In *Journal of Cognitive Systems Research*,
Vol. 4, No. 2, 2003 pp. 89–117,

Autonomous Recovery from Hostile Code Insertion using Distributed Reflection

Catriona M. Kennedy and Aaron Sloman
School of Computer Science
University of Birmingham
Edgbaston, Birmingham B15 2TT
Great Britain

Abstract

In a hostile environment, an autonomous cognitive system requires a reflective capability to detect problems in its own operation and recover from them without external intervention. We present an architecture in which reflection is distributed so that components mutually observe and protect each other, and where the system has a distributed model of all its components, including those concerned with the reflection itself. Some reflective (or "meta-level") components enable the system to monitor its execution traces and detect anomalies by comparing them with a model of normal activity. Other components monitor "quality" of performance in the application domain. Implementation in a simple virtual world shows that the system can recover from certain kinds of hostile code attacks that cause it to make wrong decisions in its application domain, even if some of its self-monitoring components are also disabled.

Key words: anomaly, immune systems, meta-level, quality-monitoring, reflection, self-repair.

1 Introduction

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 (reflective) capability in order to detect and diagnose problems in its own components and to take recovery action to restore normal operation.

The simplest way to make an autonomous system reflective is to include a layer in its architecture to monitor behaviour patterns of its components and detect deviations from expectancy (anomalies). There are situations, however, where the monitoring layer will not detect anomalies in itself (e.g. it cannot detect that it has just been deleted, or replaced with hostile code). In previous papers [Kennedy, 1999, Kennedy, 2000] we called this problem "reflective blindness".

Traditional ways of improving resistance to attacks on a monitoring layer involve the addition of features to an existing software architecture and do not consider the software as a "whole" intelligent system. Examples include replication and voting [Cristian, 1991], design diversity [Hilford et al., 1997] and program self-checking methods (e.g. [Harman and Danicic, 1995]).

In contrast, our research investigates novel ways of *integrating* existing algorithms and techniques to form an enhanced coherent architecture. The aim is to build a complete autonomous system with some cognitive capability that can protect itself in a hostile environment. We use the term "reflection" in the sense of "meta-management" [Beaudoin, 1994] which involves (among other things) the ability of a cognitive system to detect problems in its internal processing and correct them.

Autonomous response and reconfiguration in the presence of unforeseen problems is already a fairly established area in remote vehicle control systems which have to be self-sufficient (see for example [Pell et al., 1997]). However, they do not specify how the system should recover from problems in its self-monitoring and control software, or the insertion of hostile code to take over control of the system.

1.1 Distributed Reflection

We address the problem of reflective blindness by *distributing* the reflection over multiple components so that all components are subject to monitoring from within the system. This does not entirely eliminate the "blindness" problem but it makes its consequences less severe (for example, not all interactions between components will be observable by the system itself but this may not affect survival in a particular hostile environment).

In [Kennedy and Sloman, 2002a] we presented a minimal prototype of an autonomous system whose highest level components are mutually observing agents. The agents acquire models of each other's normal execution patterns which they subsequently use to detect anomalies in each other's operation and repair any damage.

This first "damage-tolerant" prototype was able to continue operating indefinitely in an environment in which random deletion of software components takes place, including deletion of anomaly-detection and self-repair components. A damaged component merely stops functioning. (The failure is an "omission" failure).

1.2 Hostility detection

The contribution of this paper is to show how the minimal "damage-tolerant" prototype architecture in [Kennedy and Sloman, 2002a] can be extended so that the system detects negative effects of *hostile code* on its performance, even if some of the monitoring components are themselves disabled. We present a simple proof-of-concept implementation which responds to hostility by identifying and suppressing "foreign" components whose activity is correlated with the hostile effects.

"Hostility" is detected as a degradation of "quality" in system performance or in the environment as evaluated by the system itself (for example, does it detect that it is repeating the same action too often with no effect?). In addition to acquiring a model of normal "patterns", it also acquires a model of "acceptable quality" by observation of its own actions in the absence of hostile code.

The damage-tolerant prototype only recognised anomalies in execution patterns (in particular rule-firing patterns). We call these "pattern" anomalies, since their detection only requires comparison of patterns without interpretation of meaning. In contrast, quality evaluation requires knowledge of the task requirements.

Responding to attack on the basis of pattern-anomalies may be called a "pessimistic" policy, since anything unknown is assumed to be "bad". The extended prototype is "optimistic" about unfamiliar execution patterns and waits until it detects a degradation of quality before initiating a response. "Pattern" triggered response (requiring no quality evaluation) is a limiting special case of "hostility" triggered response (requiring some quality evaluation).

The remainder of this paper is organised as follows:

- Conceptual framework and basic architecture (sections 2 and 3)
- Recovery from hostile code attacks in a single agent architecture (sections 4 and 5)
- Recovery from more complex hostile code attacks in a two-agent mutually reflective architecture (section 6)
- Conclusions and future work.

2 Conceptual Framework and Methodology

The concept of *architecture* is central to our investigation. The word "architecture" is typically used to refer to a design for a system that can be decomposed into functionally distinct parts.

In this paper, we restrict the term "agent" to mean a sequentially controlled intelligent program with some autonomy and reasoning capability. "Sequentially controlled" means that the highest "authority" in the agent is an explicitly programmed sequence of "sense-decide-act" cycles. In this way an agent is an elementary unit which can be combined with other units to form a larger system whose actions may be emergent properties of interactions between the components.

Our methodology is *design-based* [Sloman, 1993], which aims to understand a phenomenon by attempting to build it. In practice, this is an informal rapid-prototyping approach.

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 very beginning. 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 environment?"

It is basically a process of "matching" architectures to environments (or "designs" and "niches" as they are called in [Sloman, 1995]). The result that we are looking for is a proof-of-concept implementation of an architecture that can survive in the selected environment.

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 which we are not investigating (such as low-level pattern recognition).

2.1 Broad-and-shallow architectures

To make rapid-prototyping manageable we use a "broad and shallow" approach [Bates et al., 1991]. A *broad* architecture is one which 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 which implements the component's function. Our final aim is not a shallow architecture, but shallowness is necessary initially in order to make the rapid-prototyping possible. This approach is particularly useful when we do not *start* with a complete set of requirements and a design, but use an incremental process of development and testing to clarify what is needed.

The equivalent of "scaling up" for a shallow architecture is "deepening", which means replacing the shallow components with more complex or scaled up versions. 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 particular, if a shallow distributed architecture can overcome the limitations of a shallow non-distributed architecture then a distributed architecture with "scaled up" components can have the same advantages over a non-distributed architecture whose components have been similarly scaled up. We call this the "deepening assumption".

Furthermore we assume that the considerable experience acquired in the use of AI-techniques in self-diagnosis, replanning and reconfiguration can also be incorporated in a real autonomous system based on a distributed reflective architecture.

The broad-and-shallow design-based methodology requires a simple virtual world for the purpose of rapid prototyping. The virtual world "Treasure" (based on the Minder scenario [Wright and Sloman, 1997]) is made up of several treasure stores, one or more ditches and an energy source. An autonomous vehicle must collect treasure, while avoiding any ditches and ensuring that its vehicle's energy level is kept above 0 by regularly recharging it. The "value" of the treasure collected should be maximised and must be above 0. The system "dies" if either collected treasure value or energy-level become 0 or the vehicle falls into a ditch.The world includes spontaneous growth and decay elements. Collected treasure continually loses value as it gets less "interesting". Treasure stores become more interesting (increase in value) the longer they have not been visited (growth). A configuration of the world is shown in Figure 1.

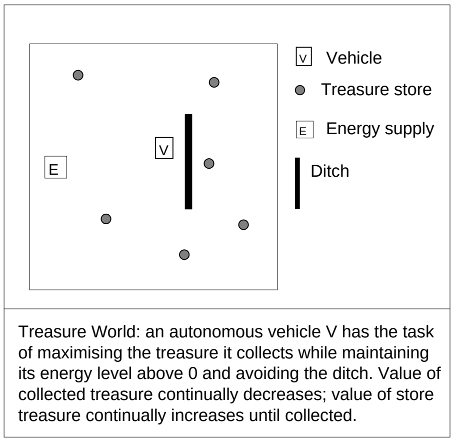


Figure 1: Treasure Scenario

2.2 Hostile environments

The Treasure world alone does not provide a sufficiently hostile environment to test the self-monitoring capability, since the spontaneous decay of vehicle treasure and energy do not involve damage or modification to the system software. We therefore simulate an "enemy" which modifies software components according to an attack scenario (which we define later). Since this damage to the software is real and not simulated, we can say that the system's software is "situated" in a hostile environment (although its hardware is not in this case). Examples of attack scenarios are in section 4.4.

The following restrictions are necessary if there is to be guaranteed anomaly detection and recovery.

1. *Effects of an attack can be sensed*: change of behaviour caused by an attack should be observable by agent sensors. The attack itself is invisible (the enemy agent's activity leaves no trace).

2. *Minimal 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.

3. *Untrusted components can be recognised*: We are excluding situations where the system can detect a quality degradation but it cannot recognise a component that may be causing it.

In practice, we have found that the minimum time interval between attacks has to be much longer in the hostility detection prototype than in the pattern-triggered system of [Kennedy and Sloman, 2002a]. This is because there may be some delay before the quality deteriorates sufficiently to be regarded as "hostile" and an identification of the rogue component can be made. Restriction 3 is a "shallowness" restriction. It may be relaxed for an architecture which can replan around a recurring problem which is undiagnosed, or alternatively the "untrusted" components can be found using a complex diagnosis algorithm (but we are not investigating replanning in this prototype).

3 Architecture

We now outline the simple agent architecture which carries out the above treasure-collecting task with no defence against intrusions. The architecture was implemented as a set of rules divided up into modules called rulesets. Table 1 shows a component hierarchy of individual rules, rulesets and ruleset groups according to function. The list of rulesets forming the middle column in the table is input to a rule interpreter to be processed in the order they appear. Similarly, rules within a ruleset are processed in the order of appearance. For space reasons, rules in the table are shown in abbreviated form and only some rules within a ruleset are shown. When processing a ruleset, the conditions of each rule are checked against the current contents of the agent's

Table 1: Selected architecture components

Function	Ruleset	Sample rules
Sense	external_sensors	see_treasure see_ditch see_energy_supply
Decide	evaluate_state generate_motive select_target	interest_growth interest_decay low_energy low_treasure new_target target_exists
Act	avoid_obstacles avoid_ditch move	near_an_obstacle adjust_trajectory near_ditch adjust_trajectory move_required no_move_required

database (which is effectively its working memory). The database is essentially a list of statements which are believed to be true at a certain time. They talk about specific objects. A rule's conditions often contain variables which match specific instances in the database. For example, if we consider the *external_sensors* ruleset, the rule *see_treasure* is expanded as follows:

"if any object O was seen with coordinates (X,Y) and
the object has features satisfying those of a treasure store
then remember the object as a treasure store with coordinates (X,Y)".

At this stage, various objects have been loaded into memory from the sensors but have not yet been "recognised". So the database will contain statements about specific values such as "An object labelled 't1' was seen with coordinates '(a, b)' and list of features '(m, n, o, p..)'. (Since this is a shallow architecture, the lowest levels of perception - such as determining the boundaries of an object - are not included in the rulesystem; the sensors are assumed to have done this pre-processing already).

Once an object has been found in the database which matches the "treasure" rule and the action of the rule has been done, the rule-interpreter can either stop processing this rule (i.e. ignore any other objects that match the treasure features) or it can continue to check if there are further objects with those features. In *external_sensors* the action is performed *for every* object satisfying the treasure condition (more generally for every combination of database statements which match the rule conditions). In some rulesets this option is not necessary and is switched off for efficiency reasons, meaning that processing of the rule (and the whole ruleset) stops after the first matching rule is processed.

The implementation is based on the SIM_AGENT package [Sloman and Poli, 1995] which uses POPRULEBASE as a rule-interpreter. In SIM_AGENT, an agent execution is a sequence of "sense-decide-act" cycles (for which we will use the term "agent-cycle"). This cycle corresponds to the ruleset groups in the first column of Table 1. Sensing, or acting, or both, may be internal or external to the agent. 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 sense-decide-act cycle for each agent is executed in one scheduler time-slice.

To monitor the operation of its own software the system must have a representation of internal events (e.g. in the form of execution traces) as well as access to its various components so that they can be repaired or replaced as necessary. We call this the "internal world". To explain this more precisely we summarise the notion of a reflective control system (introduced in [Kennedy and Sloman, 2002a] and earlier papers).

3.1 Reflective control systems

Figure 2(a) shows 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 and maintains the world within acceptable values. For example, in the Treasure scenario, acceptable values are defined in terms of treasure and energy levels as well as position of the vehicle with respect to the ditch. We assume that the agent has a model M_E (part of C_E) of the external world which represents the normal behaviour of the environment and is used to predict the next state after an action. The simple architecture in Table 1 corresponds to the object-level C_E in Figure 2(a). The representation of the world corresponds to M_E (for example, what does treasure normally look like?). The external sensors S_E show the actual state of the world.

The *meta-level* labelled C_I is 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. The model M_I (which is part of C_I) is used to predict the "normal" state of the internal world. Sensors and effectors are also used on the meta-level (S_I and E_I). We will see later that M_I has several different components. For now we just call it a single model. We call the two layered structure a *reflective agent*. The meta-level C_I is the executive component that has "authority" over other aspects of the agent and is a sequential process according to our "agent" definition. The four components M_E, S_E, M_I and S_I together can be regarded as the agent's *representation* of the world (including itself).

In Figure 2(a), reflective blindness becomes apparent in that C_I cannot reliably detect anomalies in its own operation (for example, if the anomaly-detection component of C_I is deleted or prevented from executing). To overcome the problem, Figure 2(b) shows one way in which the reflection can be "distributed". In this configuration the application task is carried out asymmetrically by one agent only (the primary) while the other is a passively monitoring backup. The diagram shows only one agent interacting with the external world. An

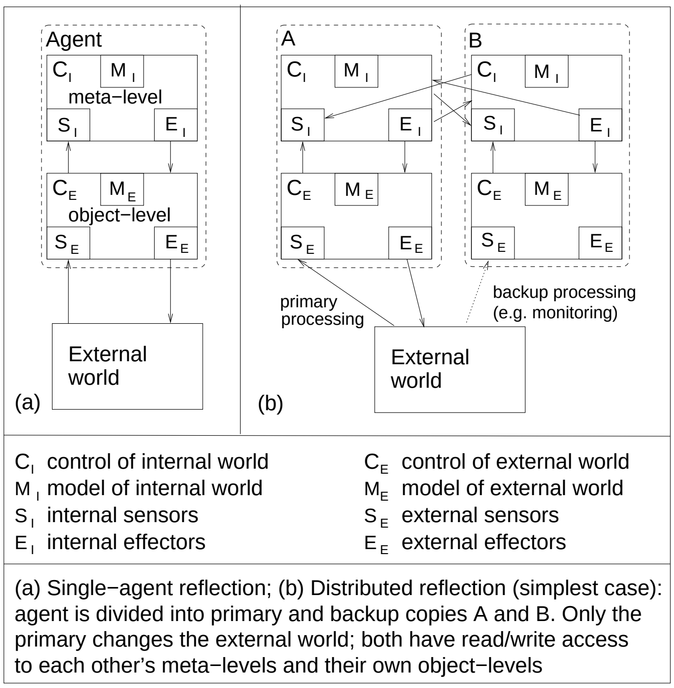


Figure 2: Reflective agent architectures

alternative which may have advantages would be to make one agent a specialist in treasure maximisation while the other is a specialist in safety and energy.

3.2 Designing an internal world and meta-level

Determining the content of the meta-level requires the definition of an internal world, which in turn requires some knowledge of the implementation.

During each cycle of an agent execution, the agent leaves an "observable trace" that can be sensed by its internal sensors (S_I). A single item in a trace is a recording of an execution event (an event-record). Events can be defined on various levels of detail: in [Kennedy and Sloman, 2002a], an "event" was recorded for every rule that had its conditions checked and for every rule whose actions were executed.

In this extended prototype, only the higher level components (rulesets) are monitored. An "event" is recorded every time a ruleset becomes active. A ruleset typically contains between three or four rules, meaning that the trace is "coarse-grained" leaving the agent with a high-level "summary" description of its own operation in the previous cycle.

Such a high-level summary is reasonable because we are now testing the *integration* of pattern anomaly detection with quality evaluation and not the anomaly-detection itself (which would require a more detailed level). We assume that lower level anomaly-detection can be successfully used if required, given increased processing resources (this is the deepening assumption above).

3.2.1 Artificial immune system as a meta-level

In our prototype, the content of C_I in Figure 2 is based on the general idea of an artificial immune system as presented by Forrest [Forrest et al., 1994], although there are differences which we explain below.

An artificial immune system (AIS) has two components: first, an algorithm which runs during a protected "training phase" to acquire a set of patterns ("self-patterns") characteristic of behaviour when no intrusion has occurred. This makes it possible later to discriminate between "self" and "nonself" patterns; secondly an anomaly-detection algorithm for use during the "operational phase" when real intrusions can occur. Forrest's training algorithm builds a database of normal patterns of activity as the model of self.

An alternative AIS is *negative selection* [Dasgupta and Forrest, 1996] which is more closely based on the natural immune system and involves the generation of a random population of unique detectors. All detectors which match "normal" patterns are eliminated during the training phase (hence the term negative selection). Thus if a detector matches some activity during the operational phase, the activity is anomalous (nonself).

The main difference between our approach and typical AIS is that we are aiming for a model containing symbolic statements about which components are normally active in which situations. In contrast, most AIS self-models are sets of subsymbolic strings which are matched with real execution patterns using numerical and string comparison methods. The "picture" of the whole system behaviour produced in that way has a more fine-grained resolution. However, an abnormal string or a significant mismatch between sets of strings does not necessarily identify a component which may be causing a problem.

3.2.2 Artificial immune systems and reflective blindness

We have focused on artificial immune systems because their stated objective is to make a distinction between self and nonself. Since this means that the system must protect *all* of its own components, it is similar to the kind of reflection that we require. However, AIS algorithms must be part of an appropriate architecture before they can satisfy this requirement. We briefly show why.

The internal sensor readings in Figure 2(a) would normally include the trace left by C_I itself. AIS algorithms will also include their own signature in the model of "self" (along with the remaining software in the system being protected, which is equivalent to our "object-level").

However, any anomaly in C_I's trace may mean that the anomaly-detection itself is defective. Similarly if the internal sensors are defective, and they leave an anomalous trace, these same defective sensors would be used to read the trace. Consequently the inclusion of C_I's own trace in the input to C_I would not be an effective use of monitoring resources and we have excluded it. The agent is therefore only partially reflective (it monitors its object-level only). We call such a configuration *hierarchical* reflection to distinguish it from the distributed case.

For the same reason, artificial immune *algorithms* on their own are not sufficient to satisfy the requirements of an immune system. Although the model of "self" would normally include the execution patterns of the immune system's anomaly detection algorithm, the system can only discriminate between self and nonself strings reliably *if its anomaly-detection component is intact*. For this reason we must consider the whole *architecture* into which the algorithm is integrated.

ARTIS [Hofmeyr and Forrest, 2000] is an architecture-based concept for an artificial immune system. It is very closely modelled on the natural immune system and is represented as a network of nodes called "localities", each of which runs an instance of the negative selection algorithm described above. This architecture covers the problem of self-monitoring implicitly using the decentralisation and diversity inherent in the natural immune system. Their approach differs from ours in that there are no explicit requirements for mutual monitoring between nodes (in this case "localities"). Later, in section 5.2 we explain how the agents can acquire models of each other's meta-levels by "taking turns" to observe each other recovering from selected forms of attack. First, we focus on what a single agent can do.

4 Hostility Detection in a Single Agent Architecture

Before considering the distributed architecture, we first show the architecture of a single participating agent (as in Figure 2(a)) and present a scenario in which the object-level of a single agent is attacked.

A summary of the main components of the agent during its operational phase is shown in Table 2. Note that the meta-level also implicitly contains *sense*, *decide* and *act* components, but they are not labelled for space reasons.

A single agent is capable of the following types of monitoring:

1. Detection of pattern anomalies in software execution patterns (e.g. an unknown component shows up in the trace)
2. Detection of quality degradation in the external world (e.g. vehicle treasure level is abnormally low) or a possible cause of future quality degradation (e.g. vehicle position is different from the anticipated position)
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

Table 2: Summary of operational phase components

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 generate_motive select_target
Act	avoid_obstacles avoid_ditch move
Anticipate	next_state

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 training phase and used to detect discrepancies in the operational phase. The result is three models of "normality" for internal patterns, external quality and internal quality respectively.

If our external environment were to undergo spontaneous changes, we might also have a model of normal *external* patterns, but this is not necessary in a static environment.

4.1 Pattern monitoring

To allow for the possibility of identifying hostile code, the agent can detect two different types of pattern anomaly:

1. Omission - characteristic activity associated with an essential component is missing from an agent-cycle trace. In our shallow architecture, all object-level components are active in each agent-cycle. The trace for a cycle is merely a way of confirming that all components are "alive". If a component identifier is missing from the trace, it is assumed that its associated component has failed.

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

A third kind, which we have not included due to the (temporary) shallowness requirement, is a *nonspecific* 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. It is difficult to use it for diagnosis since it does not point to a specific component.

As a source of hostile code, we have focused on "unfamiliarity" anomalies in the form of unknown component identifiers. Although this is a shallow form of anomaly-detection, it may not be entirely unrealistic. 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).

Examples of the kind of statements in a pattern model might be as follows:

1. "Ruleset S1 is always active"

2. "Ruleset S1 is intermittently active"

3. "Rulesets S1, S2 and S3 are always active together"

4. "Ruleset S1 of agent B is never active together with rulesets S4 and S5"

5. "Rule R1 of ruleset S3 of agent A almost always fires" (A more general version might be "... fires between 60% and 70% of the time")

The first two examples are typical of an object-level model for a single agent. Examples 3 and 4 involve some recognition of distinct types of activity in which some software components work together and represents structure in the "intermittent" set T. This becomes important when building a model of another agent's meta-level which we explain in section 5.2. (Note that example 4 refers to another agent). On a lower level, example 5 is typical of the damage-tolerant prototype presented in [Kennedy and Sloman, 2002a].

4.1.1 Training phase

In the current "summary" version of monitoring, the model-building of the object-level phase is simple: the agent counts the number of cycles each component is active: components that were active on every cycle experienced are hypothesised to be "essential" to the normal functioning of the agent's object-level. (It could also count the number of times its meta-level components are active - but for reasons outlined in section 3.2.2 we have excluded this). It is useful to define the following sets:

- S: set of all components,
- P: set of all components that are normally active on every cycle (for "positive"). An omission anomaly is detected when an element in P is absent from an agent cycle trace.
- N: set of all components that are normally never active (for "negative") but they are known to exist in the system.
- T: set of all components that are sometimes active (because they are "tolerated" during operational phase). An unfamiliarity anomaly is detected if something is in an agent cycle trace which is not in P or T.

In our prototype N is empty at the end of the training phase, meaning that all the known components have been active at least once. Therefore for anomaly detection, N is redundant. Note that the alternative immune system algorithm "negative selection" mentioned in section 3.2.1 would be centred on the set N instead. At the start of the training phase $N = S$. As the training phase progresses elements of N are gradually removed as they are encountered during "normal" system operation. An anomaly would be any behaviour that matches an element of N. To be effective, negative selection requires that elements of N are defined on a much lower level of granularity than we have here. For example, N may be a space of all possible rule firing sequences in each ruleset. An unusual firing sequence might indicate a Trojan Horse. In contrast, our prototype detects a new component if it is not in P or T and just assumes that the low-level pattern recognition has been done.

4.2 Internal and external quality monitoring

If the quality of the Treasure environment deteriorates after a new component has become active then the component should become untrusted. To detect a deterioration in quality, the system must know what "normal" quality is (which is assumed to be "acceptable"). During the training phase, the agent observes its own decisions and the external world to obtain the following boundary values of "acceptability":

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 of the world:
   (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.

An additional form of quality monitoring requires no training phase and involves the comparison between expected positions of the vehicle and its actual position. It is assumed that a large or repeated discrepancy between expected and actual position indicates a problem. The expected next vehicle position is calculated by ruleset *next_state* in the *Anticipate* component at the end of the agent-cycle. This is last entry in Table 2. At the beginning of the next cycle, *monitor_state* compares the readings of the external sensors with their expected value and is therefore able to detect a vehicle position anomaly. (In our scenarios, positions of other objects do not change, but the architecture can be extended to detect such anomalies).

A possible future generalisation would include an explicit quality specification (possibly derived from a formal requirements specification for the system being monitored). Instead of having to be trained to detect deviations from normal quality, the system could from the start monitor deviations from the requirements. Then learning mechanisms could through various kinds of training induce ways of detecting precursors of such quality deviations adding to the power and flexibility of the system. Of course, for robustness the explicit quality specification might have to be duplicated in different parts of the system, and perhaps expressed in different ways for different monitoring agents.

Typical "quality" statements in the model might look as follows:

1. "The time taken to collect treasure is always less than T "
2. "The level of vehicle energy is always above L"
3. "The interval between selection of target A and its later re-selection is always above I" (minimum observed interval between repeated actions)

They are conditions associated with "acceptable" performance.

4.2.1 Quality monitoring is part of the meta-level

Quality-monitoring is considered as part of an agent's meta-level because its function is similar to that of pattern-monitoring. They both act as triggers for internal responses, which involve *modification access to executable components*, i.e. suppressing or repairing components. In other words, they are part of the function of C_I (internal control) of Figure 2(a).

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). Such quality evaluation components are triggers *only* for actions in the external world or the planning of such actions and therefore belong to the object-level C_E. 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 object-level software; take *internal* action to suppress execution of untrusted software components, and restore (repair) trusted ones.

For example, the role of *monitor_state* looks ambiguous at first; it could be an object-level or a meta-level module. We have included it in the meta-level because its result may be a trigger for actions in the internal world. In contrast, *next_state* is part of the object-level because it is simply a calculation of what the next external state is expected to be; there is no access to executable components.

4.3 Single agent anomaly response

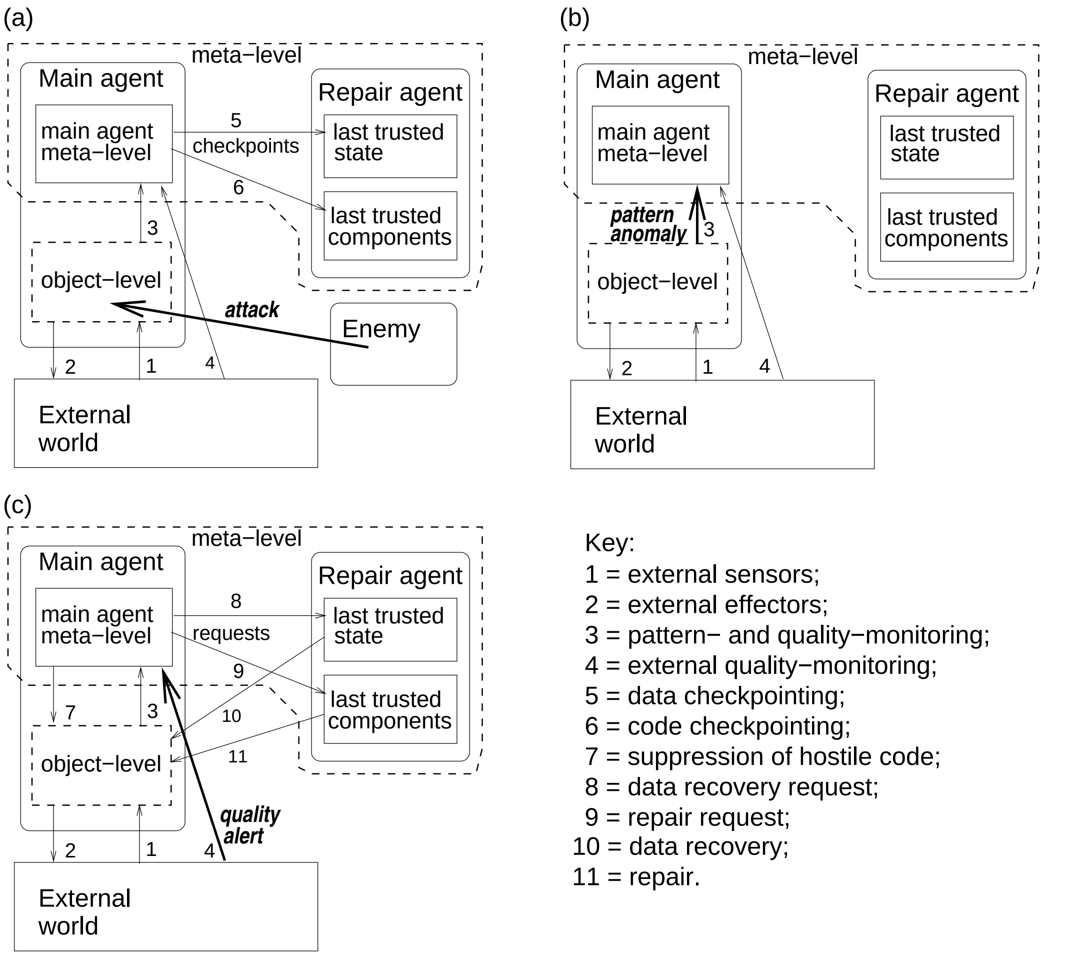
We assume that the participating agents are "optimistic" in that they will not respond to a pattern-anomaly unless it is accompanied by a quality deterioration. Once a pattern anomaly occurs, however, the new state of the system is not fully trusted until the anomaly has been present for a substantial time-period without undue problems. For example, the 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 code) and if necessary restore the old code by activating the repair-agent (if an old component was replaced). This assumes that identifying the code responsible for the anomaly is possible (restriction 3 in section 2.2)

A series of snapshots is shown in Figure 3. The meta-level of the agent is enclosed within a dashed rectangle and includes a "repair-agent". This is a service-providing agent which is activated on request and holds an independent backup copy of all rulesets as well as the last trusted state of the database. The repair agent is regarded as part of C_I of Figure 2(a). The object-level C_E is also enclosed within a dashed rectangle. There is



(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 3: Hostility Triggered Autonomous Recovery

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:

- `quality_alert`($World$):
  there is a quality problem (that is, an actual or anticipated quality deterioration) in the world
- `quality_alert`($Object\_level$):
  there is a quality problem in the agent's own object-level

- `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:
(`quality_alert`$(World)$ OR `quality_alert`$(Object\_level)$) AND `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 "self" 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.

4.3.1 Implementation summary

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. This means that a response always involves both suppression and repair.

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). There is no equivalent reversal of a repair. We assume that repairing something cannot *cause* additional damage; it can only fail to correct existing problems. (This assumption may have to be challenged in future work, since a mistaken repair may be similar to "patching" of a software error which might cause unpredictable side-effects).

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 3. Suppression and repair modules are outlined in Tables 4 and 5 respectively. The pseudocode 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.

4.4 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.)

Table 3: Outline of diagnosis component

```
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 disabled;

  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 4: Outline of suppression component

```
define RULESET suppress_execution
  RULE suppress:
  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:
  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;
```

- **Scenario 2**: modify the *external_sensors* ruleset, which collects and summarises information provided by the external sensors.

Scenario 1: Normally the longer a treasure store has not been visited, the greater the agent's "interest" in it becomes. In the first attack scenario, a critical parameter is modified so that the agent's interest in a treasure store *decreases* the longer it has not been visited.

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 all the others will have decreased below its interest in the old target (because the others have been less recently visited). In test runs, the effect was to produce repetitive selecting of the same target, which the meta-level interprets as a possible infinite loop and hence a quality alert. Although this takes some time to be detected,

Scenario 2: The second scenario involves interfering with the interpretation of sensor readings. We tested the agent's ability to recover if the position of the energy source is falsified by the sensor interpretation rules. 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.

Table 5: Outline of repair component

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

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

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

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 scenario 1, 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 of the energy source 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?

4.4.1 Implementation summary

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.

To describe the implementation of 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 6 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 7 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 8 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.

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).

To implement Scenario 2 the enemy agent modifies the ruleset *external_sensors* which extracts the relevant sensor information and stores it in summarised form. The sensors themselves are provided by the SIM_AGENT

Table 6: Correct ruleset for evaluating external world

```
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 7: Ruleset for generating a motive based on state evaluation

```
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 is 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 is 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;
```

toolkit and are not modified. The hostile version of the *external_sensors* ruleset is shown in Table 9.

4.5 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 (*adjust_trajectory*). If this goes

Table 8: Hostile ruleset for evaluating external world

```
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 9: Hostile ruleset for interpreting external sensors

```
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;
```

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.

A possible solution is to identify "critical" components of the system as those whose malfunctions cannot be repaired once they have been manifested. Then the monitoring software may include static checking of code, or regularly testing components in "simulation" mode (preventing any damage to the real environment) to see if they still work as predicted.

5 Shared Hostility Detection in a Two-agent Architecture

We now return to the distributed architecture in Figure 2(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 its neighbour's meta-level can be correlated with a quality deterioration in the external world as can anomalies in the agent's own 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.

In the current 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 in its own object-level by comparing its quality model with actual values (e.g. if an action has taken longer than the longest duration ever experienced). The backup agent also has the same model which it uses to detect external quality problems (e.g. low vehicle energy) and external anomalies. 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, which happens in scenario 1 above).

5.1 Environmental restrictions

The particular environment which "matches" the two agent distributed architecture has a number of restrictions, which we list below. In future work, we will begin to lift these restrictions gradually as we explore architectures which are more complex and less shallow.

R1: *A single attack only affects one agent*. 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). This has been tested in practice.

R2: *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 it 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.

We have found that restrictions R1 and R2 are necessary for guaranteed recovery in any two-agent system. If R1 is lifted, all available anomaly-detection components may be disabled simultaneously and the system will not recover.

If R2 is lifted, a meta-level can be "hostile". 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.

The first type of scenario can cause serious problems for a two-agent architecture. If there is hostile code on a meta-level which "lies" about an anomaly or quality deterioration, the lying agent has the authority to take over control of the vehicle and suppress any component in the neighbouring agent without its decision being "questioned". A three-agent system with voting can overcome this problem (provided that the voting mechanism is also sufficiently distributed to prevent simple disabling attacks on it). A proof-of-concept implementation of voting is in [Kennedy and Sloman, 2002b].

To survive in the above kind of environment, the two-agent prototype requires a model of a meta-level which allows it detect "omission" anomalies. We now explain the model acquisition process.

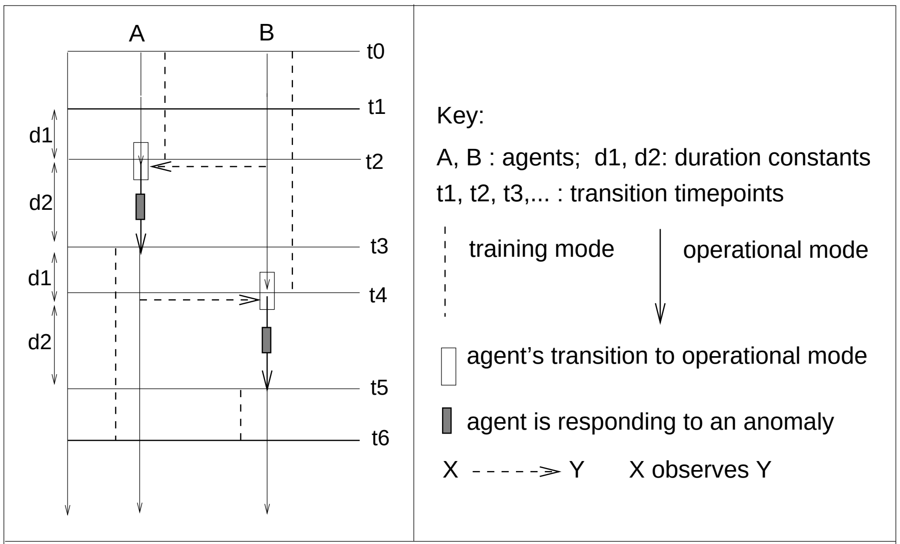
5.2 Acquiring a model of a meta-level

The training phase was divided into a series of "sub-phases" to allow agents to observe each other recovering from limited attacks, so that normal patterns of anomaly-detection and recovery can be included in the model. The model of a meta-level should be at least as detailed as the model of object-level components.

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.

5.2.1 Mutual observation

Figure 4 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*). The diagonal dashed lines highlight the interim operational phase for each agent. An observation window is a concatenation of two segments labelled



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 4: Mutual observation in stages

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. 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_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.

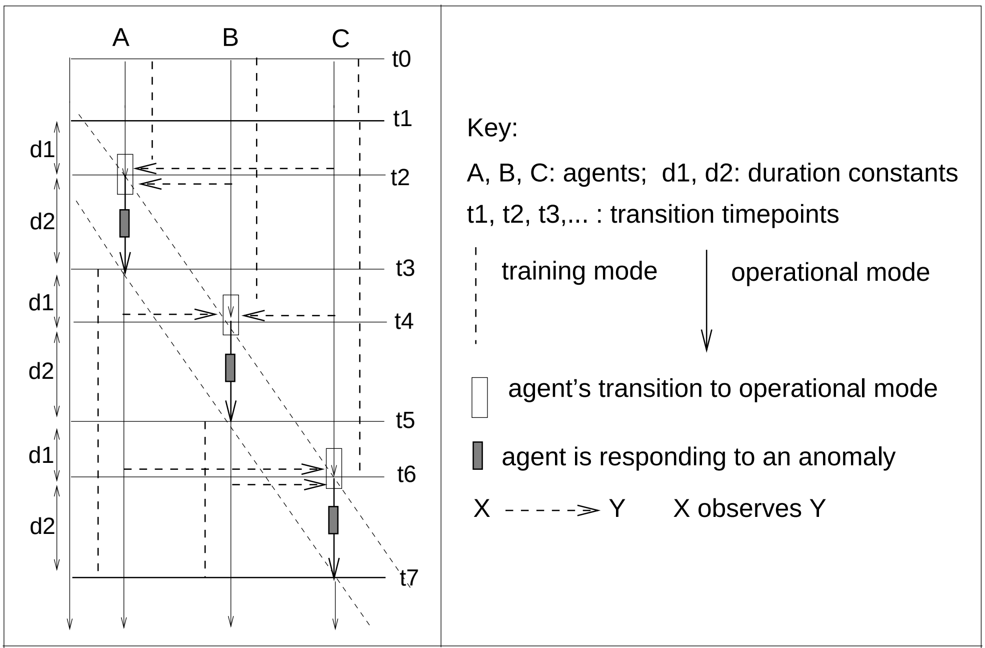
To ensure agents have an opportunity to observe each other in both modes, there is no need for a strict timing pattern. However the following constraints apply:

2. The phase durations d_1 and d_2 in Figure 4 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.

The timeslice before d_1 is the initial training phase during which an agent acquires a model of its own object-level. Its duration plays no role in the mutual observation which follows. Models acquired by mutual observation are not expected to include all possible states of the observed software. For example, the training phase now has an additional class-discovery subphase, which is not recognised by the observing agent as being different from "normal" training phase. Similarly, the agent in training mode can only observe a meta-level as it responds to problems in its object-level ("meta-object" detection); the training mode agent cannot observe a meta-level detecting and responding to an anomaly in another meta-level ("meta-meta" detection). In some cases a 3-agent configuration may be needed. However, if the same code is used for both "meta-object" and "meta-meta" detection it is difficult to disable the "meta-meta" part without also disabling the "meta-object" capability (where anomalies can be detected since it is already represented in the model).

5.2.2 Training time scalability

To show that mutual observation is not limited to two agents, a training phase design for 3 agents is shown in Figure 5. If d_T is the total duration of the training phase, d_I is initial training phase duration, w is the



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 5: Training phase for 3 agents

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 in the diagrams but its ending is marked by t_1 the first transition timepoint.

The meta-level training time does not scale up very well for more than two agents if the training is also "deepened" (for example if interactions between agents are also observed). A discussion of this would go beyond the scope of this paper, however.

5.2.3 Discovering types of activity

We now present the model acquisition algorithm which is used by an agent in training mode during an observation window. This algorithm was designed specifically for construction of the prototype.

Returning to Figure 4, 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.

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 components suddenly become inactive followed immediately by a number of new components becoming active (in other words, when there is a major discontinuity in d's execution trace). The pseudocode is shown in Table 10. POPRULEBASE provides two execution modes: "stop after the first matching rule" or "process all rules and

Table 10: Acquiring a model of a meta-level

```
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;
```

perform actions for each one that is satisfied". In Table 10 and all other pseudocode presented here, the execution mode is set to "all rules". This also means the following: when there are multiple statements in the database which match the variables in the rule's conditions in different ways, a "rule instance" is created for each one of them, and the rule's actions are executed for each instance (as was explained earlier at the start of Section 3). If "all rules" is switched off then there is only a single action per ruleset at most (for the first matching instance of a rule).

For example, the first rule in Table 10 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 for each consistent combination of bindings. The database contents contain all the internal sensor readings for this cycle (i.e. the event-records). The last starting time and (if applicable) the last stopping time of each known event is also recorded in the database.

As explained earlier an "event" can be defined on the "summary" or the "detailed" level (meaning execution of a ruleset or a rule respectively). In Table 10 and elsewhere “mode 1” and “mode 2” are the names we use for two generic modes: one immediately prior to a detected change, and one immediately after.

Clearly the algorithm above is “shallow” in that it only copes with small amounts of data and does not tolerate noisy or gradual transitions. According to the deepening assumption, the above algorithm should be replaceable by a more general learning or data mining algorithm that can handle noise and uncertainty. The only requirement is that it finds some way of clustering software components which tend to work together.

Our general philosophy is to focus on those features that provide the most information and produce crisp unambiguous distinctions. This is a commonly used technique in concept learning and some data mining algorithms such as [Quinlan, 1986], [Setiono and Leow, 2000] and [Lee et al., 2001].

5.2.4 Mutual exclusion

Modes of an activity are mutually exclusive. This means that components active in one mode are never active in another mode of the same activity. A component 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 components which 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 components should or should not be active in that mode. If there is an activity whose current mode cannot be identified (e.g. neither mode’s components are active or some combination of both are active), there is a non-specific pattern anomaly called “mode unidentified”. In this way the modes behave like contexts. Context-sensitive anomaly-detection (including mode identification) is shown in the pseudocode in Table 11.

Table 11: Context sensitive anomaly detection

define RULESET *Detect_Anomaly*;
  RULE *detect_inconsistency*:
  if there exist two events belonging to *the two* 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 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;

5.2.5 Hypothesis revision

All mode-associated components are expected to be *always* active in that mode. When the observing agent detects a transition, it hypothesises that all newly active components are essential aspects of the new mode. If in a subsequent cycle it finds that some of an activity’s new components become inactive then these inactive new components 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 active components falls 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.

5.2.6 Quality training phase

An agent's model of a meta-level contains a "quality evaluation activity" which can be in two modes: "quality training" or "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. The initial training phase happens before the mutual observation phase shown in Figure 4, i.e. before T1 of both agents (not shown in the diagram).

At present, the observing agent r 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 (during its training phase) have associated a deteriorated quality with a characteristic response by the observed agent and a 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 at least 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. In other words, there are no false-positives in this scenario.

5.3 Technical issues

To reproduce the prototype, it is useful to explain how the pseudocode in Table 10 relates to the P, T and N sets defined in section 4.1.1. The ruleset is in two stages:

1. If the agent detects a transition, it creates a set of all components that became inactive, and a set of new components that became active. Effectively, components previously belonging to P and N respectively are now moved to T.

2. The agent then creates a label for all components involved in the transition, that is the union of old components and new components. This label identifies a *type of activity* that has changed modes. The components 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 the components that are always active whether in training or operational mode or whether responding to an anomaly or not. The set T of components that are sometimes active is divided into partitions as follows:

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

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

The "structured" partition of T may be defined as a function $Struct : T \rightarrow \texttt{Powerset}(T)$ 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 active components are clustered into modes, i.e. $Struct(T) = T$.

Each $Act\_set(T, i)$ is further partitioned into m "mode sets" each containing the components that are active 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 components active 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 anmaly-detected). Thus the complement of a mode-set within its activity set is the one remaining mode-set. We use the term "anomaly complement" as an abbreviation.

5.3.1 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.

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.

During the main operational phase, if an agent d returns briefly to training phase then none of d's components associated with anomaly-response modes are active. Consequently, its observer r would wrongly detect a pattern-anomaly (a "mode unidentified" false-positive).

Similarly, there may be some components that are only occasionally active in a particular mode, but never active in any other mode of the same activity. They do not exist in the quality-monitoring prototype. Details of the more general hierarchical model acquisition algorithm can be found in [Kennedy and Sloman, 2002c].

5.4 Testing the architecture

We now show how the meta-level model helps one agent to repair another whose meta-level is disabled. 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 of A is attacked, the neighbouring agent should detect a *pattern* anomaly, indicating that a normally active component has become inactive. When the object-level component of A is attacked, a quality deterioration results. Then the meta-level's lack of response can be detected by the neighbouring agent.

The object-level attack is the same as for the single-agent case discussed in section 4. 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]).

5.4.1 Situations involving a quality alert

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):

- `quality_alert`($World$):
  quality problem in the external world

- $\texttt{quality\_alert}(\mathit{Object\_level})$:
  quality problem in the agent's own object-level

- $\texttt{pattern\_anomaly}(\mathit{Object\_level})$:
  pattern anomaly in the agent's own object-level

- $\texttt{pattern\_anomaly}(\mathit{Meta\_level})$:
  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 restrictions 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. $(\texttt{quality\_alert}(\mathit{World}) \vee \texttt{quality\_alert}(\mathit{Object\_level})) \wedge \texttt{pattern\_anomaly}(\mathit{Object\_level})$
   Response: hypothesise that the problem is in its own object-level; try to identify and repair it.

2. $(\texttt{quality\_alert}(\mathit{World}) \vee \texttt{quality\_alert}(\mathit{Object\_level})) \wedge \texttt{pattern\_anomaly}(\mathit{Meta\_level})$
   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. See also argument below.

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

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

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

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

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

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 restriction that the time interval between attacks on two different meta-levels must be sufficient for the correctly functioning meta-level 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 policy when a pattern anomaly is detected in the execution of any component of the backup agent. If the primary agent detects a pattern anomaly in the backup agent's 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 policy has already been covered by the damage-tolerance prototype and it would be redundant to include examples of it here.

5.4.2 Data recovery

During normal operation, both agents regularly checkpoint the most essential data to their repair-agents. This data includes the following:

- current state of the external environment: for example, the position of the vehicle, whether it is currently in the middle of a path around the ditch, whether it is collecting treasure etc.
- current meta-level information: for example, the time elapsed since the agent selected the same target (to detect senseless repetitions) and the time spent on the current action so far (to detect unacceptable delays).
- current state of its expectancy about the external environment, in particular the vehicle position (to minimise false alarms about unexpected vehicle position and detect true anomalies).

After an agent has suppressed or repaired one or more of its components (or its neighbour's components) it requests its repair-agent to broadcast the checkpointed data to all agents. The broadcast involves writing directly to each agent's database and overwriting all untrusted data. Once they receive the broadcast, the agents resume processing from the same starting state and their subsequent decisions are synchronised as required.

The fault may have caused the agents to have different beliefs about the external environment. The backup agent may also select a different action from the primary so that they will disagree about the expected vehicle position and about the time since they last selected the same action. If the agent that carried out the repair also broadcasts the data we can be fairly sure that the correct data is being broadcasted (if we assume the environmental restrictions R1 and R2). However, even if the data was not perfectly accurate the most important issue is that agents are starting again with *the same* beliefs. Even if they are slightly wrong initially they will be corrected if the agents are now functioning correctly.

In our current implementation, the models of the external and internal worlds are not checkpointed because they do not change during operational phase. The software itself also remains unchanged. If there are regular software upgrades, some form of "life-long-learning" may be required so that the normal patterns and acceptable quality values associated with the upgraded software may be incorporated into the respective models. The current state of the software and its associated quality and pattern models may then be regularly checkpointed. The last trusted upgrade may then be restored in the event of a serious quality problem with a subsequent upgrade.

5.4.3 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. The following meta-level components were attacked.

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

In each case, 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 target from that chosen by the backup (in the backup's "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 has an alternative way of recognising a quality deterioration by detecting a vehicle anomaly. It correlate this with a pattern anomaly in its own object-level and uses the information in the pattern anomaly report to identify and suppress the hostile code.

6 Related Work

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 expected behaviour model is similar to our pattern model and the ideal behaviour can be compared with our quality models, with the difference that our system represents what is normally *acceptable* rather than "ideal". Recognised discrepancies with expectation are similar to our "pattern anomalies" while discrepancies with ideal behaviour are comparable to our "quality alerts". 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) differ from our system in the following ways:

- They are all built on the basic assumption that the *architecture* of reflection is hierarchical with a single layer representing the meta-level. Although the meta-level may contain different components, these components do not monitor or "repair" each other as they do in our system.

- They do not have a concept of a "self" which 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. See Section 2.2.

- In general, the definition of "success" and "failure" (or good and bad "quality") is externally specified. We have made some initial steps in overcoming this limitation for self-monitoring systems, although our system is still very simple.

Apart from single agent architectures, some multi-agent models are also comparable with our work. The idea of using multiple agents to compensate for the "blindness" of a single agent is 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 fault in its own operation by observing the reaction of other agents.

Kaminka and Tambe's agents are not intended to be components in a single cognitive system, but a team of separate agents. In contrast, our system is more like a distributed "nervous system" for a single control system. It is interesting to point out the differences between the two architectures: 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 3.1). 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).

7 Conclusions and Future Work

The prototype has shown that it is feasible to implement distributed reflection in scenarios involving hostile code, although still in a very limited form. It has also shown that an integrated cognitive architecture can be applied to areas such as intrusion detection where traditionally such an approach has not been considered.

One of the main limitations of the prototype is that agents are based on identical code. Both agents evaluate the quality of the world (and their own decision-making) using the same measurements. The current configuration requires that both agents' processing must be synchronised. This is because their expectancies about the next state of the world must be in agreement. Otherwise the backup agent will continually be "surprised" by correct behaviour. For example, the primary agent may move the vehicle in a way that the backup agent would not have done, causing the backup to wrongly anticipate a quality problem.

Synchronisation has the disadvantage that correct operation depends critically on precise timing, a situation which may be exploited by an attacker (for example, by slowing processing in one agent). Therefore, future work will investigate distributed quality evaluation using different measures of "quality" which are not dependent on precise timing.

The mutual observation stage of the training phase may be more easily implemented in parallel with agents running on separate processors gathering data on each other's normal activities. It may also be advantageous to have several agents which remain in training mode after the remainder of the network has become operational. In this way the models of normal behaviour can be regularly updated to include new patterns of activity whose effects are "acceptable".

7.1 More complex cognitive architectures

Our prototype is intended to be a rudimentary cognitive architecture, although it is not comparable to human-like cognition. However, the recognition of "hostility" is a step towards giving artificial systems human-like "concerns", since the system does not just execute any instructions indiscriminately. In contrast, a typical AI system would not "care" if some of its rules are changed to make it do the opposite of what it was designed for. This may have serious consequences in areas such as medical diagnosis. The issue is discussed in more detail in [Kennedy, 2000].

In the more general case with more than two agents, and where each agent may be a specialist, we may have some characteristics of a "Society of Mind" model of cognition [Minsky, 1986]. Then instead of a primary/backup system, the resultant internal processing and external actions of the system are a combined effect of cooperation and competition between the various specialist agents (some of which may represent certain kinds of motivation (such as hunger, curiosity etc.) Applying distributed reflection to such a system means that some or all of the agents monitor each other and can intervene in each other's actions.

Such an architecture would not on its own represent a theory of human cognition. However, the general concept of mutual monitoring is supported by Minsky in section 1-10 of [Minsky, 2002] where he argues that the advantages outweigh the risk of "chaos" (where agents continually interrupt each other for example). A major challenge is ensuring that the system as a whole acts in an orderly and correct manner (in particular, if it is a safety-critical system). We have begun to investigate this problem in [Kennedy and Sloman, 2002b] which presents a three agent proof-of-concept implementation with voting.

References

[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.

[Cristian, 1991] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

[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.

[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)*.

[Ghosh et al., 1999] Ghosh, A., Schwartzbard, A., and Schatz, M. (1999). Using program behavior profiles for intrusion detection.

[Harman and Danicic, 1995] Harman, M. and Danicic, S. (1995). Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162.

[Hilford et al., 1997] Hilford, V., Lyu, M., Cukic, B., Jamoussi, A., and Bastani, F. (1997). Diversity in the software development process. In *Proceedings of IEEE WORDS'97*.

[Hofmeyr and Forrest, 2000] Hofmeyr, S. A. and Forrest, S. (2000). Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473.

[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, 1999] Kennedy, C. M. (1999). 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.

[Kennedy and Sloman, 2002a] Kennedy, C. M. and Sloman, A. (2002a). Acquiring a self-model to enable autonomous recovery from faults and intrusions. *Journal of Intelligent Systems*, 12. (To appear).

[Kennedy and Sloman, 2002b] Kennedy, C. M. and Sloman, A. (2002b). Closed Reflective Networks. Technical Report CSR-02-3, University of Birmingham, School of Computer Science.

[Kennedy and Sloman, 2002c] Kennedy, C. M. and Sloman, A. (2002c). Reflective Architectures for Damage Tolerant Autonomous Systems. Technical Report CSR-02-1, University of Birmingham, School of Computer Science.

[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., 2001] Lee, W., Stolfo, S., and Mok, K. (2001). Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14:533–567.

[Lenat, 1983] Lenat, D. B. (1983). EURISKO: a program that learns new heuristics and domain concepts. the nature of heuristics III: program design and results. *Artificial Intelligence*, 21(1 and 2):61–98.

[Minsky, 1986] Minsky, M. (1986). *The Socety of Mind*. Simon and Schuster, New York.

[Minsky, 2002] Minsky, M. (2002). The Emotion Machine. On-line book available at: `http://web.media.mit.edu/~minsky/E1/eb1.html`.

[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.

[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.

[Quinlan, 1986] Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[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, 1993] Sloman, A. (1993). 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 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.

[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.

[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.