Reflective Architectures for Damage Tolerant Autonomous Systems

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

Abstract

Most existing literature on reflective architectures is concerned with language interpreters and object-oriented programming methods. In contrast, there is little work on reflective architectures which enable an *autonomous system* to have these types of access to its own operation for the purpose of survival in a hostile environment. Using the principles of natural immune systems, we present an autonomous system architecture which first acquires a model of its own normal operation and then uses this model to detect and repair faults and intrusions (self/nonself discrimination in immune systems). To enable the system to repair damage in *any* part of its operation, including its monitoring and repair mechanisms, the architecture is distributed so that all components are monitored by some other component within the system. We have distributed the system in the form of mutually protecting agents which monitor and repair each other's self-protection mechanisms. This paper presents the first version of a prototype implementation in which only omission failures occur.

1 Introduction

Most existing literature on reflective architectures is concerned with language interpreters and object-oriented programming methods, e.g. a Lisp interpreter written in Lisp. Examples include 3Lisp [27] and object oriented reflection [28]. Their purpose is typically to make the internal operation of components inspectable and modifiable by a user, to enable more flexibility in implementation and experimentation with language extensions.

In contrast, there is little work on reflective architectures which enable an *autonomous system* to have these types of access to its own operation for the purpose of survival and (if possible self-repair) in a hostile environment without human intervention. Reflection of this kind is particularly necessary if the environment contains "enemies" which can damage or modify the software. An important capability is the recognition of deviations from normal patterns of activity (anomalies).

Natural immune systems may be regarded as having this kind of reflective capability in that they can distinguish between "self" and "nonself", i.e. they can recognise a "foreign" pattern (due to a virus or bacterium) as *different* from those associated with the organism itself, even if the pattern was not previously encountered. Artificial immune system algorithms [3] apply this principle to anomaly detection systems, where "nonself" may be any form of deliberate intrusion or random anomalous behaviour due to a fault.

1.1 Reflective Blindness

In the literature, artificial immune systems are not considered as complete autonomous systems. This means that vulnerabilities in the *high-level architecture* incorporating these algorithms are not addressed because the work is focused on the properties of the algorithms themselves. In contrast, we focus on the architecture level, where some of the *components* may implement simplified versions of immune system algorithms. Regardless of how powerful the algorithm is that carries out the functionality of a component, it will not help if the component itself is not protected. For example, an algorithm cannot reliably detect that its own code has been modified. In previous papers, we called this problem “reflective blindness”, see e.g. [12] and [11]. It is a major cause of “indifference”, a situation where an agent can blindly execute any code without question if its high-level decision-making rules are compromised. For a detailed discussion of this problem in an ethical scenario, see [13].

1.2 Distributed Reflection

To satisfy the requirement for autonomous recovery, we must take into account an attack against *any* part of the system, including its anomaly-detection and repair components. Our solution to this problem is to distribute the reflection so that components mutually observe and protect each other. This involves two stages: first, multiple versions of the anomaly-detection system acquire models of each other’s “normal” behaviour patterns by mutual observation in a protected environment; then they compare these models against actual patterns in a environment allowing intrusions, in order to detect and repair anomalies in each other’s behaviour. Both stages may be repeated in a way that allows increasingly accurate models of anomaly-detection and repair to be acquired while the protection from intrusions is gradually lifted.

The first part of the paper presents a conceptual framework for distributed reflective architectures. The second part presents an implementation of such a distributed architecture, showing in detail how it acquires and uses a self-model.

2 Broad-and-Shallow Methodology

Our methodology is to focus on whole architectures, instead of particular algorithms or techniques. We may characterise the architecture of an autonomous system as the pattern of interrelationships (causal connections etc.) between entities representing specialist functions (e.g. they may be layers, individual components or sets of functionally similar components). Each entity may be regarded as a “slot” into which a specialist technique can be plugged in (e.g. a human interface layer or a planning specialist).

When exploring whole architectures, these slots must be “shallow”; otherwise the problem becomes unmanageable. In practice, this means that a slot contains only a minimal implementation of a technique. We will see later that this methodology leads to surprising conclusions about what algorithms are actually required by the architecture slots. More detailed arguments for this “broad and shallow” approach are given in [1].

We adopt the “design-based” approach explained in [21] which effectively means that we improve our understanding of a phenomenon by attempting to build it. The design-based approach involves the exploring of “design-space” and “niche-space” (that is, the space of requirements) as outlined in [25] and [22]. Our specific way of doing this is first to define a sufficiently challenging environment and then to find an architecture which enables survival in that environment.

The results of the investigation will be a statement of whether or not a particular architecture satisfies a particular set of requirements. If the requirements were satisfied, we wish to understand the limitations of the architecture, what assumptions it depends on and whether it scales up. If the requirements could not be satisfied, we wish to understand why not.

3 Control systems

An autonomous system may be regarded as a control system if it maintains the world in a desirable state and ensures critical requirements are not violated. In the simplest case the system is homeostatic (in the same way that a thermostat does not allow temperature “requirements” to be violated).

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

Such a "homeostatic" system is very brittle, since an attacker may first disable the part of the software which ensures that user rights are not violated. Although the system may *detect* that the quality of the world deteriorates, it cannot diagnose the problem or take any autonomous action to correct it.

We propose that the problem should be addressed as follows:

1. The status of the system's software should be included as part of the world which it is observing, i.e. the system should be *reflective*.

2. The system's capability to continue operating is also a required state of the world and should be preserved in the same way as user rights should be preserved.

This two-layered architecture is inspired by autopoiesis theory [16] (page 13) and Second Order Cybernetics [32]. The precise content of the desirable states for self-protection is not expected to be externally specified because they refer to internal states of the software and it is unlikely that these states could be known in advance by a user, unless the software is trivial. We assume that the software is non-trivial. This means that *acquisition* of the model of normal behaviour is a central part of the problem, and will be discussed in detail in this paper. This acquisition has similarities to data mining. For convenience, we call the internal state of the system its *internal world*, in contrast to its external world.

A self-protective system should defend its software and internal data from unauthorised changes and ensure that it has sufficient computational resources to do its task. Similarly, it should monitor how it uses these resources: does it spend most time and resources on the most relevant aspects of its problem domain? E.g. it should recognise when it is being swamped with meaningless data and false alarms which "distract" it from doing productive work. This is related to the concept of "meta-management", see e.g. [24].

3.1 Agents as building blocks

Figure 1(a) shows an agent with two layers labelled "meta-level" and "object-level" respectively. We call the multi-level entity an agent because its internal structure is a special case of the agent concept supported by the SIM_AGENT toolkit [26] which we use for simulating environments and exploring architectures. The object-level is a simple control system C_E for an external world.

If we ignore the meta-level, we can initially consider the agent to be C_E only. We assume that C_E has a model M_E of the external world which it uses to maintain the environment within acceptable values. We assume further that M_E is represented in a symbolic declarative form (e.g. rules) and enables the agent to predict the outcome of different actions. The agent selects its next action according to the quality of the predicted state. Thus M_E allows the "simulation" of hypothetical states ("what-if" reasoning) which is often called "deliberative" processing (see e.g. [23]), in contrast to reactive processing which involves immediate reactions to the external world without the use of a predictive model.

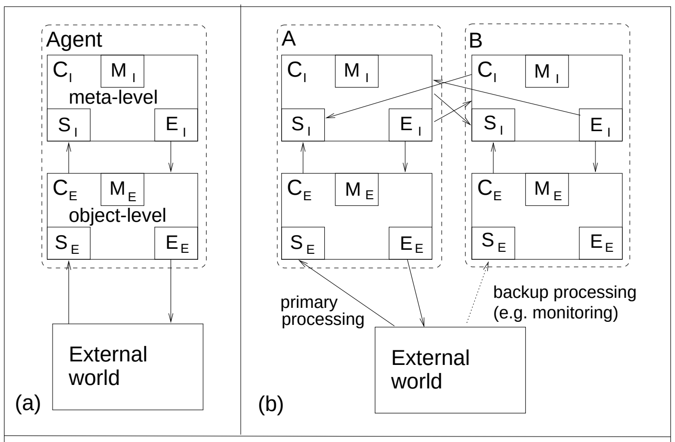
The simplest way to introduce reflection (and hence self-protection) is to add a meta-level as shown in the top level of the agent in figure 1(a). The meta-level is like a second control system which is applied to the agent's internal world (i.e. aspects of its own execution) to maintain its required states (hence the label C_I). In the simplest situation the model M_I at this level only predicts that the internal world will remain "normal". Note that sensors and effectors are also used on this level (S_I and E_I).

Since we are assuming that hand-coding of "normal" patterns of software execution is unrealistic, M_I is autonomously acquired by C_I during a protected training phase.

3.2 Distributed reflection as a multi-agent system

A fundamental weakness of the system in figure 1(a) is that it cannot easily monitor the status of its reflective capability (as this apparently requires an infinite tower of meta-levels). For example, if its meta-level is prevented from executing, there is nothing within the system itself that can detect this.

To work around the reflective blindness problem, we have implemented an architecture using a *distributed* form of reflection, where the first agent's functionality is produced collectively by two or more mutually protective agents. The whole control system then becomes a multi-agent system.



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

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

Figure 1: Reflective agent architectures

Figure 1(b) shows this configuration for two agents (labelled A and B). In the simplest configuration, the agents use copies of the same code, and only one is "in control", i.e. it is responsible for maintaining the external environment. (In figure 1(b), B does not act on the world but only senses it; it may take over control if it detects an anomaly in the operation of A).

The agents mutually acquire models (M_I) of each other and subsequently use their models to observe each other and detect deviations from normal execution patterns (anomalies). This is not expected to eliminate all forms of reflective blindness; rather it gives the whole control system *sufficient* reflective coverage, in that it enables monitoring and repair of any components necessary for survival in a particular type of hostile environment described below. (See also [14] which introduces the related concept of "reflective self-sufficiency").

It should be mentioned that Figure 1(b) is only one form of distributed reflection. We are not maintaining that this is the only possible configuration. As in many software engineering situations, some design decisions are often made to enable ease of implementation and not because they are fundamentally the "best" or only design.

Similarly, we do not restrict the role of "meta-level" to that of control of the internal environment, as it could sometimes involve aspects of the external environment. Instead, the key idea is that of *trust*. If a component plays the role of meta-level within an agent A_i, then its code is *trusted* by that agent, since it is relying on this code to monitor its own object level and the meta-level of the other agent. This same code (meta-level of A_i) plays the role of object-level for the other agent A_j, meaning that it is *not* trusted by that agent, but is continually observed and evaluated.

3.2.1 Meta-level towers do not work

It may be argued that distributed reflection is overly complex, and simpler solutions cannot be ruled out. The simplest apparent solution would be the addition of new "layers" to trap the intrusions that lower ones have missed, and to detect any problems in the layers themselves. This might be similar to a kind of "reflective tower" of monitoring levels, where there is *no* mutual observation between meta-levels. Instead there would be a simple monitoring hierarchy from a higher meta-level to the next lower one.

A tower of levels does not help with autonomous recovery from intrusions, however, because the top level can be disabled without any other part of the system detecting this. The attacker then has an arbitrary amount of time to disable the next level down the hierarchy (which again will not be detected) and so on, until all levels are disabled. Even worse, an attacker could *modify* the code of the topmost level so that it disables or misleads all other levels. The enemy only needs to attack one component. (Note that this is not the same type of "reflective tower" as that in reflective languages such as 3Lisp. The term "meta-level" is used in a different sense in these systems, namely as a layer which "implements" another one, which is not the same as monitoring of a separately running process and detecting anomalies in it).

4 Designing a Hostile Environment

Reflection of the type discussed here serves the purpose of survival in a hostile environment. We define *survival* as recovery from faults and intrusions without human intervention and restoring the environment to an acceptable quality.

A partially known environment is one in which events can occur which are not taken into account by the agent's model of the world (based on current knowledge about it) and may include situations which the agent was not explicitly designed to handle. We call these events "anomalies". In particular, we assume that the agent has no knowledge of "enemies".

We define a *hostile environment* as one in which the agent's executive and control systems (*including* any anomaly-detection and self-repair mechanisms) can be disrupted or directly attacked. No system is invulnerable; there are environments where self-protection would be impossible, for example if the interference happens too fast for the system to react, or if it has no appropriate sensors. We assume that this is not the case. The general questions are: *What kind of reflective architectures can (or cannot) enable an agent to increase its chances of surviving in what kind of environment? Why (or why not)?*

We imagine there is an "enemy" which can interfere destructively in any of the following ways:

1. *Direct modification* of a system's control software by deleting, corrupting or otherwise modifying the code.

2. *Weakness exploitation*: present it with a situation that its software cannot cope with.

3. *Resource blocking*: prevent it from achieving its goal by stealing, blocking or diverting its computational resources (e.g. denial of service attacks).

Deception is mostly covered by (1) and (2) above. The simplest example is direct modification of sensor operation so that they give false readings. It is possible that all these types of interference can be detected as anomalies in external or internal sensors.

4.1 Treasure: A simple control scenario

It is useful at this point to introduce a simple virtual "external world", which we initially use for rapid prototyping of architectures. This scenario, called "Treasure" is based on the "Minder" scenario [33] and a configuration of it is shown in figure 2. The world is made up of several treasure stores, one or

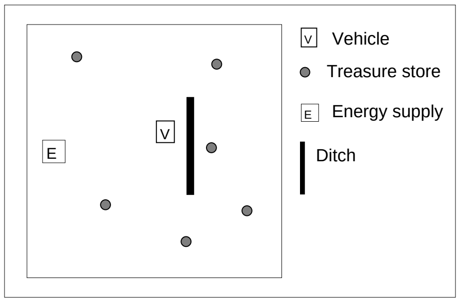


Figure 2: Treasure Scenario

more ditches and an energy supply. An autonomous vehicle V must collect treasure, while avoiding any ditches and ensuring that its energy supply is regularly restored. The "value" of the treasure collected should be maximised and must be above 0. Collected treasure continually loses value as it gets less "interesting". Treasure stores that have not been visited recently are more interesting (and thus will add more value) than those just visited. The system "dies" if any of the critical requirements are violated.

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

In the Treasure scenario, distributed reflection means that control of the vehicle is shared between two agents A and B. Figure 1(b) shows the situation where A is in control. More complex configurations are possible, in which agents are specialists.

4.2 Simulating an Enemy

Normally an autonomous system (whether multi-agent or not) has no difficulty surviving in the Treasure environment as defined above. To simulate an enemy, we introduce an additional agent which attacks the software in various ways as listed above.

Thus in addition to control of its external world, the autonomous system must also maintain its "internal" world (i.e. its software) in an acceptable state. To do this it must first detect an anomaly in its software execution patterns, diagnose it and either repair it directly or find some other recovery action.

4.3 Internal sensors and effectors

The meta-levels in figure 1 operate on an internal world using so-called internal sensors and effectors. To define these, we must describe the implementation which is based on the SIM_AGENT package [26]. In SIM_AGENT, an agent execution is a sequence of sense-decide-act cycles which may be called *agent-cycles*. Each agent is run concurrently with other agents by a scheduler, whose execution is also divided up into cycles. In one scheduler cycle, each agent is allocated a time-slice during which the agent executes a discrete number of its agent-cycles (may be 0). For simplicity we assume that agents are run at the same "speed" and that exactly one agent-cycle is executed in one time-slice. This means that a scheduler cycle-number will correspond to an agent cycle-number (and also the agent's time-slice number). We will normally use the term "cycle" to mean an "agent-cycle" unless otherwise specified.

Each agent architecture is encoded in a set of rules (called a rulesystem) which is divided up into modules (rulesets). Rules are generally either reactive or deductive. Selected modules of a single agent are shown schematically in table 1 (for simplicity, many components have been omitted, e.g. interpretation and evaluation of sensory data). Note that the meta-level is also included as a set of rules (to be defined later).

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

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

In other words, agents can observe and repair each other's self-observation and self-repair processes. In the two-agent case, there are limits to how far an agent can determine how well it is being monitored

Table 1: Selected architecture modules

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

or repaired by its neighbour (as this would need a third "neutral" agent), but we will not consider this situation at present.

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

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

4.3.1 Terminology

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

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

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

1. a whole ruleset - identified by the ruleset name.
2. a rule within a ruleset - identified by a (ruleset, rule) pair

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

An event record is also a declarative statement which uniquely identifies a component, i.e. a (ruleset, rule) pair, along with what happened to it. An example of an event record is: "Rule R of ruleset S of observed agent A had its conditions checked at cycle T". Consequently a whole trace for one agent-cycle is a conjunction of statements that are true for that cycle (usually the last observed cycle). This allows high-level reasoning and diagnosis as well as the kind of statistical pattern-matching used in existing artificial immune systems such as in [6].

4.4 A Random Failure Environment

Initially, we choose an environment where randomly selected components are disabled at random intervals, and where the minimum interval between attacks is long enough to allow recovery if a problem is detected. The agent which simulates the "enemy" selects a target agent at random (either A or B) and deletes a randomly selected component (in this case a rule). More challenging environments may include situations where rogue software is inserted, or where computational resources are diverted.

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

Furthermore, if we assume that the system has no intrusion detectors at all, and can only detect absences of normal events in its own operation then this is actually a more difficult problem than detection of the actual intrusion, since the damage will already have happened before it is detected. This is also an effective test of distributed reflection, since an anomaly-detection or repair component may have been disabled, and this must still be detected and repaired by the system itself.

5 A Working Prototype

We now describe the first results of a proof-of-concept implementation of the two-agent distributed architecture for the random failure environment above. The motivation was to show that agents can build models of each other's meta-levels and use these models to survive in the restricted environment.

5.1 Artificial Immune System Methods

In accordance with the "broad and shallow" approach, we apply a scaled-down version of artificial immune system methods to our architecture. Such an immune system component may act as the meta-level in figure 1(a) because this agent's model of "self" (which we call M_I) is acquired during a training phase. The self-model is a signature of "normal" rule execution events.

In the artificial immune systems literature, two basic approaches are relevant: signature-based intrusion detection, Forrest et. al. [6] and negative selection, Dasgupta and Forrest [4].

In signature-based approaches, a database of "normal" patterns is constructed during the training phase, which may be obtained by repeatedly observing the system while it runs under normal conditions, protected from intruders. This becomes the model of normal operation. The patterns are typically in the form of binary encoded strings which hold a record of a particular action or sequence of actions, e.g. Unix system call sequences [7]. In our implementation, an event record can represent one such pattern.

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

The negative selection does not detect absences of patterns associated with the normal functioning of components, as would be ideal for our random failure environment. Therefore we implemented a simple version of the signature-based immune algorithm to deal specifically with absences caused by unauthorised disabling of components (i.e. rules). In particular, we addressed the following question:

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

The assumption of ensuring that critical components are active on every cycle is reasonable because similar techniques are used in fault-tolerant software systems, e.g. message logging and checkpointing). For our purposes, we define the critical components to be the meta-level rules implementing the immune system itself (shown schematically in table 1).

Although a critical component can be designed to be active on each cycle, we do not know how this activity will register on an observing agent's sensors because of our assumption that the precise sequences of events and their frequencies are not known in advance. Nor do we know which events (in the whole event space) actually occur in the execution. It is possible that some software components (rules, or even whole rulesets) are never active.

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

5.2 Training and operational phases

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

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

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

To detect more subtle anomalies (such as modification or resource diversion), it is possible to detect changes in the overall trace, while tolerating noise. There will be some mismatches for detectors whose weights are less than the total number of cycles (as activity in each cycle varies). The simplest method is to define a threshold above which the weighted sum of mismatches would be considered as an anomaly. This is the method used in Forrest et. al. [6], except that they do not use weights.

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

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

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

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

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

Note that a very simplified form of negative selection would use the set N instead of P, since an anomaly would be any activity which was never detected in the training phase. We have not included this in the initial implementation because we are focusing on *absences*.

5.3 Design of the Training Phase

In accordance with the meta-level configuration in Figure 1(b), our first building block is a single agent system which can build a model of its its own object-level and use this model to survive. The end result is a working meta-level (immune system) which can be observed in operation.

In the next stage, both agents must acquire models of each other's meta-levels. While their immune systems are both in training phase, they can acquire models of their own object-levels but they cannot observe the operational phases of each other's immune systems. Hence, as soon as they enter operational phase, they will find each other's immune patterns to be "foreign" (failure to tolerate "self" in immune system terminology). In our current scenario, this is a failure to tolerate absences of the training phase events, which is the same problem in principle.

We solve the problem by allowing agents to "take turns" observing each other in different states. To explain our solution in detail, we introduce some terminological conventions.

5.3.1 Terminology

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

To show how an agent acquires a model of another one, the following naming convention is useful: the agent whose observation process we are currently describing is labelled r, while the agent it is observing is labelled d (for observe$r$ and observe$d$ respectively).

5.3.2 Temporal diversity

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

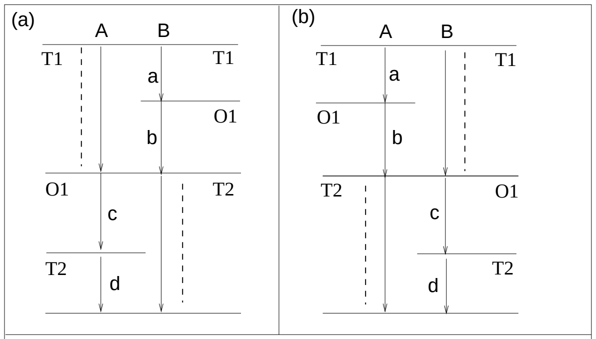
In natural evolution and learning, this diversity is provided by massive asynchronous parallelism. On a sequential processor, the same effect may be approximated by a sequence of interleaved phases during which agents observe each other in different modes. This is, however, an improvisation, and the method does not scale up easily for a larger number of agents.

5.3.3 Temporal constraints

To ensure that each agent gets a chance to observe the other in both modes, we use temporal constraints to enable the transitions between modes to happen at different times for each agent. An observed agent d should make a transition between training and operational modes while its observer r is in training mode. The roles are then reversed: the agent that was d plays the role of r and vice versa.

Figure 3 shows two ordering possibilities (depending on whether A or B is first to observe the other's transition). Each agent's development phase is shown as a vertical line and is divided into training phase 1 (T1), interim operational phase (O1) and training phase 2 (T2). The constraints are as follows: (1) if T1 of A ends before T1 of B then O1 of A should end before O1 of B; (2) phase durations experienced by each observing agent (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. During the interim operational phase (O1) of any observed agent d, the fault insertion agent is permitted to place faults in its object-level so that its "normal" reaction to an anomaly (including its self-repair) can be observed by the agent in training phase r. Thus, the anomaly-detection and self-repair process of d can be incorporated into r's model.

Formal temporal constraint networks [5] as applied to autonomous control systems, e.g. [17] may be relevant in future extensions to the current implementation.



Key: Vertical lines = time lines for agents A and B;
Horizontal lines = phase transitions (new phase under line);
Dashed line segments = observations of phase transitions
T1 = training phase 1; O1 = interim operational phase
T2 = training phase 2; a,b,c,d = observed phase durations

(a) A observes B's transition first; (b) B observes A's transition first

Figure 3: Temporal constraints

5.3.4 Limits of mutual observation

The observation (and hence the resulting model) is not expected to cover *everything.* For example, the training phase now has an additional class-discovery subphase, which is not recognised by r as being different from "normal" training phase. Similarly there are smaller fluctuations associated with transition detection itself. Recognition of these minor distinctions was not important for survival in our scenario, however. In more complex scenarios it may be necessary to refine the training phase in order to achieve a closer approximation to reality. "Life-long learning" may also be useful.

An additional problem is that r can only observe the meta-level of d as it responds to problems in its own object-level; it cannot observe d's detection and response to problems in r's meta-level. To do this would require a three agent configuration for which the above mutual observation scheme may be inefficient. Before we consider this, however, it is first necessary to explore the limitations of a two-agent system. In particular if the same meta-level code is used by an agent to observe both its own object-level and the other agent's meta-level, then most problems may be detectable by observing its handling of anomalies in its own object-level, which is already done by the above scheme.

6 Model Acquisition

We now present a model acquisition method which was specifically developed to build the prototype architecture. In accordance with the "broad-and-shallow" methodology, we have chosen initially to build the model on the basis of simple and high-level discontinuities in the observed execution patterns. Later, this may be refined by taking noise and uncertainty into account (using weights and more general pattern matching). The focus on sharp distinctions is similar to the idea of selecting features with maximum "information content" used in many existing concept-learning and data-mining algorithms, for example [18], [19] and [15].

6.1 Transition Detection and Class Discovery

One important piece of information that the observing agent r can obtain is that d makes fairly sudden *transitions* from one mode to another. This involves large numbers of events stopping and new ones starting. We believe this information should be included explicitly in the model to reflect the actual discontinuity between the different modes and the fact that the modes themselves map onto collections of active components in the software.

The agent playing the role of r can detect that d makes a sudden transition to operational mode if a sufficient number of known events (in P) stop occurring and a sufficient number of new ones suddenly occur. This transition is the first clue that r gets that there might be two different classes of behaviour.

All stopped and started events are hypothesised to belong to a class of events which are all associated with a particular activity (e.g. immune system operation).

The new class is then partitioned into two subclasses, each containing the stopped and started events respectively. The subclasses are hypothesised to be mutually exclusive *modes* of the activity. The hypothesis may be refuted later, e.g. if the new events do not persist. A label for the new activity type (along with its modes) is generated in the evolving model. At present, the system generates two activity types:

Type 1: immune-related activity: either training or operational.
Type 2: anomaly-related activity: either it detects an anomaly or it does not.

Type 2 is a subtype of operational mode. Both types exist on different levels of a tree-structure (defined in later sections).

Due to the specific structure of the software, the two classes were sufficient to enable the system to survive indefinitely in the restricted random failure environment; it detected all attacks on any agent's meta-level (no false negatives) and was able to repair the damage in each case. The simplicity of the model merely produced occasional false positives which had no serious consequences.

If there is a false positive, the execution pattern of the observed repair components will sometimes unusual (because the observed agent finds that a repair was not necessary). The effect of this is another false positive immediately afterwards by the observing agent. The system recovers from these false positives autonomously because an agent detects that it is repeatedly reacting to a false alarm. A more elegant solution is to include false alarms in the training phase (in addition to real anomalies). Then a third activity type, "repair-related activity" can be generated with modes corresponding to "repair required (normal)" and "no repair required (false alarm)".

6.1.1 Mutual exclusion

There will be some events associated with one mode that *always* occur when the agent is in that mode, and some other events that *never* occur in that mode but are always observed in the other mode. Thus the agent can hypothesise an XOR relationship: *either* one set of events are observed together (with none missing) *or* a different set is observed. Members of two sets are never observed together; they represent two different contexts. This is useful information because it provides a microcosm of the whole signature (where we have the P, T and N sets). Thus provided we know the current mode, we know what the context-dependent versions of P, T and N are, and therefore we know what normally *ought* to happen (the P set) in a particular context (in this case the mode), and what should *never* happen (the N set) for that context. This enables context-sensitive anomaly-detection.

6.1.2 Top-down approach to model acquisition

We have chosen initially to develop a high-level model based on simple logical distinctions (such as XOR) in accordance with the "broad and shallow" principles mentioned above. Further reasons for using this top-down approach are as follows:

- The nature of the scenario and our requirements: we want to *guarantee* that critical parts of the meta-level are protected. Making the model include strict boolean requirements (such as XOR) makes such guarantees more feasible.

- Ease of diagnosis: we know which events are missing from what mode. This gives a good idea of which component is faulty (because an event record contains the component identity). Self-diagnosis and repair are important for autonomy.

- Transparency: a conceptual model with sharp distinctions is easier to understand.

- Flexibility: it may be easier to dynamically focus resources (at run-time) on protecting those parts of the software which turn out to be essential. This may not always be known at design time.

6.2 Generating contexts

The modes are intended to be contexts; they are like sets of statements which are true (about a particular type of activity) at a certain time (events that are occurring) and things with are not true (events that are not occurring). Many important kinds of anomaly can then be detected as things which are true in the world but the model says that they should not be true in that context (a physical anomaly such as anti-gravity would be an extreme example, e.g. something does not fall when it is dropped).

As an everyday example, a person who is observed painting for some time puts away the painting materials and starts reading a book instead. Within the larger context of sitting at a desk a transition takes place: some patterns stay the same (the person remains sitting and attending to something on the desk) and some patterns stop appearing (painting) while new ones begin (holding an open book). The person is never seen to paint on the pages of the book. Therefore the observer begins to assume that they never happen together (hypothesis of mutual exclusion was not refuted). We can say that the actions of painting are "prohibited" when the object on the desk is an open book (current context). They would be regarded as anomalous if they happened.

6.3 Distinction levels

When an initial transition occurs, the observer in the above example could postulate that something has fundamentally changed and provisionally call it "kind of work at a desk" (activity type). The new distinction then involves two levels:

1. "Things which change" (activity-type) vs. "things which stay the same" (remainder of "parent" context, in the above example, sitting at a desk); this is like separating an object from its background.

2. "Old patterns" vs. "new patterns" (content of the two modes).

It may seem overly complex to use two levels of distinction when possibly one would do. The two levels are necessary because we wish to identify those groups of events that mutually exclude each other (so that a local version of P, N and T is possible). Hence we must separate those groups of events that have this clear structure from those background events that are not part of it.

The new mode is experienced as a *negation* of the old one; we have already experienced the old mode, but all we know about the new one is that it is different (e.g. it may contain further submodes). In the same way, the transition to the mode of e.g. "responding to a real anomaly" is experienced as a transition from "normal" operational mode to "not normal operational mode" (i.e. every other kind of operational mode). Therefore it is simplest to include only two modes initially, although this could be extended.

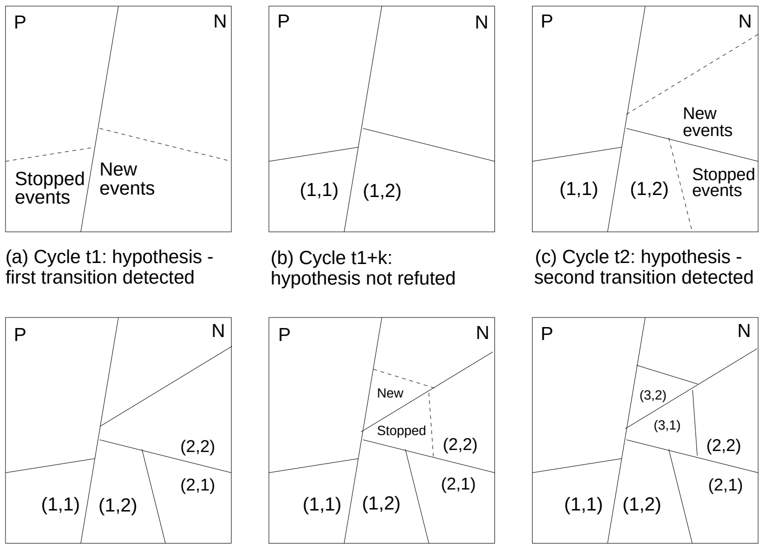
6.4 Knowledge growth of the observing agent

We now consider the evolution of the agent's knowledge with time. Here there are also two levels which should be kept separate:

1. *Internal change*: How does the agent's knowledge change?

2. *External change*: How does the world change? The question is answered using concepts acquired in level 1 above. For example, the current mode of an activity may oscillate between "normal" and "anomaly-detected", but this only has meaning after the discovery of the activity type with "normal" and "anomaly-detected" modes.

We first look at internal changes within the agent. The time evolution of its knowledge can be shown schematically as a series of snapshots in figure 4. We assume that the initial training phase has completed and the global P, T and N sets have stabilised. At this point the observed agent makes a transition to operational phase. The times (cycle numbers) at which transitions to unknown states are encountered can be listed as $t_1, t_2, ...$ etc. where t_1 is the time of the first transition, t_2 the time of the second etc. Note that we are talking here about detections of new transition *types*. Transitions returning to previous modes are not included; neither are transitions which are not detected by an



(a) Cycle t1: hypothesis - first transition detected

(b) Cycle t1+k: hypothesis not refuted

(c) Cycle t2: hypothesis - second transition detected

(d) Cycle t2+k: hypothesis not refuted

(e) Cycle t3: hypothesis - third transition detected

(f) Cycle t3+k: hypothesis not refuted

Key: Numbers in backets (i,j) = (activity-type, mode)
(1,1) = (immune-related, training); (1,2) = (immune-related, operational)
(2,1) = (anomaly-related, normal); (2,2) = (anomaly-related, anomaly-detected)
(3,1) = (repair-related, normal); (3,2) = (repair-related, false-alarm)

Figure 4: Partitioning the event space: a series of snapshots

observing agent in training mode. For example, a transition back to training mode of an agent that has already been observed changing to operational mode is not associated with any new type of activity.

For the purposes of building a prototype, we are assuming that a mode is "reachable" only from one type of activity. For example, a transition to the "anomaly-detected" mode can only happen when the agent is checking for an anomaly (i.e. as part of "anomaly-related activity"). Generalisations of this are discussed later in section 6.10.

Figure 4(a) shows the state of the agent's knowledge when it encounters the first transition from training to operational mode at time t_1. The box is the universal set of events that can be detected by the agent's sensors during one cycle (i.e. everything that can possibly occur in a single trace). The global P and N sets are shown as partitions of this set. For simplicity, the diagram does not include the initial T set (resulting from the initial training phase) or its subsequent states.

When the transition is detected at time t_1, the stopped events are a subset of P and the new events a subset of N. They are provisionally allocated to two different modes shown as dotted lines. This means that a hypothesis of a new activity with these modes is generated. The *currently* active mode (containing events which are currently appearing in the trace) is assumed to be the second mode. The new partitions are allocated to T (because both partitions are now "tolerated"). The T set at any point in time is simply the union of all the partitions that are neither P nor N (not shown in these diagrams).

Figure 4(b) shows the state after some time k. The lines outlining the new class have "solidified" meaning that the mutually exclusive nature of the events has persisted.

6.4.1 Indices represent distinction levels

The new partitions are labelled with pairs of indices $(1,1)$ and $(1,2)$. In all the partitions, the two numbers in brackets (i,j) are indices for the two distinction levels introduced earlier. The index i points to the subset of events which were involved in the change (the activity), in contrast to "non-activity" (things which remain the same), i.e. the background of the change. This background may include some events in the currently active parent mode, as is always the case in figure 4, i.e. the "stopped" mode of the new activity is always a proper subset of the parent mode. For example, checking for anomalies will happen in both submodes of operational mode (whether or not anomalies actually occur).

The first number i is called the *activity index* assigned in chronological order whenever a new pair of mutually exclusive sets are discovered. The union of these sets is the *activity set*. The *mode set* for a mode j of activity i is the set of all events that happen in that mode. We use the convention of labelling the mode set as (i, j).

6.4.2 Knowledge refinement and retraction

If some pairs of events allocated to modes 1 and 2 are both present or both absent in the same cycle then these events are removed from the partitions. If there are two few events left in the partitions (determined by a threshold), then the hypothesis is refuted that a kind of activity exists with mutually exclusive modes involving a sufficient number of events.

Figure 4(c) and (d) are a repeat of the same type of event transition at a later time t_2. The new activity type is given index 2. Figure 4(e) and (f) show two further submodes (3, 1) and (3, 2) (for "normal repair mode" and "false alarm mode" respectively) whose discovery was not required by the current implementation. (Note that the transition-detection for this new mode would also be more complex, since the parent mode "anomaly-detected" is not continuous but intermittent).

6.5 Currently active modes

We now address the second level of time evolution, which is about the *use* of the model to keep track of mode changes, a problem which becomes important in the operational phase.

An alternative representation of partitioning is shown in the tree structure in figure 5 which shows an additional "optional" activity type (occasional increased alertness). The tree represents a particular viewpoint of software modes. We consider the situation in which a single observing agent has one such viewpoint.

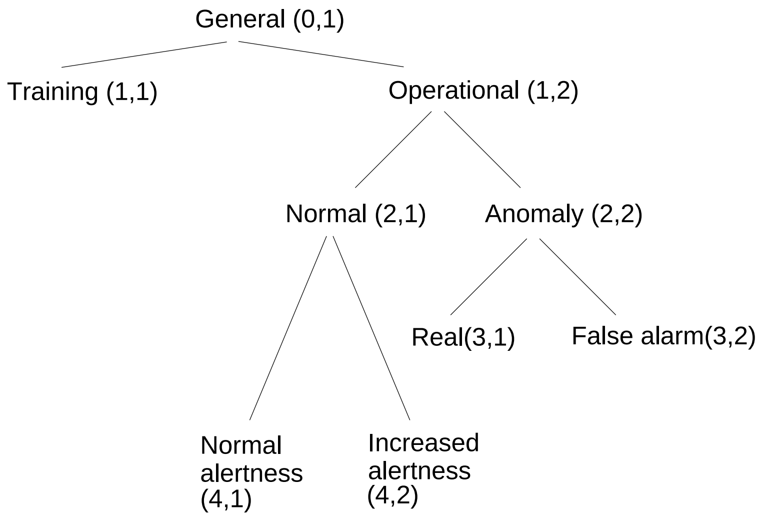


Figure 5: Tree diagram of successive partitions

A path through the tree's levels represents a list of modes which is active at any instant of time. E.g. if the observed agent modelled by figure 5 is currently in the "increased alertness" mode then a total of three modes $(1, 2), (2, 1), (4, 2)$ must be active during that cycle, along with the constant root "mode" $(0, 1)$ corresponding to the P set. The active modes are listed as facts in the database. If the observed agent were to change to training mode, then all submodes of operational mode become inactive.

6.6 Definition of mode signatures

Figure 6 is the same as figure 5 with the addition of the "local" (i.e. context-dependent) P and T sets for each mode. Both these sets represent a mode's signature and can be treated in the same way as the global signature (the whole tree). At each branching point, the branches are shown emanating from the T set of each parent mode. This is to show that the union of modes at the new level is a subset of T at the parent level. E.g. processing of real anomalies is a subset of everything that *sometimes happens* in operational mode. We normally expect that there will be some events in a T set that do not belong to either mode. This may be called the *unstructured* partition of the growing T set, corresponding to the "background" introduced earlier. E.g. there may be occasional actions in operational mode, independently of whether the agent is reacting to an anomaly or not. We can define

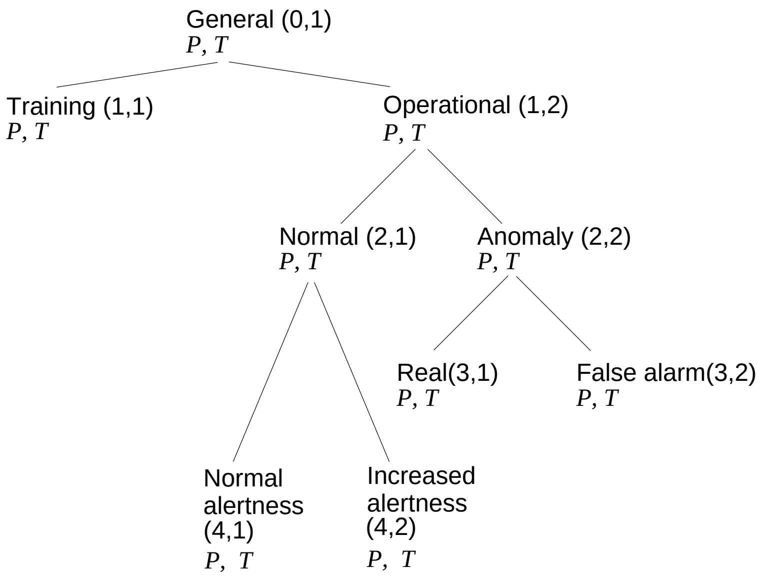


Figure 6: Tree diagram showing context-dependent P and T sets

$P_{i,j}$ as the set of events that always occur when activity type i is in mode j, but never when it is *not* in mode j. This means that if the current mode is 1 then $P_{i,1}$ is the local P set and $P_{i,2}$ is the local N set. Similarly, $T_{i,j}$ is the set of events that sometimes occur in mode j but never in the opposite mode. Depending on the parameters selected to determine whether something is a transition, one of the modes may be empty (e.g. if a large number of new events suddenly occur). The current software assumes that they are non-empty, however.

At level 0, the two "modes" are agent/non-agent (or self/nonself from the observing agent's viewpoint). This means that the set N is defined as $P_{0,2} \cup T_{0,2}$. By convention, $T_{0,2}$ is $\{\}$, since the agent does not need to distinguish between different types of nonself strings (at least in the current implementation).

In accordance with the top-down principles of model-acquisition, we initially only use the P set for anomaly detection. This was sufficient for survival in the selected environment because of our assumption that critical components can be made to "broadcast" the fact that they are alive, thus producing "essential" event records on every execution cycle. The T sets play an *implicit* role only in the implementation; they are not explicitly represented as evolving data structures. We can now define the following objects:

- A set of strings S corresponding to all possible events detectable by an agent's internal sensors. A string is an event record as it appears in the trace and also as it appears in the event field of a detector.
- A set of activity indices $A = \{0, 1, 2, ...\}$ where 0 corresponds to the activity of the whole agent and is always assumed to be in mode 1 by convention.
- A set of mode indices $M = \{1, 2\}$.

S contains subsets $P_{i,j}$ and $T_{i,j}$ which are created and evolve with time. The *parent* mode of mode (i, j) is the mode whose T set contains the mode set of (i, j). We can define a function $Mode\_set$:

$A \times M \longrightarrow \wp(S)$ which returns the mode set of (i, j) as follows:

$$Mode\_set(i, j) = P_{i,j} \cup T_{i,j}$$

The function $Act\_set : \wp(S) \longrightarrow \wp(S)$ which returns the activity set is defined as:

$$Act\_set(T_{i,j}) = (P_{k,1} \cup T_{k,1}) \cup (P_{k,2} \cup T_{k,2})$$

where k is the index of the activity set (in contrast to i which is the activity index of the parent set. If the parent set is the root then i is 0). The background set is the unstructured part of $T_{i,j}$. $Bck\_set(T_{i,j}) = T_{i,j} - Act\_set(T_{i,j})$

The allocation of a label (i, j) is analogous to the naming of a new concept. We have chosen to use numerical indices instead of symbolic names because there is no guarantee that an activity type will actually correspond to an existing named human concept, although some degree of correspondence is desirable. The indices are important in the formal characterisation because they serve as symbolic names for the agent (but not necessarily for the human reader). The signature pointed to by a label (i, j) is a primitive form of "grounding" for that label (roughly in the sense of [20] and [8]).

6.7 Formal characterisation of knowledge growth

The series of snapshots in figure 4 gives only a schematic view of the agent's knowledge growth. To make this more precise, we use the above definitions to formally characterise the time evolution of the agent's knowledge. We first define some additional objects and functions which are time-related:

Time-related objects:

- The set of all scheduler cycles C for which the observed agent's components were active. Each cycle is a time slice in which the observed agent's rules are run at least once. For our implementation this is the same as the natural numbers $\mathbb{N} = \{1, 2, 3, ...\}$, but we use a different label for clarity and also to allow for situations where the observed agent is not intended to run on every cycle as counted by the observing agent. (e.g. once every two cycles would give $C = \{2, 4, 6, ...\}$).
- The growing set of *discovery* timepoints $D = \{d_1, d_2, d_3, ...\}$. This is a subset of C. Each one is associated with an activity type index. They are cycle numbers at which a new type of activity was discovered.

Each d_i (where $i \geq 1$) is the cycle-number (timepoint) at which a particular type of transition was *initially* encountered and which subsequently led to the creation of a new activity type. These initial transitions are the points in time at which the observing agent's *knowledge* changed significantly (although for some time after the transition, this knowledge is only in the form of a tentative hypothesis). E.g. $D = \{150, 275, 316\}$ could mean that at cycle 150, operational mode events first occurred during this particular observation period; at cycle 275 anomaly-detection events were first encountered; at cycle 316, the reaction to a false alarm was first encountered etc. In this way, the timepoints show the historical discontinuities in the agent's knowledge growth.

Since we are constructing a minimal prototype, there is a considerable simplification here. In the real world a transition of this type is not expected to happen in a single instant but will more likely be spread over a range of cycles. This simplification does not affect the overall argument, however.

Time-related functions:

- $Pstate : A \times M \times C \longrightarrow \wp(S)$.
  $Tstate : A \times M \times C \longrightarrow \wp(S)$.
  $Pstate(i, j, t)$, $Tstate(i, j, t)$ return the state of the P and T sets for mode (i, j) respectively at time t, that is $P_{i,j}$ and $T_{i,j}$ at time t. If the agent has not yet generated these sets, the returned value is $\{\}$.
- $Stopped : A \times M \times C \longrightarrow \wp(S)$.
  $Stopped(i, j, t)$ returns the subset of $Pstate(i, j, t-1)$ which stopped at t.
- $New : C \longrightarrow \wp(S)$.
  $New(t)$ returns new events which started at t.

6.7.1 A simplified world

For more clarity it is useful to show how these functions behave in a simplified world where the event records are the following strings: $S = \{\texttt{AA}, \texttt{BB}, \texttt{CC}, ..., \texttt{ZZ}, \texttt{AAA}, ...\}$. We suppose that the occurrence of these strings undergoes transitions in the same way as in the world of real software execution traces. We assume that initial training is complete and the first transition is encountered at cycle t. Using the simplified alphabet, we list some example values for the above functions just before the transition as follows:

- $Pstate(0, 2, t-1) = \{\texttt{II}, \texttt{JJ}, \texttt{KK}, \texttt{LL}, \texttt{MM}, \texttt{NN}, \texttt{OO}, \texttt{PP}\}$: none of these strings ever occurred in time period $1, ..., t-1$;
- $Tstate(0, 2, t-1) = \{\}$ (by convention);
- $Pstate(0, 1, t-1) = \{\texttt{AA}, \texttt{BB}, \texttt{CC}, \texttt{DD}, \texttt{EE}, \texttt{FF}, \texttt{GG}\}$: all these strings were present on every cycle up until $t-1$; so it is expected that they will continue to be present;
- $Tstate(0, 1, t-1) = \{\texttt{UU}, \texttt{VV}, \texttt{WW}\}$: all these strings were present on some cycles up until $t-1$, so it is expected that they may or may not be present in general;
- $Pstate(1, 1, t-1) = \{\}$;
- $Tstate(1, 1, t-1) = \{\}$;
- $Pstate(1, 2, t-1) = \{\}$;
- $Tstate(1, 2, t-1) = \{\}$.

During cycle t, the agent detects that strings $\{\texttt{EE}, \texttt{FF}, \texttt{GG}\}$ have stopped occurring and that new strings $\{\texttt{II}, \texttt{JJ}, \texttt{KK}, \texttt{LL}\}$ have appeared instead. The new values are then as follows:

- $Pstate(0, 2, t) = \{\texttt{MM}, \texttt{NN}, \texttt{OO}, \texttt{PP}\} = Pstate(0, 2, t-1) - New(t)$;
- $Tstate(0, 2, t) = \{\}$
- $Pstate(0, 1, t) = \{\texttt{AA}, \texttt{BB}, \texttt{CC}, \texttt{DD}\} = Pstate(0, 1, t-1) - Stopped(0, 1, t)$;
- $Tstate(0, 1, t) = \{\texttt{UU}, \texttt{VV}, \texttt{WW}, \texttt{EE}, \texttt{FF}, \texttt{GG}, \texttt{II}, \texttt{JJ}, \texttt{KK}, \texttt{LL}\} = Tstate(0, 1, t-1) \cup Stopped(0, 1, t)$;
- $Pstate(1, 1, t) = \{\texttt{EE}, \texttt{FF}, \texttt{GG}\} = Stopped(0, 1, t)$;
- $Tstate(1, 1, t) = \{\}$: insufficient information at this stage;
- $Pstate(1, 2, t) = \{\texttt{II}, \texttt{JJ}, \texttt{KK}, \texttt{LL}\} = New(t)$;
- $Tstate(1, 2, t) = \{\}$: insufficient information at this stage.

For all known modes (i, j) the P-sets decrease with time:

$$Pstate(i, j, t) = Pstate(i, j, t-1) - Stopped(i, j, t)$$

The T-sets increase:

$$Tstate(i, j, t) = Tstate(i, j, t-1) \cup Stopped(i, j, t)$$

A P set only "increases" when it is first created (i.e. it goes from being empty to non-empty) on generation of a new activity index.

6.7.2 Relations

To show the rules governing the above changes, we can define some relations and use them to express the changes in the partitions with time.

- $\texttt{In\_trace}(s, t)$: string s is present in the trace at time t.
- $\texttt{Mode}(i, j, t)$: the current mode of subtype i is j at time t.
- $\texttt{Xor\_string}(s, i, t)$: s belongs to a mutually exclusive mode for activity i at time t.

Now we can give more general definitions of each type of set at each level (variables are universally quantified unless otherwise stated).

Definition of an "xor-string":
If any string s of activity type i is an xor-string then it is in exactly one P set (exactly one mode) for this activity:

$$\begin{aligned}&\texttt{Xor\_string}(s, i, t) \Rightarrow \\ &\quad (s \in \mathit{Pstate}(i, 1, t) \wedge s \notin \mathit{Pstate}(i, 2, t)) \vee (s \in \mathit{Pstate}(i, 2, t) \wedge s \notin \mathit{Pstate}(i, 1, t))\end{aligned}$$

Any string that violates this xor requirement is removed from the xor list. The reverse does not apply, i.e. there may be strings that mutually exclude each other but are not listed as xor-strings because they were not involved in an obvious transition, (e.g. only one string stopped before a new one started, which would be classed as a fluctuation, not a transition).

Only the P sets are used to make the decision on whether something is an xor-string. A scaled up version (taking into account the T sets) would be:

$$\begin{aligned}&\texttt{Xor\_string}(s, i, t) \Rightarrow \\ &\quad ((s \in \mathit{Pstate}(i, 1, t) \vee s \in \mathit{Tstate}(i, 1, t)) \wedge (s \notin \mathit{Pstate}(i, 2, t) \wedge s \notin \mathit{Tstate}(i, 2, t))) \vee \\ &\quad ((s \in \mathit{Pstate}(i, 2, t) \vee s \in \mathit{Tstate}(i, 2, t)) \wedge (s \notin \mathit{Pstate}(i, 1, t) \wedge s \notin \mathit{Tstate}(i, 1, t)))\end{aligned}$$

Definition of candidate pre-transition strings:
They are allocated to the *Stopped* set. More precisely, if any string s in $P_{i,j}$ at cycle $t-1$ is *not* currently present in the trace (at cycle t) and the current mode of i is j then s is not in $P_{i,j}$ at cycle t but is in $T_{i,j}$ and in $\mathit{Stopped}(i, j, t)$.

$$s \in \mathit{Pstate}(i, j, t-1) \wedge \texttt{Mode}(i, j, t) \wedge \neg \texttt{In\_trace}(s, t) \Rightarrow s \in \mathit{Tstate}(i, j, t) \wedge s \in \mathit{Stopped}(i, j, t) \wedge s \notin \mathit{Pstate}(i, j, t)$$

Definition of candidate post-transition strings (i.e. every new string): If any string s which was in N (i.e. in $P_{0,2}$) at cycle $t-1$ is present in the trace at t then s is in $\mathit{New}(t)$ and not in N.

$$s \in \mathit{Pstate}(0, 2, t-1) \wedge \texttt{In\_trace}(s, t) \Rightarrow s \in \mathit{New}(t) \wedge s \notin \mathit{Pstate}(0, 2, t)$$

Definition of a new activity type: If the number of stopped and started strings exceeds $\texttt{Mode\_Threshold}$ then mode 1 of the P set with the new activity index contains the stopped strings, mode 2 of the P set is provisionally the set of new strings and the parent mode is the mode in which the stopped strings were previously present. For this we require two new functions:

- $\mathit{Act\_index} : D \longrightarrow A$.
  $\mathit{Act\_index}(\mathit{timepoint})$ returns the activity type index associated with a particular transition timepoint.
- $\mathit{Parent} : A \longrightarrow A$.
  $\mathit{Parent}(i)$ returns the activity index of the parent mode of activity i.

The incorporation of the new activity type into the evolving knowledge structure can be defined as follows:

$$\mathit{Count}(\mathit{Stopped}(i, j, t)) \geq \texttt{Mode\_Threshold} \wedge \mathit{Count}(\mathit{New}(t)) \geq \texttt{Mode\_Threshold} \Rightarrow$$

$Pstate(Act\_index(t), 1, t) = Stopped(i, j, t) \wedge$
$Pstate(Act\_index(t), 2, t) = New(t) \wedge Parent(Act\_index(t)) = i$

Note that from a purely logical point of view these implications would also apply in reverse. Only one direction is shown because it is closer to the way the programming was actually done.

6.7.3 Uncertainty and knowledge refinement

For each discovered activity-type i, pre-transition events that have just stopped are allocated to $P_{i,1}$, while the post-transition events are only *tentatively* allocated to $P_{i,2}$. In the post-transition mode, some new events may be active in the mode's first cycle that will not continuously occur along with all other events associated with the new mode. Such events are transferred to the new mode's T set $T_{i,2}$ when they are found not to persist (mode-refinement). This initial uncertainty does not apply to $P_{i,1}$ because the events belonging to it have already been experienced as continuous. If the transition hypothesis is later refuted, the sets $P_{i,1}$ and $P_{i,2}$ become empty.

T sets for levels ≥ 1 represent very approximate knowledge initially. In particular, for mode 1 of any activity type, $T_{i,1}$ is initially empty because the agent has no knowledge of which members of the parent's T set are no longer present in the new mode (2) until the mode is observed for some length of time. Very roughly, the longer any parent T events do not occur in mode 2, the more likely it is that they will belong to $T_{i,1}$.

Due to the asymmetry of time, the rates at which the sets decrease or increase will not be the same for both modes, at least initially. If a new type has only just been discovered, it is much more likely that changes will be made in mode 2 (the "unknown" mode) than in mode 1.

6.7.4 Implementation summary

The POPRULEBASE psuedocode for generating an activity type is shown in table 2.

```
define RULESET ImmuneTraining_XOR
  RULE new_xor_hypothesis1:
  if time in training-phase is at least mode_duration_threshold and
    at least one event stopped occurring this cycle and
    it 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 new_xor_hypothesis2:
  if a new xor hypothesis has just been made and
    at least one new event started occurring this cycle then
  then
    hypothesise that the new events belong to mode 2

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

Table 2: Module specification for class discovery training phase

Note 1: Among the many options in POPRULEBASE, two execution modes are provided: "run first runnable rule only" or "run all runnable rules". In the former, only the first rule with matching conditions is run. In the latter, all matching rules are run. In Table 2 and all other modules presented here, the execution mode is set to "all runnable rules". This means that when there are multiple matches between a rule's conditions and the database contents, a "rule instance" is created for each one of them, and the rule's actions are executed for each instance. For example, the first rule in Table

2 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.

Note 2: For efficiency reasons, this pseudocode does not implement conditions which are identical to those in the definition of a new activity type given earlier; it implements a special case. Because survival in the current environment only required the generation of two activity types (immune-system activity and anomaly-related activity), only the second one has a parent which is not the root of the tree. Since, in this architecture, a participating agent never *re-enters* training mode after the main operational phase has begun (see temporal constraints diagram), the operational mode's P set can be regarded as the global P set. Thus, when the anomaly-related activity is being discovered, we do not need to recognise that operational mode is not a root. The only requirement is that the stopped events belong to the P set of any current mode. A quick way of being almost certain of this is to look at how long the stopped event had been active continuously (without interruption); if this is greater than the minimum observed duration of a mode (mode_duration_threshold) then it is likely that the event was an essential part of a mode, and not just a fluctuation. In general, however, this shortcut will not work.

6.8 Context-sensitive Anomaly Detection

Once the model has been acquired, an agent in operational phase can identify which mode (of each activity type) the observed meta-level is in. First it checks to see if there is an XOR violation in the trace's event records. Since we wish to identify a disabled component and repair it, we only need to detect one kind of violation: when two events from a mutually exclusive pair of modes are missing from the trace (e.g. one event is missing which is associated with training mode and another is missing which is associated with operational mode).

An XOR violation means that a critical component associated with one of the events may have been disabled, but we do not know which one yet, since one event should be missing in any case. To find out which event is anomalously missing, the agent identifies the current mode of the meta-level by looking at any remaining events which do not violate the XOR requirement and matching them to one of the mode partitions in the model. If the matching partition contains the missing event record, then this event should normally have been present during this mode and we can use this information to diagnose and repair the problem (as the event record points to the rule that should have been active).

From the trace, the agent can build a list of current modes $P_{i,j}$. If it is observing an agent in training phase there will only be one current mode, otherwise there will be more than one.

Definition of a mode omission anomaly: If there is any activity i with current mode j for which the set $P_{i,j}$ contains a string s and this s is not present in the trace then there is an omission anomaly.

$$\mathtt{Mode}(i,j,t) \wedge \exists s : s \in P_{i,j} \wedge \neg \mathtt{In\_trace}(s,t) \Rightarrow \mathit{Mode\_omission}(s,t)$$

Note 1: since it applies to operational mode, this definition refers to the object $P_{i,j}$ instead of the function $Pstate(i,j,t)$, since it is assumed that it no longer evolves. Instead of talking about changes in the model, we are now talking about about how the *world* is evaluated by the stabilised model. $P_{i,j}$ is now a specification for what should happen and not what actually happens.

Note 2: Since a deletion may indirectly cause disallowed events to occur in a mode, one may ask why we also do not include different types of XOR violation involving "disallowed" events (e.g. an event associated only with training mode suddenly appears in operational mode as a result of disabling a component). This was not found to be necessary in practice, as the ability to detect such a disallowed event does not produce any diagnostic advantage in the current environment (which only causes deletions). It could be added if needed, however.

6.8.1 Implementation summary

The pseudocode is shown in Table 3 and implements the above definition. **Note**: Since mode identi-

```
define RULESET ImmuneOperational_XOR;
  RULE xor_violation_type1:
  if there exist two events belonging to different modes of an activity and
    these events are both missing from the trace
  then
    delete both events from the activity description;
    store both events temporarily as "missing_xor";

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

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

Table 3: Module specification for context sensitive anomaly detection

fication is computationally intensive, the agent only identifies the current mode if there is actually an xor-violation of the type required.

7 Autonomous Recovery

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

The details of the implementation are as follows. Each agent has a "repair" agent associated with it. Repair is initiated by the main agent on detection of an anomaly in one of the following:

1. in the agent's own object-level;
2. in its neighbour's meta-level;
3. in its neighbour's repair agent

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

The number of false positives depends on how fine-grained the activity class discovery is and how long each mode of an activity is observed. For example, there may be occasional variations in execution patterns (hidden sub-modes) in the "no-anomaly-detected" mode of the observed agent's operational phase that were not discovered because the activity was not observed for long enough during the training phase. If such a fluctuation happens in operational phase, and it involves the sudden absence of an event previously thought to be essential for that mode, the absence will wrongly be treated as an anomaly. False positives cannot be completely eliminated, but if they only happen occasionally, we can assume this is acceptable in most situations.

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

If a backup rule is deleted this will not be detected until an agent attempts to repair itself or its neighbour using the backup rule. This limitation is unavoidable due to the nature of execution monitoring: a fault can only be detected in something that is executing (or should be executing). One way to overcome this problem is to inspect the static contents of the rulesystem at the end of every cycle (which was not done in this implementation). Other possibilities include monitoring of database access patterns in addition to execution patterns.

8 Results

When running the distributed architecture, we found the following:

1. The values of the critical variables (treasure and energy) remain stable (vehicle treasure value slowly increases as in the friendly environment); the vehicle always avoids the ditch.
2. There is repeated recovery from attacks against critical meta-level components, including those concerned with anomaly-detection and self-repair.

The system also recovers from an attack against its internal sensors without external intervention, although this leads initially to a large number of false-positives. The false-positives are unavoidable in this implementation because the internal sensors depend on a single rule to load the execution trace into the agent's database. When this rule is deleted, *all* required execution events are omitted, and it appears that nothing has happened in the last execution cycle (either in the neighbouring agent or in the anomaly-detecting agent's own object-level).

When the agent attempts to repair its neighbour's internal sensing rule, it looks up a copy of it in its own rulesystem (as described above). When the repairing agent detects that its own internal sensing rule is missing, it sets its status to "failure_mode" and waits for the neighbouring agent to repair its rule. The correct behaviour cannot be guaranteed if the assumption of a minimum interval between attacks (section 4.4) does not hold.

9 Discussion

We now ask if the prototype has relevance to real-world systems and whether it can "scale up". In particular, we consider the limitations of the following components and the assumptions they require:

- model acquisition algorithm
- internal sensors

9.1 Model acquisition algorithm

The model acquisition algorithm was primarily designed for the purposes of building a prototype. However, its basic principles may be applied to a more complex software system. One major assumption is that the design of the software can be constrained so that it is feasible to acquire a model of it (e.g. by only allowing certain kinds of interaction between components). This may also make the software easier to visualise.

Our initial approach is to provide only minimal capabilities to deal with the likely situations for a very specific problem (acquiring a model of a meta-level). For example, a minimal system only requires boolean modes, since the two modes represent an object and its negation.

One may argue that a third mode is required if, for example, the observed agent occasionally does more intensive checking during the normal submode of operational mode. Instead of treating "increased alertness" as a third submode of operational mode, however, the observing agent can treat it as a new activity subtype "alertness-related activity" of "normal" mode. The new subtype then has its own two modes: mode 1 is the "usual alertness" normal mode, while mode 2 is the "increased alertness" normal mode. In the context of negation discovery, the new modes may be more accurately called "usual alertness" and "not usual alertness" since the agent only detects the *difference* between the two, not the fact that the alertness is actually increased. As another example, "repair-related activity" has two modes: "repair required" (usual) and "no repair required" (not usual, e.g. false alarm). This makes the software design considerably simpler. In contrast, a real-world system may be more efficient with three or more submodes.

9.1.1 Relevance of classes discovered

What is discovered depends on how the training phase is designed. During the training phase, the parts of the software that are considered critically important or valuable are activated in various ways so that their different modes can be observed. For example, we can guarantee the protection of the capability to handle false alarms, as well as the capability to handle real anomalies if we "show" these behaviours in the training phase.

The decision on whether a transition should be regarded as a new activity depends on various numerical parameters (such as the number of events involved in the change). The most useful parameter setting could itself be learned (for example, an initially low threshold for the required number of events in a transition could be gradually raised until the number of refuted hypotheses decreases to a stable level). It does not matter if additional (unintended) modes are discovered. This may actually be useful if some interactions turn out to be critically important which were not known at design time. This does not happen in the current software because of its modular and hierarchical design (component hierarchy of rulesets and rules).

The precise structure of the acquired model may depend on the temporal ordering of states experienced in the training phase e.g. if increased alertness is experienced before an anomaly then the training phase will have to show the response to an anomaly in *both* these states to cover all possibilities. Consequently there will be two versions of "anomaly-related" activity - one that happened during increased alertness and one that happened during normal alertness). We assume, however, that the training phase can be designed so that the most "important" behaviours (relating to components that are most critical) will be shown first, and less important ones can be shown later.

To avoid dependence on temporal ordering, the model could be built by first accumulating the traces over many cycles and then analysing them. (Such a data mining approach cannot be implemented easily in SIM_AGENT however).

9.1.2 Multiple tree structures

We have assumed that a single observing agent builds only one tree structure of the type in figure 6 which requires that software observed by a particular agent has only one path into any mode. E.g. it is impossible to get into the "increased alertness normal mode" unless the agent is first in the normal mode of operational mode. This simplifies the programming because the context-sensitive anomaly-detection can be activated recursively e.g. the false-alarm modes only "exist" if the anomaly-detected mode is first active.

However, if we wish to activate the same "increased alertness" code from different branches of the tree, this results in exactly the same alertness events being active *regardless* of whether the observed agent has detected an anomaly or not. Both modes of "anomaly-related activity" would contain copies of the "alertness" activity type (which means that modes of different activities can contain the same events, and our partition diagram would no longer be valid).

Similarly, if an agent's external actions are also being modelled then the following activities could be generated:

1. treasure-related activity: seeking treasure (1) or not seeking treasure (2)
2. ditch-related activity: not avoiding a ditch (1) or avoiding a ditch (2)

Not seeking treasure may be further subdivided into "collecting treasure" and "not collecting treasure" (e.g. because it is seeking energy). To represent the fact that the ditch-related activity is independent of this, a single agent can build multiple trees. The agent might either first hypothesise that the new mode is a submode of the current one (until refuted) or it may start with the hypothesis that it is independent.

9.1.3 Related work on model acquisition

The model acquisition method introduced here has a similar purpose to existing data mining methods for intrusion detection, e.g. Lee and Stolfo [15].

Another related area is that of concept acquisition for mobile robots. Of particular interest is the algorithm used by Cohen [2] for clustering the robot's "experiences" in time, where each experience is a time series of low-level physical events. One of their aims is to get the clusters to correspond to human-recognisable events (such as bumping into a wall). In principle, we are addressing the same kind of problem on a simpler and more abstract level, in that the classes discovered should correspond to the normal operation of human-defined software components (such as checking for an anomaly).

Our context-sensitive anomaly-detection method is related to other work on context-sensitive reasoning in autonomous systems, in particular, the ORCA autonomous underwater vehicle project (Turner [30], [31]). An important feature of this architecture is the idea of a "current context" and the kind of actions that are appropriate or disallowed within it. This is similar to our concept of a "current mode" within which certain events are expected to happen and others are "prohibited".

9.1.4 Life-Long Learning

The distributed architecture may allow both training and operational modes to exist simultaneously if a single agent can be dedicated to one mode or the other. Each agent would have its own version of the evolving model which is either operational or under construction depending on which mode the agent is in. (Operational versions of the model could be "upgraded" regularly by agents in training mode).

9.2 Internal Sensors

The internal sensors used in the prototype are very similar to the sensors defined by [29] as follows:

> Internal sensor - a piece of software (potentially aided by a hardware component) that monitors a specific variable, activity or condition of a program and that is an integral part of the program being monitored.

This agrees fairly precisely with the kind of internal sensors used in our prototype. Sensors were added to the POPRULEBASE interpreter (an implementation level of the agent being monitored), which allows a monitoring agent to access rule execution events of the agent being monitored. Such sensors have been shown to work successfully in real-world intrusion detection by Spafford and Zamboni.

In our architecture, however, there are two levels of internal sensors: the low-level sensors embedded in an implementation level of the agent being monitored *produce* the trace because they record what they have seen. The high-level sensors of the monitoring agent's control system (which may be the agent being monitored) reads (and uses) the trace for anomaly-detection. In this way, the high-level sensors (implemented as as rules) are are controlled directly by the autonomous system in the way that a cognitive system directs its sensors.

9.2.1 Attention focusing

In the prototype, an agent's high-level sensors assume that much of the low-level processing has already been done, keeping the amount of data manageably small. This is not the case in real-world intrusion detection where sensors have to process very large amounts of data. Most of this data may be irrelevant to the diagnosis of a particular problem. It is therefore important to adjust dynamically the type of data being monitored. A related problem in agent societies is the *monitoring selectivity* problem [10].

A scaled up version of our prototype will also have this problem. For example, the data available to an agent might be increased by including data access traces in addition to rule-firing traces (for example, the number of reads or modifications of a selected data object within a designated time period). However, due to the strong coupling between an agent's high-level reasoning and its sensors, an agent can "focus attention" on the most relevant data according to the results of its reasoning. For example, it may be possible to select certain data objects for particular attention or try to obtain a more detailed trace of a suspect software component, while reducing the data obtained from trusted components. This may involve retasking of low-level sensors. For example, a component being monitored (which may be part of another agent) could have more of its trace-production "switched on" by the monitoring agent, assuming that it has authority to do this. Another possibility is to use mobile agents as intelligent sensors [9].

10 Conclusions and Future Work

Our exploratory implementation has been useful in the following ways: First, we have seen that the principle of distributed reflection is feasible and can be demonstrated to work in a restricted environment. Secondly, we have seen that the distributed architecture overcomes some of the limitations of a hierarchical multi-layer architecture. Thirdly, the methodology of first specifying the architecture on the conceptual level while using very simple versions of existing algorithms has led to some surprising conclusions about the kinds of algorithms and methods that are actually required. For example, we did not initially expect temporal constraints to play the important role that they did. Similarly, the exact requirements for the model acquisition algorithm were only known when a large part of the architecture was already implemented. These requirements in turn have led to extensions of the SIM_AGENT toolkit to support a wider range of self-monitoring and self-manipulation.

Future work will largely be concerned with scaling up the architecture. Some questions to be addressed are as follows:

- Allow more varied types of attacks, e.g. insertion of hostile code.
- Detailed analysis of the limitations of two agents and the feasibility of scaling up to three or more agents if necessary.
- Allow agents to be allocated varying amounts of processing time so that they can run at different speeds, e.g. to implement varying levels of vigilance or focus of attention. Observation of execution speed changes could then be included in the training phase so that it will be incorporated in the signature.

Another important consideration is the introduction of software diversity. A system with diversity is likely to be much more effective in practice than one that merely uses identical copies of code. For example, in the Treasure scenario, if one agent acts as a backup to the agent in control of the vehicle, it makes sense for it to use different software, since the problem could be due to a fundamental design error, and not due to a random fault. Similarly, the repair processes may hold backup components (rules) which implement the requirements of the faulty rule in an alternative way.

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] Paul Cohen. A method for clustering the experiences of a mobile robot that accords with human judgments. In *Proceedings of AAAI'2000*, July 2000.

[3] D. Dasgupta and N. Attoh-Okine. Immunity-based systems: A survey. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Orlando, October 1997.

[4] 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.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal contraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[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] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[8] S. Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

[9] W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Mobile agents in intrusion detection and response. In *12th Annual Canadian Information Technology Security Symposium*, Ottowa, Canada, 2000.

[10] G.A. Kaminka. *Execution Monitoring in Multi-Agent Environments.* PhD thesis, Computer Science Department, University of Southern California, 2000.

[11] C. M. Kennedy. Distributed reflective architectures for adjustable autonomy. In *International Joint Conference on Artificial Intelligence (IJCAI99), Workshop on Adjustable Autonomy*, Stockholm, Sweden, July 1999.

[12] 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.

[13] 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.

[14] S. Kornman. Infinite regress with self-monitoring. In *Reflection '96*, San Francisco, CA, April 1996.

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

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

[17] Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, USA, 1998.

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

[19] Rudy Setiono and Wee Kheng Leow. FERNN: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1-2):15–25, 2000.

[20] A. Sloman. What enables a machine to understand? In *Proceedings of the 9th International Joint Conference on AI*, pages 995–1001, Los Angeles, 1985.

[21] 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.

[22] A. Sloman. Exploring design space and niche space. In *Proceedings of the 5th Scandinavian Conference on AI (SCAI-95)*, Trondheim, 1995. IOS Press, Amsterdam.

[23] A. Sloman. Designing human-like minds. In *Proceedings of the 1997 European Conference on Artificial Life (ECAL-97)*, 1997.

[24] A. Sloman. Damasio, descartes, alarms and meta-management. In *Symposium on Cognitive Agents: Modeling Human Cognition, at IEEE International Conference on Systems, Man, and Cybernetics*, pages 2652–7, San Diego, CA, October 1998.

[25] A. Sloman. "semantics" of evolution: Trajectories and trade-offs in design space and niche space. In Helder Coelho, editor, *Progress in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, pages 27–38. Springer-Verlag, 1998.

[26] 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.

[27] B. C. Smith. Reflection and semantics in a procedural language. Technical Report MIT-LCS-TR-2721, MIT, 1982.

[28] J. M. Sobel and D. P. Friedman. An introduction to reflection-oriented programming. In *Reflection '96*, San Fransisco, CA, April 1996.

[29] E.H. Spafford and D. Zamboni. Design and implementation issues for embedded sensors in intrusion detection. In *Third International Workshop on Recent Advances in Intrusion Detection (RAID2000)*, 2000.

[30] Roy M. Turner. Context-sensitive reasoning for autonomous agents and cooperative distributed problem solving. In *Proceedings of the 1993 IJCAI Workshop on Using Knowledge in Its Context*, Chambery, France, 1993.

[31] Roy M. Turner. Context-sensitive, adaptive reasoning for intelligent AUV control: Orca project update. In *Proceedings of the 9th International Symposium on Unmanned Untethered Submersible Technology (AUV'95)*, Durham, New Hampshire, 1995.

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

[33] 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.