Distributed Reflective Architectures for Adjustable Autonomy

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

Abstract

A decision made by an autonomous system to adjust its autonomy status (e.g. override manual control) must be based on reliable information. In particular, the system's anomaly-detection mechanisms must be intact. To ensure this, a high degeee of self-monitoring (reflective coverage) is necessary. We propose a distributed reflective system, where the participating agents monitor each other's performance and software execution patterns. We focus on two things: monitoring of the anomaly-detection components of an agent (which we call meta-observation) and evaluating the "quality" of the agent's actions (does it make the world better or worse?). Using a simple scenario, we argue that these features can enhance the reliability of autonomy adjustment.

1 Introduction

In some anomalous situations, it may be appropriate for an autonomous system to suspend its operations while in others it may be necessary to override manual control. This paper addresses the problem of how the system should obtain accurate and relevant information about the world and about its own operation in order to support the correctness of such adjustments. We assume that that the system already has some degree of autonomy in that it can detect an anomaly and react to it independently. Very roughly, adjustments to autonomy fall into three categories:

- wait for user intervention (reduce autonomy)
- continue to initiate own activity, but allow manual control if available (stay the same)
- override manual control (increase autonomy)

The reliability of these decisions depends crucially on the accurate detection of anomalies. For example, if any of the system's sensors or anomaly-detection software is faulty, it may not detect a serious problem in its hardware (e.g. engine temperature).

We assume that anomalies can occur in the following areas:

- external world, e.g physical anomalies
- system hardware
- system software

An anomaly is defined here as a discrepancy between the model-predicted next state of the component under consideration and the actual next state as shown by sensors. The simplest methods include things like self-testing, parity-checking etc. where the "models" do not involve explicit reasoning. An example of more sophisticated modelling is [Pell *et al.*, 1998], although this is concerned with hardware components only.

Provision of such models and sensors may be called *coverage*. We focus on the problem of *reflective coverage*, where the system must detect anomalies in its own software.

2 Improving Reflective Coverage

A system which reasons about its own software is generally called a reflective architecture (see e.g. [Maes and Nardi, 1988]). However, such systems are usually not designed for autonomy, but instead allow a programmer to inspect and modify any part of the system on request. An exception is the self-monitoring system of [Kornman, 1996], although its purpose is to detect specific failure patterns (e.g. loops) and not anomalies as defined here.

As an example of type of reflection we are aiming for, we look briefly at the emerging area of artificial immune systems (AIS). The requirement is to detect computer viruses or other forms of intrusion which do not necessarily follow a known pattern (in contrast with typical virus scanning software). In the language of immunology, the system must distinguish between "self" and "nonself". For details, see e.g. [Forrest *et al.*, 1994]. A survey of immune-system inspired models can be found in [Dasgupta and Attoh-Okine, 1997].

In the context of adjustable autonomy systems, this requirement can be translated roughly as follows: all monitoring and control components should themselves be subject to monitoring by the system itself. In other words, there are no meta-levels (things which do the

monitoring or control) that are not simultaneously object levels (things which are monitored).

This is clearly an ideal situation but we take the view that an approximation is possible. The concept has its historical roots in"Second Order Cybernetics" (see e.g. von Foerster [von Foerster, 1984]) and the related idea of *organizational closure* of a network ([Maturana and Varela, 1980], [Varela, 1979]). (We subsequently use the term "meta-level closure").

2.1 Why Meta-Level Closure?

To show in detail why it is desirable to solve this problem, and why existing methods do not solve it, we use current AIS models as an example. Typically a database of "normal" patterns is used to define "self", which is collected in advance by running the system in isolation (in a "protected" environment). This database contains characteristic patterns produced by the execution of all legitimate programs and may be called the "signature" of the system being protected. There are many possible forms of recording such activity, e.g. file access patterns or system call sequences. Later, in the operational phase, the immune system continually compares actual patterns produced by currently active programs with the system's signature. If there is any significant deviation, a "nonself" has been detected. For details, including the definition of "significant", see [Dasgupta and Forrest, 1997].

To distinguish between self and nonself, an algorithm must ensure that the comparison between sets of patterns is carried out in the intended way. We call this algorithm the meta-level R. To our knowledge, existing AIS always have such a meta-algorithm. Even if the architecture is based on distributed pattern detectors e.g. [D'haeseleer *et al.*, 1996]), a meta-algorithm must "manage" the detectors and ensure that recognition of nonself occurs as intended. But if the underlying algorithm R of the anomaly-detection process behaves anomalously as a result of an intrusion, there is nothing that can sense this state in current AIS systems. One can say that there is a serious "gap" in their reflective coverage.

Simply including the pattern produced by the meta-algorithm R within the signature does not solve the problem, because this assumes that R will always be intact. If R becomes compromised then the whole immune system becomes unreliable. Indeed, R could be replaced by a "disinformation" algorithm which raises false alarms and covers up real anomalies. It follows therefore, that meta-level closure is desirable.

2.2 Distributed Reflection

If we are aiming for meta-level closure, the most immediate problem is to avoid an infinite regress of monitoring levels. We therefore propose a distributed, multi-agent architecture where the agents mutually observe each other's operation (mutual reflection). We assume that each agent has a "homeostatic" task, i.e. maintaining the quality of the world within critical values. The architecture of a single agent is summarized as follows:

- Reflection, R, detects anomalies in own operation and environment.
- Model, M, makes predictions about own next state and next state of environment.
- All other functions: in our present implementation, this is purely reactive and does not include any planning or scheduling.

We assume that an agent is a single thread. The idea of mutual observation is that an observing agent A_i should compensate for another agent A_j's inability to detect certain kinds of anomaly in its own operation. In particular, A_j will not be able to detect the failure of its component R.

Our approach has some similarities with existing models of mutual observation such as agent tracking e.g. [Tambe and Rosenbloom, 1996], and in particular social diagnosis [Kaminka and Tambe, 1998], where agents observe other agents' actions and infer their beliefs to compensate for deficiencies in their own sensors. However, such models are normally based on the "society" or "team" metaphor; i.e. the agents are separate entities which observe each other as if they were individuals in a society. Our architecture is intended as a distributed control system where the type of interactions between agents is much less restricted. For example, they may inspect (and perhaps even repair) each other's software.

In existing distributed control systems, decentralisation is normally not done for the purpose of improving reflective coverage. Advantages of decentralisation mentioned in the literature usually relate to issues such as agent specialisation and teamwork (e.g. [Laengle and Rembold, 1996]) or resource management and load balancing (e.g. [Meyer *et al.*, 1995]) although fault-tolerance and redundancy is sometimes mentioned in a more general sense, e.g. [Lueth and Laengle, 1994].

3 An Example Scenario

To describe our kind of architecture in detail, we present a simple scenario which is a modification of the "minder" scenario [Wright and Sloman, 1995]. This was originally designed to simulate motivation and emotional states in a nursemaid which is taking care of a number of babies. The reason for selecting this scenario is that we are also interested in agents which represent the "values" of users; i.e. the agents' autonomy (and its adjustment) should be centred around those things that the user is most *concerned* about (see e.g. [Norman, 1996] for related work on motivated agents). We will return to this later when considering design constraints for reflective coverage.

In our scenario, the nursemaid "looks after" a single baby, but the baby is gifted and can detect changes in the nursemaid's activity. Both agents are components of a larger "body" which is moved around the virtual world as a single entity. We call this the "vehicle". The agent software is embedded into this (simulated) hardware. For our purposes, the relevant hardware components are the

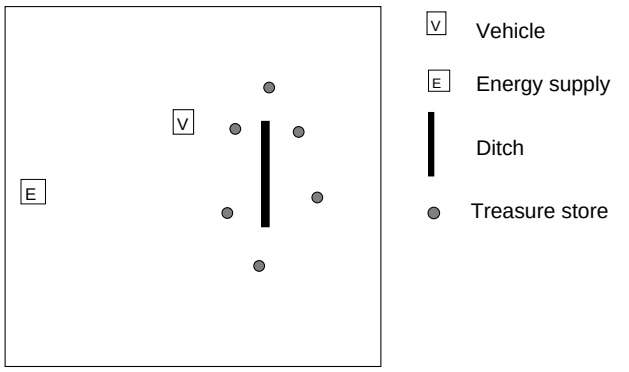

Figure 1: External world for modified minder scenario

sensors and effectors. The internal components of the vehicle (hardware and software) are called the "network". This includes the software of both agents, the processors on which they run and internal interconnections between them.

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

3.1 External World

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

- A static energy supply,
- a ditch, which the vehicle might fall into,
- various treasure stores near the ditch which the baby finds interesting.

The baby's task is to find as much treasure as possible in order to maintain its level of interest, which falls rapidly in the absence of anything new. It seeks out the treasure stores and collects treasure from them (although in practice this only means that it remains stationary until its interest level falls off). The nursemaid's task is to maintain the energy level and safety of the whole vehicle. Normally the baby has control of the vehicle but it is taken over by the nursemaid if the energy level falls below a certain level or if the vehicle is in danger. Both agents can detect anomalies in each other's activity as well as in the external world. A schematic diagram of the scenario is shown in Figure 1. Depending on the amount of redundancy built into the design, an agent can take over another's task in a failure situation. However, the ability to do this is limited since the nursemaid is intended as a specialist in energy and safety while the baby is a specialist in treasure.

3.2 Internal World

In order to approximate the meta-level closure requirement, our architecture provides the following features which contrast with existing multi-agent systems: *First*, the agents' world (which is being observed) includes execution patterns of software (in addition to hardware and external world). We call this the "internal" world. The method of software monitoring may involve the checking of expected signatures with actual patterns (e.g. rule-firing patterns [Kornman, 1996] or statistical analysis of time-series as used in immunity-based systems [Dasgupta and Forrest, 1997]). Any software which is used to detect actual execution patterns of other software may be called an "internal sensor".

Secondly, there should be no critical reliance on a single method of global coordination (e.g. a particular communication protocol). Any coordination method C should be monitored by a meta-level R that does not rely on it. Otherwise if C fails then R may also fail and the detection of the failure of C may not be possible. Therefore we have a serious fault that goes undetected.

Thirdly, not only must the decision-making software be monitored but also the software which detects anomalies, i.e. we require monitoring of the monitoring process (which we call "meta-observation"). Since this is a difficult problem it is necessary to show how it works in more detail.

Meta-Observation

Meta-observation requires monitoring of software using internal sensors. An architecture with those components is shown schematically in Figure 2. The external world is labelled "EW". The internal world ("IW") is divided into nursemaid (N) and baby (B) and includes all their hardware and software. Their different components S_E (external sensors), E_E (external effectors), S_I (internal sensors) and E_I (internal effectors, e.g. error-recovery) are assumed to be low-level software in which the high-level agent software C (for control) is embedded. The IW is embedded within the vehicle which is in turn embedded in the EW. The meanings of the term "sensor"

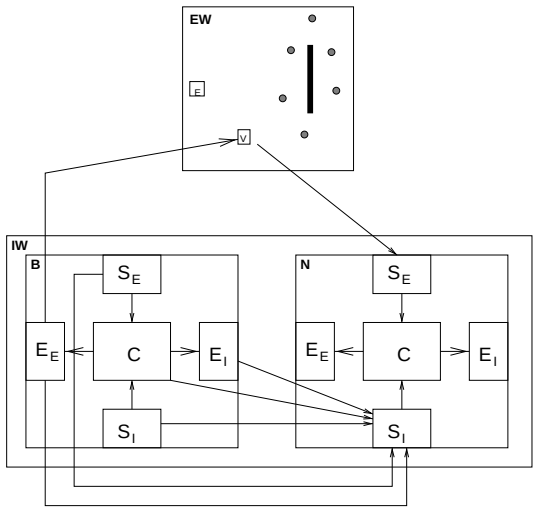

Figure 2: External and internal interfaces

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

We assume for simplicity that an agent repeatedly "calls" its internal sensor in the same way that it calls any other procedure. (We will ignore timing considerations for the moment). We use the following convention: The *pattern of execution* of the internal sensor of an agent A_i will be labelled $S_I(i)$. In contrast, the *value* returned by the sensor when applied to an object X will be labelled $S_I(i, X)$. Then we can write A_i's meta-observation of A_j as $S_I(i, S_I(j))$.

Meta-observation of internal sensors appears at first to lead to an infinite regress:

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

However, this need not occur in an implementation. There should be two concurrent threads monitoring each other: $S_I(i, S_I(j))$ which is part of the execution cycle of the observing agent A_i and $S_I(j, S_I(i))$ which belongs to the execution cycle of the observed agent A_j (if we assume for simplicity that each agent is a single thread). A_i's thread does not wait to evaluate the content of $S_I(j, ...)$; instead A_i monitors the *pattern of operation* of $S_I(j, ...)$. The result is A_i's specific way of describing $S_I(j)$. For illustration we can imagine a thermometer T whose operation is monitored using a device D (e.g. by ensuring that there is mercury in it). But D can only work at a certain temperature, which in turn needs to be monitored using T.

In the same way, the high-level anomaly-detection software can be monitored, i.e. the observed control system $C(j)$ which repeatedly calls $S_I(j)$ and checks its value against a signature, can be monitored by $C(i)$ using $S_I(i)$. Instead of simply using a stored signature, a model may be used to predict the next state of the software as it behaves in a "normal" situation.

3.3 Fault Injection

An "enemy" is simulated by a random number generator which determines when the baby or nursemaid should be damaged or if there should be an environmental anomaly, e.g. the ground around the ditch becoming unexpectedly slippery. The enemy can only inflict *direct damage* on an agent's components (by deleting, corrupting or otherwise modifying its software), e.g. there is no deception or stealing of resources at present.

It is assumed that the enemy cannot damage two agents simultaneously. Furthermore, it can only damage one agent component within a certain time period which should allow detection and recovery. A "component" is defined as of the above categories (e.g. control system or internal sensor).

3.4 Implementation

For exploratory implementations and rapid prototyping, we use the SIM-AGENT development environment [Baxter *et al.*, 1998]. This environment provides generic object classes which may be specialised into agents or inanimate entities. Each agent class has its own database and rulebase (called a rulesystem). A scheduler "runs" each agent by first updating the agent's sensors with the current state of the world, then using a rule interpreter to run its rulesystem, and finally updating the spatial world according to any external actions required by the rules. Normally each agent is run once in a single time-slice.

Each agent uses its own method of observation. The nursemaid has a copy of the rulesystem code of the baby and regularly compares this with the actual state of the baby's code (in this way it can detect random damage or unauthorised modification); the baby has a model of the nursemaid's rule firing pattern, which it compares against the actual pattern.

When monitoring the firing pattern of the nursemaid's anomaly-detection rules, the baby will observe one of three classes of patterns. First, there are two patterns indicating "correct" activity: the nursemaid's recognition of a normal situation and its recognition of an anomaly (in which case the baby can indirectly infer a problem that it had not detected itself - as in the social diagnosis approach). We may label these patterns "no" and "yes" patterns respectively. Any pattern which does *not* fall into the "no" or "yes" categories is an anomaly in the nursemaid's anomaly-detection. Details of the implementation are in [Kennedy, 1999].

4 Focusing on What Matters

The above implementation only concentrates on monitoring of anomaly-detection (meta-observation). This alone does not provide an approximation of meta-level closure, although it is a necessary component of it. However, we can arbitrarily extend the coverage to include other software components, e.g. decision-making. In particular, those components which the observed agent's anomaly-detection *relies on* should be given priority, e.g. its mechanisms for predicting the next state and for evaluating quality (see later). Similarly, the "resolution" of internal sensors can be arbitrarily increased, so that they do not just determine whether something is a yes/no pattern but also detect fine differences between patterns within a "no" or "yes" class.

The aim is to make the reflective coverage "sufficient" for a particular application, i.e. anomalies should be detected in any system operation which is necessary to meet critical requirements.

Quality evaluation

All of the above is concerned with the detection of *pattern* anomalies. Sufficient reflective coverage should also include the evaluation of *quality* of the world, i.e. the

observing agent evaluates the quality of the observed agent's actions. This is important because a change in *pattern* does not necessarily indicate problems unless it is coupled with a negative change in external behaviour. In the minder scenario, a modification to the nursemaid's control software may result in an anomalous trace, but it may still continue to maintain the energy level as required. It would look problematic only if the nursemaid ceased to act at all or acted in a destructive manner. (Using the immune system metaphor, it is important to minimise "allergic" reactions, particularly during learning or authorised software upgrades).

It is therefore necessary to ask "what is important?" and this depends on the human or other agent the system is acting on behalf of: for example, in a network security system, data integrity must be preserved; in a life-support system, oxygen supply must be maintained etc. Maintaining minimal quality is a critical requirement.

4.1 Overriding another Agent's Control

We now show how meta-observation and quality evaluation can enhance the reliability of adjustable autonomy.

If a system is to override manual control, it must determine whether a critical requirement is being violated. E.g. an operating system detects an anomalous pattern of commands typed in by an intruder who has the superuser password and is trying to delete all files. Such a sequence of commands would be anomalous from the point of view of *pattern* (it is extremely irregular) and from the point of view of *quality* (it is a violation of requirements). Therefore the system should override the user's commands.

Adjustable autonomy in the Minder scenario

In the Minder scenario, the user is normally a passive observer (although this is modifiable). Normally there is an exchange of vehicle control between the baby (B) which does the treasure collecting and the nursemaid (N) which replenishes the energy level.

In anomalous situations, there may be an exceptional override of control. This happens when the quality of the world is perceived to have deteriorated beyond acceptable levels. The reason for the deterioration may be due to an environmental anomaly or a system failure. The following are of situations which N may encounter while monitoring B's control of the vehicle:

- B remains stationary for more than t cycles (where t is much longer than the longest "observation" period). If this is accompanied by an internal pattern anomaly in B's software then N concludes that B's software has failed. Otherwise it reports (to the user) a possible fault.
- B has led the vehicle too close to a ditch. This is anomalous because the B's software should prevent this, but it may also be due to an environmental or hardware anomaly. N takes over control of the vehicle (since it is a specialist in ditches) and reports a possible fault. If this is accompanied by an *internal* pattern anomaly in B's software then N concludes that B's software has failed.

In any of the situations where N concludes a definite failure, it takes over control of the vehicle and tries to repair the fault or damage. Taking over control in the event of certain failure means that it may also attempt to do some of B's work depending on the amount of redundancy we want to build into the system. Full redundancy would mean that N has the potential to do the job of B as well as B itself can and vice versa.

The following are examples of situations where B detects that the minimal requirements are violated during its monitoring of N:

- The energy level has dropped to the danger level and N has not yet taken control of the vehicle. B takes over the task of N in this case and concludes there is a failure.
- N moves the vehicle towards the ditch. If it moves into the danger zone or there is also an internal pattern anomaly, B takes over the task of N and reports a failure.

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

It may be argued that there is no need to monitor software execution patterns here, since external anomaly-detection and quality evaluation alone appear to be enough. However, if the quality evaluation software is corrupted or modified, there would be no way to detect this if we did not have pattern anomaly detection. To ensure that the pattern anomaly detection is intact, we require meta-observation as outlined above.

Clearly, there are many other complex issues here that we have not addressed, such as making a decision on the basis of a prediction or using uncertain information. However this goes beyond the scope of the paper.

5 Conceptual Exploration of Designs

The main purpose of the simulation is to investigate the feasibility and advantages of meta-level closure, and in particular, how does it affect the reliability of autonomy adjustment decisions. We are also exploring the following design tradeoffs:

- Number of agents: what are the advantages of more than two? It would appear that three agents would be more reliable. For example, if agents detect

failures in each other's anomaly-detection, an additional agent must determine which one has actually failed.

- *Design* redundancy vs. *version* redundancy: what is the difference between agents based on the same design and those based on independently developed designs (as for example in [Minsky, 1996]. It would seem that independent designs are more robust. However, this makes it more difficult for agents to repair each other's code. Similarly, the agents will generally not have the ability to do each other's tasks equally well (E.g. B is motivated by the need to maintain its "interest" level but N is not).
- Load distribution vs. specialist monitoring agents: should there be agents whose sole task is to monitor other agents? In the present implementation, both agents switch attention between their monitoring tasks and their "normal" tasks. Quality evaluation of the performance of a specialist monitoring agent is difficult (as it is mostly a passive observer).

References

[Baxter *et al.*, 1998] Jeremy Baxter, Richard Hepplewhite, Brian Logan, Aaron Sloman. SIM-AGENT: Two Years On. Technical Report CSRP-98-02, School of Computer Science, University of Birmingham, 1998.

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

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

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

[Forrest *et al.*, 1994] Stephanie 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.*

[Kaminka and Tambe, 1998] Gal A. Kaminka and Miland Tambe. What is Wrong With Us? Improving Robustness Through Social Diagnosis. In *Proceedings of the 15th National Conference on Artificial Intelligence* (AAAI-98)

[Kennedy, 1999] Catriona Kennedy. Collective Anomaly Detection using Immune Systems Principles. In *Proceedings of the ESSLLI99 Workshop on Collective Agent Based Systems*, Amsterdam, August 1999 (forthcoming).

[Kornman, 1996] Silvie Kornman. Infinite Regress with Self-Monitoring. In Proceedings *Reflection'96*, San Fransico, April 21-23, 1996.

[Laengle and Rembold, 1996] Thomas Laengle and Uwe Rembold. A Distributed Control Architecture for Intelligent Systems. In *Proceedings of the International Conference on Advanced Sensor and Control Systems*, 1996.

[Lueth and Laengle, 1994] Fault-Tolerance and Error Recovery in an Autonomous Robot with Distributed Control Components. In Proceedings of the DARS'94, 1994.

[Maes and Nardi, 1988] Pattie Maes and Danielle Nardi, editors. *Meta-Level Architectures and Reflection*, North-Holland, 1988.

[Meyer *et al.*, 1995] Kraig Meyer, Mike Erlinger, Joe Betser, Carl Sunshine. Decentralized Control and Intelligence in Network Management. In *Proceedings of the 4th International Symposium on Integrated Network Management*, Santa Barbara, CA, May 1995.

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

[Maturana and Varela, 1980] Humberto Maturana and Fransisco J. Varela. *Autopoiesis and Cognition: The Realization of the Living.* D. Reidel Publishing Company, 1980.

[Norman, 1996] Timothy J. Norman. Motivation-based direction of Planning Attention in Agents with Goal-autonomy. Phd Thesis. DAI Unit, Department of Electronic Engineering, Queen Mary and Westfield College, 1996.

[Pell *et al.*, 1998] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner and Brian C.Williams. An Autonomous Spacecraft Agent Prototype. In *Autonomous Robotics*, **5**, 1-27, Kluwer Academic Publishers, Boston, 1998.

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

[Varela, 1979] Fransisco J. Varela. *Principles of Biological Autonomy*, North-Holland, 1979.

[von Foerster, 1984] Heinz von Foerster. *Observing Systems* Intersystems, Seaside, CA, 1981.

[Wright and Sloman, 1995] Ian Wright and Aaron Sloman. MINDER1: An Implementation of a Proto-emotional Agent Architecture, Technical Report CSRP-97-1, School of Computer Science, University of Birmingham, 1997.