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 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 its component's behaviour patterns 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 [11, 12] we called this problem "reflective blindness".

Reflective blindness is easily overlooked in software development because the usual approach is to focus on algorithms within a security layer and not on the architecture that contains it. Some forms of this problem have been recognised in network security systems, for example [3], page 18:

> Fault tolerance is (also) a recursive concept; the mechanisms designed to tolerate faults must be protected against the faults likely to affect them.

However, traditional solutions to the problem 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 [4], design diversity [9] and program self-checking methods (e.g. [8]).

1.1 Distributed Reflection

Our solution to the problem is to *distribute* the reflection over multiple components so that all components are subject to monitoring from within the system. In [13] 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 architecture enabled the system to survive indefinitely in an environment in which random deletion of software components took place, including deletion of anomaly-detection and self-repair components. This "damage-tolerant" architecture recovers from *omission* failures only, meaning that a damaged component merely stops functioning (because of its deletion).

1.2 Extending the Basic Architecture

This paper shows how the minimal prototype architecture in [13] was extended to enable an autonomous system to detect negative effects of *hostile code* on the performance of its software components, even if some of the anomaly-detection components are themselves disabled. Negative effects are degradations of "quality" in system performance or in the environment (for example, is it repeating the same action too often with no effect?)

The first prototype only recognised anomalies in execution patterns (in particular rule-firing patterns). We call these "pattern" anomalies, since their detection only requires syntactic pattern comparisons. In contrast, quality evaluation requires some semantic 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. However, a "quality degradation" may itself be partially defined as a deviation from "normal" actions in satisfying the task requirements. Therefore pattern-anomaly triggered response is a limiting special case of response on the basis of quality degradation. The opposite extreme is to use a definition of quality degradation which is independent of unfamiliarity.

1.3 Why Autonomous Response?

One may argue that autonomous response based on quality evaluation and pattern anomalies is dangerous, since it depends on correct identification of the hostile component. The wrong diagnosis may have unacceptable consequences. In many situations, however, there may be "safe" ways of reconfiguring a system by replacing an untrusted component with a trusted version, even if the untrusted component was not the cause of the problem (in which case the diagnosis process would have to continue).

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 [14]). It is also becoming increasingly necessary for ordinary software systems (such as operating systems) to be fault- and intrusion-tolerant which means acting autonomously, since there may not always be time for a system administrator to understand what is going on and take action. An example is the need to respond very fast to prevent the negative effects of a Trojan horse by identifying and suppressing hostile code execution and attempting to reverse or limit any damage that has already happened.

Although this is clearly a very hard problem, we assume that the considerable experience acquired in the use of AI-techniques in autonomous robotics is also applicable to intrusion-tolerant systems. Examples include fault-diagnosis, reconfiguration and replanning. Intrusion-tolerant systems have the following additional features:

- more concern with *software* diagnosis and its reconfiguration than with hardware (although both may be required).
- more concern with data *content* and its relationship to user requirements (e.g. some types of content may be sensitive or valuable and require more protection, while others are relatively unimportant)
- deadlines and resource constraints are usually not physically imposed but relate to the user's application or business.

In this paper we are mostly concerned with the conceptual framework of an autonomous vehicle because its actions and responses can be easily visualised. Later we consider how to apply the same architecture to a less easily visualised scenario such as information retrieval systems.

2 Conceptual Framework and Methodology

Our methodology is *design-based* [15], which aims to understand a phenomenon by attempting to build it. In practice, this is an informal rapid-prototyping approach. The concept of *architecture* is central. We define an *architecture schema* to be a specification of constraints on the functions of components and on the interaction between components. An *architecture* is a specification of actual components and their interconnections. It is an *instance* of the constraints being applied and can be directly implemented. In contrast, architecture schemas can exist at different levels of refinement. Examples are given later.

We should emphasise that the constraints in an architecture schema are "soft" rather than strict formal requirements. The design process should 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.

2.1 Broad-and-shallow architectures

Our approach is also "broad and shallow" [1]. 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. It is also necessary to make some assumptions, which we list below.

2.1.1 Shallowness assumption

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

2.1.2 Recovery assumption

If the system can *detect* a problem (and make a decision on whether there there really is a problem) then it can diagnose and repair it. In other words, all problems are recoverable if they can be recognised. This is a consequence of the "broad" approach. Since the investigation is about reflection on the architecture level, and about compensating for reflective blindness, we are not concerned with details of diagnosis and recovery algorithms. Only minimal versions of these are implemented, in order to produce a "complete" autonomous system.

2.2 Designing a scenario

The broad-and-shallow design-based methodology requires a simple scenario or virtual world for the purpose of rapid prototyping. We can relate this to software engineering by considering the user-specified requirements a software system must satisfy, and must be protected by an intrusion-detection system. We can call these requirements the *primary task* to distinguish it from the *secondary task* of protection.

Survival is the capability to satisfy the primary task in the presence of faults and intrusions without external help. A *hostile environment* is one in which the system's executive and control components (*including* any anomaly-detection and self-repair mechanisms) can be disrupted or directly attacked.

2.2.1 A simple virtual world

For rapid prototyping of architectures, we use a virtual world called "Treasure" in which the primary task is defined. Treasure is based on the Minder scenario [18]. The world 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. Collected treasure continually loses value as it gets less "interesting". Treasure stores that have not been visited recently are more interesting (and thus will add more value) than those just visited. The system "dies" if either collected treasure value or energy-level become 0 or the vehicle falls into a ditch. A configuration is shown in figure 1. This is the task whose satisfactory performance needs to be protected. We have

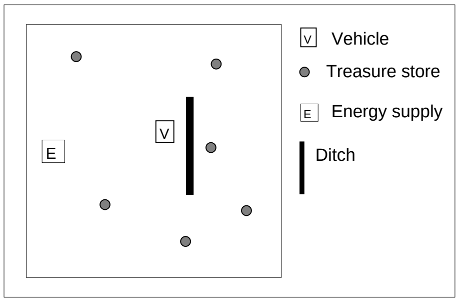


Figure 1: Treasure Scenario

chosen this scenario partly for practical reasons (because it is easily visualised) and partly because it involves two different levels at which intrusions and responses can take place, namely the vehicle control and the value of the "treasure" produced. It can be compared with a scaled-down version of a science scenario which contains the following components:

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

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

A real world example might be environmental monitoring using autonomous ocean sampling vehicles such as in [17]. It is important to keep such examples in mind to show up areas where our current architecture and simulated environment are extremely shallow and to identify the kinds of scaling up that would be required. For example, in the Treasure scenario, the fault- and intrusion-tolerance capability of the software (which is real) does not depend the energy-level of the vehicle (which is simulated). In the real world, however, it probably *would* depend on the energy-level of the vehicle, since it might have to conserve computational resources.

In a general network security scenario, the level corresponding to the "vehicle" would be the network routing and congestion control components and the level corresponding to "treasure" would relate to user-visible applications such as e-mail or web pages. An example primary task might be a data mining service. However, such a scenario is more difficult to visualise and to represent graphically in a virtual world and the system's boundaries are less clearly defined.

3 Architecture

Before defining the architecture of the whole system, we first outline the simple agent architecture which carries out the above primary task with no defence against intrusions. The architecture was implemented as a set of rules divided up into modules (rulesets). Table 1 shows a component hierarchy of individual rules, rulesets and ruleset groups according to function.

For example, the "Sense" component has one ruleset "external_sensors", in which a rule fires if a certain object is recognised. All rules normally fire because all expected objects are usually present,

Table 1: Selected architecture components

Function	Ruleset	Sample rules
Sense	external_sensors	see_treasure see_ditch see_energy
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

although changes can be tolerated. For example, the system continues to work correctly if objects are moved around (although adding or removing objects requires re-running the simulation with different parameters).

For space reasons, rules in table 1 are shown in abbreviated form. For example, the rule "see_ditch" is effectively asking the question: "is there an object whose dimensions and features match those of a ditch?" and taking the appropriate action if the conditions are true. Similarly, the rule "adjust_trajectory" asks "should the trajectory be adjusted?". For example, the vehicle may be moving very close to the ditch and it may have to make some corrections to avoid it.

Rules are run by a rule-interpreter, which runs each ruleset in the order specified in the agent architecture definition. The implementation is based on the SIM_AGENT package [16]. In SIM_AGENT, an agent execution is a sequence of sense-decide-act cycles which may be called *agent-cycles*. Each agent may be run concurrently with other agents by a scheduler, which allocates a time-slice to each agent. For simplicity we assume that agents are run at the same "speed" and that exactly one agent-cycle is executed in one scheduler time-slice.

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 [13] and earlier papers).

3.1 Reflective control systems

Figure 2(a) shows a two-layer structure (architecture schema) containing an object-level and a meta-level respectively. The object-level is a control system C_E for an external world. We assume that the agent has a model M_E (part of C_E) of the external world to maintain the environment 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. The simple architecture in table 1 plays the role of the object-level C_E in figure 2(a). The internal representation of the world plays the role of M_E (for example, what does treasure normally look like?).

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 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 call the two layered structure a *reflective agent*.

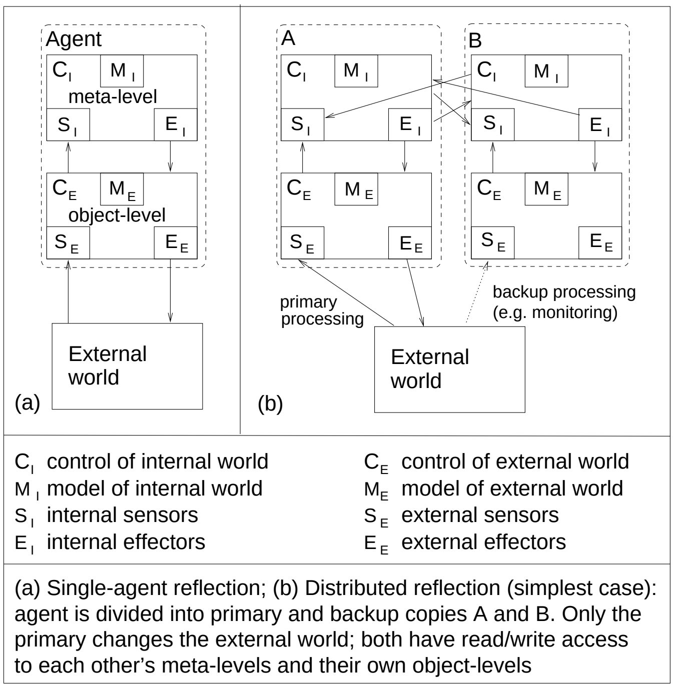


Figure 2: Reflective agent architectures

3.2 Distributed reflection

In figure 2(a), reflective blindness problem 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". A more detailed discussion of this problem is in [13].

3.2.1 Primary/Backup Configuration

In figure 2(b) 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 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.

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

3.3 Designing an internal world and meta-level

We first show the refinement of the architecture schema for the single agent shown in figure 2(a). Designing an architecture that can be implemented involves filling the meta-level with content. This 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). An *event record* is generated for each rule that has its conditions checked and for each rule whose actions are executed. The trace read by S_I is a sequence of such event records. Thus the agent has a record of its own operation in the previous cycle.

The internal sensor readings would normally include the trace left by C_I itself. However, as argued above, there is no component within the single agent that can monitor this trace independently and respond reliably to problems indicated by it. For example, 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 have 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).

3.3.1 Artificial immune systems

In our prototype, the content of C_I in figure 2(a) is similar on a high level to Forrest's artificial immune system introduced in [6].

An artificial immune system has two components: first, an algorithm which runs during a protected "training phase" to acquire the capability to discriminate between "self" and "nonself" patterns, and secondly an anomaly-detection algorithm for use during the "operational phase" when real intrusions can occur. In Forrest's algorithm, the training phase involves the building of a "signature" of the normal activity of the system, which can be compared with actual activity during operational phase. In our very simplified version of an immune system, the "signature" is a set of logical statements indicating which events almost always happen (and would be anomalous if they did not happen) and which events rarely or never happen (meaning they would be anomalous if they happened). Details of training and model acquisition algorithms are different from the details of Forrest's algorithm and are given in [13].

Other artificial immune system algorithms include *negative selection* [5] 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). We have used a simplified version of signature-based detection because it can be used to detect *absences* if normal execution events, something which negative selection cannot do.

There is no need for the meta-level C_I to be an artificial immune system. Any kind of intrusion-detection system might also be used. We have focused on immune system algorithms because their stated objective is to build a model of "self" and to make a distinction between self and nonself. In other words, they serve to make a system reflective in the way that we require. However, as argued in [13] and [10], artificial immune systems do not explicitly address reflective blindness as a software architecture problem.

4 Hostile environments

The Treasure scenario alone does not provide a hostile environment according to our definition, 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).

A hostile modification environment is one in which "enemy" agents can cause the software of the autonomous system to behave in a destructive (or self-destructive) way. The "enemy" agents may represent deliberate intruders or "environmental damage agents" such as radiation.

The targeted autonomous system must recognise that the new code is hostile and restore itself to its former state by suppressing the execution of the hostile component and carrying out any self-repair actions as necessary.

4.1 Restrictions

The prototype environment has a number of restrictions, which we list below (they are effectively assumptions required for the correct operation of the prototype architecture). In future work, we will begin to lift these restrictions gradually as the architecture becomes less shallow.

4.1.1 Effects of an attack can be sensed

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

4.1.2 Only one agent is attacked at a time

A single attack only affects *one agent*. 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.

4.1.3 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.

In practice, we have found that the minimum time interval has to be much longer in the hostile insertion environment than in the omission failure environment of [13] because there may be some delay before the quality deteriorates sufficiently to be detected and diagnosed. This is largely due to the optimistic policy.

4.1.4 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 is 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.

It is useful to show what kind of situations are excluded by this restriction. Destructive behaviour on a meta-level (*non*-omission failures) fall into he 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 is difficult without the capability to evaluate the performance of a meta-level directly; for example, does the false-positive rate suddenly increase beyond its normal value? Our architecture currently only determines when the meta-level is *not* responding when it should. Quality monitoring of a meta-level would involve difficult problems such as recognition of deception, which go beyond the scope of the current implementation. However, they are being considered in a three-agent prototype.

The second type of scenario does not introduce anything that is not already covered by external attack, given the present restrictions.

4.1.5 Shallowness restrictions

There are some additional restrictions related to shallowness, which are necessary in order to make the problem manageable. As a concrete example of the need for shallowness, we can consider the problem of diagnosis, which in our case requires the precise identification of a component to be suppressed (i.e. the one containing hostile code). One of the difficulties of identifying hostile code from a rule-firing trace is that the new code may have triggered a number of unfamiliar events which happened shortly afterwards. Each of these events can be associated with different components (rules or whole rulesets), thus leaving a long list of "untrusted" components (e.g. rules that were never active previously), only one of which is the real cause of the problem. In the case of autonomous recovery, a wrong diagnosis can result in more damage, since a harmless component may be suppressed.

In the real world, such a problem could be managed by using many independent sources of information and different kinds of monitoring, e.g. include monitoring of executable file sizes, or comparison of their content or monitor data or code access patterns. Since this cannot be done in a rapid-prototyping environment with limited resources, we make the problem manageable by making the architecture and environment "shallow" in the following ways:

- Restricting the number of forms in which hostile code can exist and the methods by which it can be inserted into the system.
- Simplifying the anomaly-detection mechanism so that most of the low-level pattern analysis is assumed to be already done.

Although this sounds like a large number of restrictions, there is one additional restriction required by a hierarchical architecture, namely that the anomaly-detection components are never attacked. This restriction is *not* required by the distributed architecture.

4.2 Design of Intrusion Scenarios

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

5 Quality Monitoring 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 a single agent is attacked. A summary of the main components of such an agent during 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.

5.1 Monitoring types

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

1. Detection of "syntactic" anomalies in software execution patterns (e.g. an unknown component leaves an event record in the trace)
2. Detection of an actual or imminent quality degradation in the external world (e.g. vehicle's energy level is abnormally low)
3. Detection of an *internal* quality-related problem in one of its own software components; this may be an actual performance degradation (e.g. it is taking too long to perform an action) or the component may be acting in a way that is expected to cause quality degradation in the world or in other components (e.g. interval between repeated actions is abnormally low).

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

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

5.1.1 Pattern anomaly detection

The damage-tolerant system presented in [13] was only designed for *omission* anomalies - absences of expected events in the execution trace. The hostility-tolerant system allows an agent to detect two different types of pattern anomaly:

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

2. Unfamiliarity - unfamiliar activity appeared in the trace. In our shallow architecture, an event-record itself is "unfamiliar" in that it contains an unrecognised component identifier; in a deeper architecture, "unfamiliarity" may mean a novel sequence of lower-level events such as program behaviour profiles [7].

A third kind, which we have not included due to the 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 therefore 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).

5.1.2 Quality monitoring

If the quality of the Treasure environment deteriorates after a new component is added then the component should become untrusted; otherwise its behaviour may be accepted as part of the system's own signature (depending on the "optimism" or "pessimism" of the policy).

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

1. Internal observation of own decisions:
   (a) Shortest observed interval between repeated actions for each kind of action (to help in detecting infinite loops later)
   (b) Longest observed durations of each action (to help in timeout detection)
2. External observation 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. It is possible to extend it in various ways, e.g. by including statistical methods.

An additional form of quality monitoring requires no training phase and involves the comparison between expected positions of the vehicle and their 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. At the beginning of the next cycle, "monitor_state" compares the readings of the external sensors with their expected value and 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).

5.1.3 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 rules. 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 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.

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

5.2 Single agent anomaly response

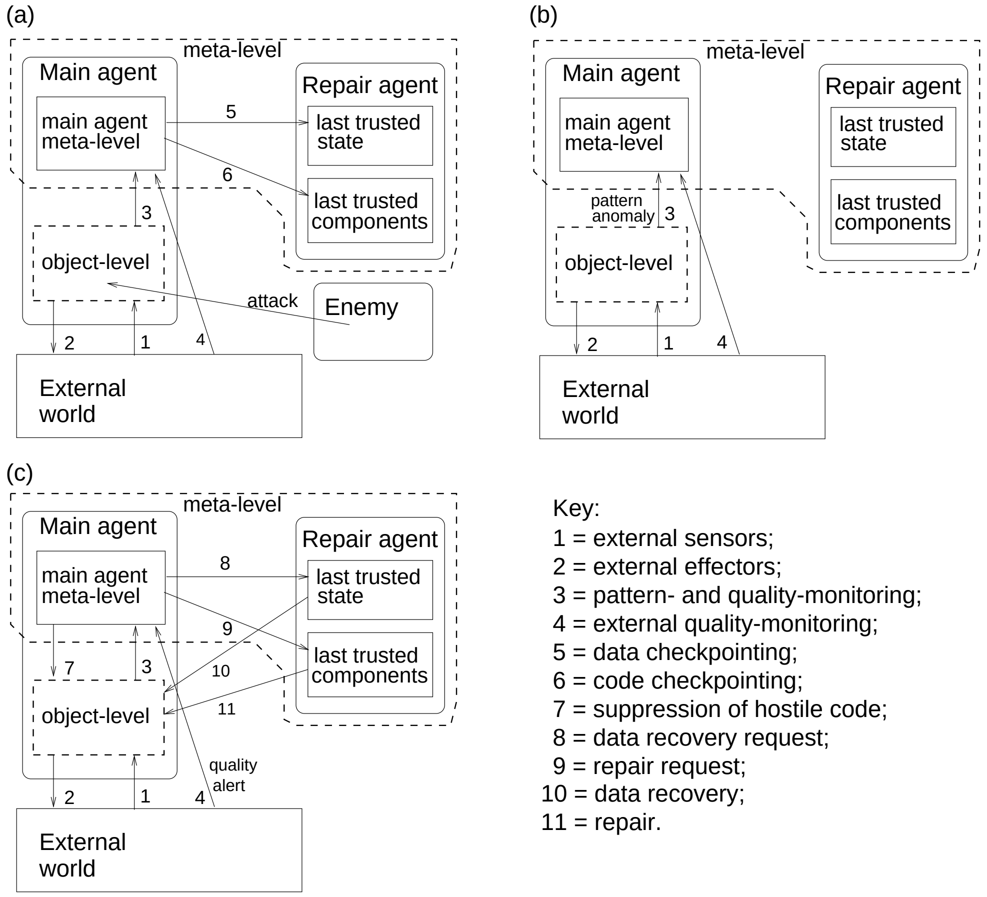
We assume that the participating agents are "optimistic" in that they will not respond to a pattern-anomaly unless it is accompanied by a quality deterioration. Once a pattern anomaly occurs, however, the new state of the system is not fully trusted until the anomaly has been present for a substantial 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).

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.



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

Figure 3: Quality Triggered Autonomous Recovery

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

- `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.

5.2.1 Implementation summary

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

[`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:

[`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

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 requiring repair;

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

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.

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;
```

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 indicates that repair is complete
  then
    wait for possible quality improvement

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

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.

5.3 Object-level Attack Scenarios

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

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

Scenario 1: To describe the first attack scenario, we give pseudocode outlines of two important rulesets on the object-level which determine the agent's decisions about which target to seek and whether it should interrupt its current activity to recharge its energy level. "Subjective interest" is the value the agent assigns to a treasure store. The first ruleset, shown in table 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. 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. The effect is to produce repetitive selecting of the same target, which the meta-level interprets as a possible infinite loop and hence a quality alert.

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;
```

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

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;
```

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;
```

continually choose to remain at the current target, even if it is no longer collecting any "objective" value from it. (Objective value is also "consumed" by a vehicle located at it, while the objective value of other stores tends to grow). The agent's meta-level observes that it is spending too much time at the same treasure store and interprets this as a quality problem (timeout).

Scenario 2: The second attack scenario involved interfering with the interpretation of external sensor readings. The sensors themselves are provided by the SIM_AGENT toolkit and cannot (easily) be modified. However, the agent has a ruleset *external_sensors* which extracts the relevant sensor information and stores it in summarised form. The hostile version of the *external_sensors* ruleset is shown in table 9. In the hostile code, the position of the energy source is falsified as shown in the

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;

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

In test runs, the first problem to be detected was a timeout due to the recharge operation taking longer than expected (the time it takes to be fully charged). As with the *evaluate_state* scenario, the agent was able to recover after some delay.

The ability of the agent to recover from the situation depends on its distance from the true position 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?

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.

5.4 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 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.

6 Quality Monitoring in the Distributed 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.

6.1 Primary/backup organisation

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

If the backup agent detects a problem in the primary agent, it immediately takes over control. The primary agent can detect quality problems by comparing its internal quality model with actual values (e.g. if an action has taken longer than the longest duration ever experienced or if the energy level has reached its lowest level ever). The backup agent also holds the same model and can compare its expectancy with reality. It can also detect anomalies in the vehicle position. For example if it has chosen a target and made a decision on how to move the vehicle, it will expect a certain vehicle position in the next sensory update. If there is a repeated large discrepancy between the actual and expected position of the vehicle, this may indicate a problem (e.g. the primary agent's target selection criteria may have been illegally modified, which happens in scenario 1 above).

6.2 Acquiring a distributed quality model

6.2.1 Meta-level modes

In distributed reflection, agents acquire models of each other's meta-levels by observing each other in various states during the training phase. For example they should observe each other detecting and recovering from an anomaly so that they have a model of normal execution patterns during these states. Two alternative training phase structures are shown in figure 4. Each agent's total training

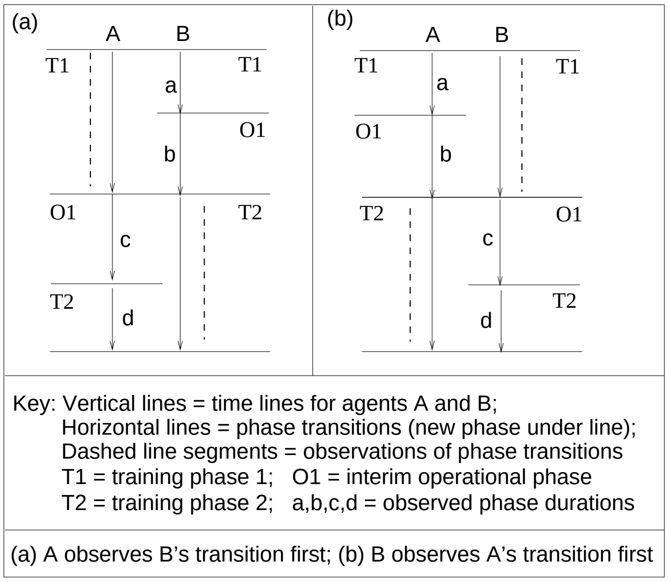


Figure 4: Temporal constraints

time is shown as a vertical line and is divided into training phase 1 (T1), interim operational phase (O1) and training phase 2 (T2) with the following constraints:

1. if T1 of A ends before T1 of B then O1 of A should end before O1 of B;

2. observed phase durations (a, b, c and d 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.

At the end of the training period, an agent divides up the execution events of its neighbour's meta-level into mutually exclusive classes called *modes*. The meta-level events are divided into training or operational mode; operational mode events are further subdivided into normal events and anomaly-response events. In other words, if the meta-level is in operational mode then it is either in normal mode or responding to an anomaly. The algorithm is presented in detail in [13].

In the distributed implementation, meta-level components (rulesets) are clustered together in modules (rule-families) in which related components work together. Within a family, only one "current" ruleset is active in a particular cycle; all others in the family are inactive. This makes it easier for the model-acquisition rulesets to make the required distinctions. The complete list of meta-level rule-families is shown in table 2.

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

6.2.2 Quality training phase

An agent's model of a meta-level now contains two new modes: "quality training" and "quality operational". The quality training mode only takes place within the initial training phase, in which an agent acquires a model of its own object-level and the *training phase* of its neighbour's meta-level (because they are both in training phase). The initial training phase happens before the mutual observation phase shown in figure 4, i.e. before T1 of both agents (not shown explicitly).

At present, the observing agent only detects that certain kinds of response *exist* and that they mutually exclude each other in well-defined ways. In a future stage of our investigation, when quality-monitoring of a meta-level is planned, it will become important to evaluate whether the meta-level's response is "correct" given a deteriorated or non-deteriorated quality. To do this the observing agent must (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.

6.3 Basic attack scenario

The basic attack scenario is in two stages as follows:

1. *Disable* a meta-level component in the primary agent A (replace with dummy code)

2. After a delay: replace an object-level component in the same agent A with hostile code.

When the meta-level component 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 above. The meta-level attack may involve more than one component simultaneously and we will describe this in detail later. One or more meta-level components are replaced by "dummy" versions. Their interaction with other components appears normal but they contain no internal code; they simply take control and relinquish control in the correct way. For example, if a diagnosis component is activated, it will just give up control again after the normal delay, but will not do any diagnosis. (This is similar to camouflage techniques in Trojan horse attacks, for example, making them look like standard applications [2]).

6.4 Situations involving a quality alert

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

- `quality_alert`(*World*):
  quality problem in the external world
- `quality_alert`(*Object_level*):
  quality problem in the agent's own object-level
- `pattern_anomaly`(*Object_level*):
  pattern anomaly in the agent's own object-level
- `pattern_anomaly`(*Meta_level*):
  pattern anomaly in the neighbour agent's meta-level

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

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

1. (`quality_alert`(*World*)∨ `quality_alert`(*Object_level*))∧ `pattern_anomaly`(*Object_level*)
   Response: hypothesise that the problem is in its own object-level; try to identify and repair it.
2. (`quality_alert`(*World*)∨ `quality_alert`(*Object_level*))∧ `pattern_anomaly`(*Meta_level*)
   Possible response: hypothesise that the problem is in the backup's meta-level; try to identify and repair it.
   Excluded in the current environment because a meta-level can only cause a quality problem in the world (or in an object-level) by executing hostile code. Current attack scenarios only delete meta-level components. See also argument below.
3. (`quality_alert`(*World*)∨ `quality_alert`(*Object_level*))∧
   ¬`pattern_anomaly`(*Meta_level*)∧ ¬`pattern_anomaly`(*Object_level*)
   No response: problem cannot be diagnosed because there are no pattern anomalies which can be used to identify suspect components.
   Excluded because of recovery assumption.

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

1. `quality_alert`(*Object_level*)∨ `pattern_anomaly`(*Object_level*)
   Excluded because this scenario introduces nothing more than is already covered by an attack on the primary agent. (See also argument below).
2. `quality_alert`(*World*)∧ `pattern_anomaly`(*Meta_level*)
   Response: identify and repair the problem in primary's meta-level.
3. `quality_alert`(*World*)∧ ¬`pattern_anomaly`(*Meta_level*)∧
   ¬`pattern_anomaly`(*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 assumption 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.

6.4.1 Data recovery

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

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

6.5 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 a 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.

7 Conclusions and Future Work

The prototype have shown that it is feasible to implement distributed reflection in scenarios involving hostile code, although still in a very limited form.

One of the main limitations of this 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 agent's 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 causing a slowing of processing in one agent). Therefore, future work will investigate distributed quality evaluation using different measures of "quality" which are not dependent on precise timing.

A further limitation is that a two-agent system does not enable recovery from hostile code on a meta-level. If a hostile anomaly-detection component produces a "flood" of false-positives, there is no agent which can independently determine whether its behaviour is "correct". A minimum of three agents with majority voting is required.

References

[1] J. Bates, A. B. Loyall, and W. S. Reilly. Broad agents. In *AAAI spring symposium on integrated intelligent architectures*. American Association for Artificial Intelligence, 1991. (Repr. in SIGART BULLETIN, 2(4), Aug. 1991, pp. 38–40).

[2] Steven M. Beattie, Andrew P. Black, Crispin Cowan, Calton Pu, and Lateef P. Yang. CryptoMark: Locking the Stable door ahead of the Trojan Horse, 2000. White Paper, WireX Communications Inc.

[3] Christian Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.-C. Laprie, J.-C. Lebraud, D. Long, T. McCutcheon, and J. Mller. MAFTIA – reference model and use cases.

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

[5] D. Dasgupta and S. Forrest. Novelty-detection in time series data using ideas from immunology. In *Proceedings of the International Conference on Intelligent Systems*, Reno, Nevada, 1996.

[6] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukun. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, 1994.

[7] A. Ghosh, A. Schwartzbard, and M. Schatz. Using program behavior profiles for intrusion detection, 1999.

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

[9] V. Hilford, M. Lyu, B. Cukic, A. Jamoussi, and F. Bastani. Diversity in the software development process, 1997.

[10] C. M. Kennedy. Evolution of self-definition. In *Proceedings of the 1998 IEEE Conference on Systems, Man and Cybernetics, Invited Track on Artificial Immune Systems: Modelling and Simulation*, San Diego, USA, October 1998.

[11] C. M. Kennedy. Towards self-critical agents. *Journal of Intelligent Systems. Special Issue on Consciousness and Cognition: New Approaches*, 9, Nos. 5-6:377–405, 1999.

[12] C. M. Kennedy. 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, April 2000.

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

[14] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 253–261, New York, 5–8, 1997. ACM Press.

[15] A. Sloman. 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, 1993.

[16] A. Sloman and R. Poli. Sim_agent: A toolkit for exploring agent designs. In Joerg Mueller Mike Wooldridge and Milind Tambe, editors, *Intelligent Agents Vol II, Workshop on Agent Theories, Architectures, and Languages (ATAL-95) at IJCAI-95*, pages 392–407. Springer-Verlag, 1995.

[17] R. Turner, E. Turner, and D. Blidberg. Organization and reorganization of autonomous oceanographic sampling networks, 1996.

[18] I. Wright and A. Sloman. Minder1: An implementation of a protoemotional agent architecture. Technical Report CSRP-97-1, University of Birmingham, School of Computer Science, 1997.