 2002] Mittal, V. and Vigna, G. (2002). Sensor-Based Intrusion Detection for Intra-Domain Distance-Vector Routing. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS'02)*, Washington, DC, USA.

[Muscettola et al., 1998] Muscettola, N., Morris, P., Pell, B., and Smith, B. (1998). Issues in temporal reasoning for autonomous control systems. In *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, USA.

[Oehlmann, 1995] Oehlmann, R. (1995). Metacognitive adaptation: Regulating the plan transformation process. In *AAAI Fall Symposium on Adaptation of Knowledge for Reuse*.

[Oehlmann et al., 1993] Oehlmann, R., Sleeman, D., and Edwards, P. (1993). Learning plan transformations from self–questions : A memory–based approach. In *Eleventh International Joint Conference on Artificial Intelligence (IJCAI93)*, pages 520–525, Washington D.C., U.S.A.

[Overill, 1998] Overill, R. E. (1998). How Reactive should an IDS be? In *1st International Workshop on Recent Advances in Intrusion Detection (RAID'98)*, Louvain-la-Neuve.

[Overill, 2001] Overill, R. E. (2001). Reacting to cyberintrusions: Technical, legal and ethical issues. *European Financial Law Review*, 8(1).

[Paxson, 1999] Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463.

[Pell et al., 1997] Pell, B., Bernard, D. E., Chien, S. A., Gat, E., Muscettola, N., Nayak, P. P., Wagner, M. D., and Williams, B. C. (1997). An autonomous spacecraft agent prototype. In Johnson, W. L. and Hayes-Roth, B., editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 253–261, New York. ACM Press.

[Podnieks, 1992] Podnieks, K. (1992). Around Gödel's Theorem: Hyper-textbook for students in mathematical logic. `http://www.ltn.lv/~podnieks/gt.html`.

[Porras and Neumann, 1997] Porras, P. A. and Neumann, P. G. (1997). Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference. National Institute of Standards and Technology*.

[Powell et al., 1988] Powell, D., Bonn, G., Seaton, D., Verissimo, P., and Waeselynck, F. (1988). The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing Systems (FTCS-18)*, Tokyo, Japan. IEEE CS Press.

[Powell and Stroud, 2001] Powell, D. and Stroud, R. (2001). Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) – conceptual model and architecture. Project deliverable - D2.

[Powell and Stroud, 2003] Powell, D. and Stroud, R. (2003). Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) – conceptual model and architecture. Project deliverable - D21.

[Pradhan and Banerjee, 1996] Pradhan, D. K. and Banerjee, P. (1996). Fault-Tolerant Multiprocessor and Distributed Systems: Principles. In Pradhan, D. K., editor, *Fault-Tolerant Computer System Design*. Prentice-Hall, New Jersey.

[Ptacek and Newsham, 1998] Ptacek, T. H. and Newsham, T. N. (1998). Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc.

[Quinlan, 1986] Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[Robinson, 2001] Robinson, C. L. (2001). Security requirements models to support the accreditation process. In *Second Annual Sunningdale Accreditor's Conference*.

[Rosen, 1991] Rosen, R. (1991). *Life Itself*. Columbia Univerity Press, New York, Complexity in Ecological Systems Series.

[Rosenbloom et al., 1988] Rosenbloom, P., Laird, J., and Newell, A. (1988). Meta-levels in SOAR. In Maes, P. and Nardi, D., editors, *Meta-Level Architectures and Reflection*, pages 227–239. North-Holland.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall International, Englewood Cliffs, NJ, USA.

[Schneider, 1990] Schneider, F. B. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319.

[Sekar et al., 1999] Sekar, R., Bowen, T., and Segal, M. (1999). On preventing intrusions by process behavior monitoring. In *USENIX Intrusion Detection Workshop*, pages 29–40.

[Setiono and Leow, 2000] Setiono, R. and Leow, W. K. (2000). FERNN: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1-2):15–25.

[Sloman, 1990] Sloman, A. (1990). Must intelligent systems be scruffy? In *Evolving Knowledge in Natural Science and Artificial Intelligence*. Pitman, London.

[Sloman, 1993a] Sloman, A. (1993a). The mind as a control system. In Hookway, C. and Peterson, D., editors, *Proceedings of the 1992 Royal Institute of Philosophy Conference 'Philosophy and the Cognitive Sciences'*, pages 69–110, Cambridge. Cambridge University Press.

[Sloman, 1993b] Sloman, A. (1993b). Prospects for AI as the general science of intelligence. In *Proceedings of the 1993 Convention of the Society for the Sudy of Artificial Intelligence and the Simulation of Behaviour (AISB-93)*. IOS Press.

[Sloman, 1995] Sloman, A. (1995). Exploring design space and niche space. In *Proceedings of the 5th Scandinavian Conference on AI (SCAI-95)*, Trondheim. IOS Press, Amsterdam.

[Sloman, 1997] Sloman, A. (1997). Designing human-like minds. In *Proceedings of the 1997 European Conference on Artificial Life (ECAL-97)*.

[Sloman, 1998a] Sloman, A. (1998a). Damasio, Descartes, alarms and meta-management. In *Symposium on Cognitive Agents: Modeling Human Cognition, at IEEE International Conference on Systems, Man, and Cybernetics*, pages 2652–7, San Diego, CA.

[Sloman, 1998b] Sloman, A. (1998b). "Semantics" of Evolution: Trajectories and Trade-offs in Design Space and Niche Space. In Coelho, H., editor, *Progress in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, pages 27–38. Springer-Verlag.

[Sloman, 2001] Sloman, A. (2001). What are virtual machines? Are they real? In *Workshop on Emotions in Humans and Artifacts*, University of Birmingham.

[Sloman and Poli, 1995] Sloman, A. and Poli, R. (1995). SIM_AGENT: A toolkit for exploring agent designs. In Mike Wooldridge, J. M. and Tambe, M., editors, *Intelligent Agents Vol II, Workshop on Agent Theories, Architectures, and Languages (ATAL-95) at IJCAI-95*, pages 392–407. Springer-Verlag.

[Smith, 1982] Smith, B. C. (1982). *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT.

[Somayaji, 2000] Somayaji, A. (2000). Automated response using system-call delays. In *9th Usenix Security Syposium*.

[Spafford and Zamboni, 2000a] Spafford, E. and Zamboni, D. (2000a). Design and implementation issues for embedded sensors in intrusion detection. In *Third International Workshop on Recent Advances in Intrusion Detection (RAID2000)*.

[Spafford and Zamboni, 2000b] Spafford, E. and Zamboni, D. (2000b). Intrusion detection using autonomous agents. *Computer Networks 34*, pages 547–570.

[Stolfo et al., 1997] Stolfo, S. J., Prodromidis, A. L., Tselepis, S., Lee, W., Fan, D. W., and Chan, P. K. (1997). JAM: Java agents for meta-learning over distributed databases. In *Knowledge Discovery and Data Mining*, pages 74–81.

[Tambe and Rosenbloom, 1996] Tambe, M. and Rosenbloom, P. S. (1996). Architectures for agents that track other agents in multi-agent worlds. In *Intelligent Agents, Vol. II*, LNAI 1037. Springer-Verlag.

[Turner et al., 1996] Turner, R., Turner, E., and Blidberg, D. (1996). Organization and reorganization of autonomous oceanographic sampling networks. In *Proceedings of the 1996 IEEE Symposium on Autonomous Underwater Vehicle Technology*, Monterey, CA.

[Turner, 1993] Turner, R. M. (1993). Context-sensitive reasoning for autonomous agents and cooperative distributed problem solving. In *Proceedings of the 1993 IJCAI Workshop on Using Knowledge in Its Context*, Chambery, France.

[Turner, 1995] Turner, R. M. (1995). Context-sensitive, adaptive reasoning for intelligent AUV control: ORCA project update. In *Proceedings of the 9th International Symposium on Unmanned Untethered Submersible Technology (AUV'95)*, Durham, New Hampshire.

[Tygar and Whitten, 1996] Tygar, J. D. and Whitten, A. (1996). WWW Electronic Commerce and Java Trojan Horses. In *Second USENIX Workshop on Electronic Commerce (EC96)*, Oakland, California.

[Valdes and Skinner, 2001] Valdes, A. and Skinner, K. (2001). Probabilistic alert correlation. In *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS, Davis, California. Springer.

[Varela, 1979] Varela, F. (1979). *Principles of Biological Autonomy*. North-Holland, New York, USA.

[Verissimo and Neves, 2001] Verissimo, P. and Neves, N. (2001). Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) – Service and Protocol Architecture for the MAFTIA Middleware. Project deliverable - D23.

[Verissimo and Rodrigues, 2001] Verissimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[Vigna et al., 2001] Vigna, G., Kemmerer, R., and Blix, P. (2001). Designing a web of highly-configurable intrusion detection sensors. In *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS, pages 69–84, Davis, California. Springer.

[Vigna et al., 2002] Vigna, G., Valeur, F., Zhou, J., and Kemmerer, R. (2002). Composable tools for network discovery and security analysis. In *ACSAC '02*, Las Vegas, NV, USA. network topology representation.

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

[Wieringa and Meyer, 1993] Wieringa, R. J. and Meyer, J.-J. C. (1993). Applications of deontic logic in computer science: A concise overview. In *Deontic Logic in Computer Science: Normative System Specification*, pages 17–40. Wiley.

[Wright and Sloman, 1997] Wright, I. and Sloman, A. (1997). Minder1: An implementation of a protoemotional agent architecture. Technical Report CSRP-97-1, University of Birmingham, School of Computer Science.

[Wu et al., 1999] Wu, T., Malkin, M., and Boneh, D. (1999). Building intrusion-tolerant applications. In *Proceedings of USENIX Security Symposium (Security '99)*, pages 79–91.

[Xu et al., 1995] Xu, J., Bondavalli, A., and Giandomenico, F. D. (1995). Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewoods, B., editors, *Predictably Dependable Computing Systems*. Springer Basic Research Series.

[Zamboni, 2001] Zamboni, D. (2001). *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, CERIAS TR 2001-42, West Lafayette, IN.