A Sentinel Approach to Fault Handling in Multi-Agent Systems

Staffan Hägg

Department of Computer Science
University of Karlskrona/Ronneby, Sweden
Staffan.Hagg@ide.hk-r.se
http://www.sikt.hk-r.se/~staffanh

Abstract

Fault handling in Multi-Agent Systems (MAS) is not much addressed in current research. Normally, it is considered difficult to address in detail and often well covered by traditional methods, relying on the underlying communication and operating system. In this paper it is shown that this is not necessarily true, at least not with the assumptions on applications we have made. These assumptions are a massive distribution of computing components, a heterogeneous underlying infrastructure (in terms of hardware, software and communication methods), an emerging configuration, possibly different parties in control of sub-systems, and real-time demands in parts of the system.

The key problem is that while a MAS is modular and therefore should be a good platform for building fault tolerant systems, it is also non-deterministic, making it difficult to guarantee a specific behaviour, especially in fault situations. Our proposal is to introduce *sentinels* to guard certain functionality and to protect from undesired states. The sentinels form a control structure to the MAS, and through the *semantic addressing* scheme they can monitor communication, build models of other agents, and intervene according to given guidelines.

As sentinels are agents themselves, they interact with other agents through agent communication. The sentinel approach allows system developers to first implement the functionality (by programming the agents) and then add on a control system (the sentinels). The control system can be modified on the fly with no or minimal disturbance to the rest of the system.

The present work is conducted in cooperation with Sydkraft, a major Swedish power distribution company. Examples are taken from that venture, and it is shown how problems can be solved by programming DA-SoC agents, developed here.

Keywords

Multi-Agent Systems, Fault Tolerance, Semantic Addressing, Sentinel

1 Introduction

Fault handling in Multi-Agent Systems

When discussing fault handling in multi-agent systems (MAS) there are often two opposite positions. The first says that a MAS is inherently fault tolerant by its modular nature; faults can be isolated and generally do not spread. The other says that a MAS is inherently insecure, as control is distributed. It is non-deterministic, and a specific behaviour is hard to guarantee, especially in fault situations.

The Societies of Computation (SoC) Research Project

The background to this paper is a research project, the Societies of Computation (SoC) [5][10], at the University of Karlskrona/Ronneby in Sweden. In a joint effort with Sydkraft, a major Swedish power distribution company, technologies for automating the power distribution process are studied. The goal is to make the distribution process flexible and robust to load changes, re-configurations, faults, etc. (Distribution Automation - DA), and to offer new services to customers, such as the choice between different levels of service and even between different suppliers (Demand Side Management - DSM). Furthermore, and perhaps the most challenging goal is to exploit a new infrastructure for Home Automation (HA), integrated with the DA/DSM system. The physical electric grid is used for communication, and small and simple pieces of hardware are plugged into wall-sockets and lamp-sockets, operating as interfaces to appliances, or are integrated within them. Basic techniques for this is in the process of being standardized. Our challenge is therefore to develop methods for using this infrastructure in terms of computation and communication, thus creating truly plug-in software. Computation in this environment is thus distributed *en masse*, and changes are unpredictable; there exists no generic way of describing the total system at any one point in time.

The first phase of the joint project, called DA-SoC [7], takes an agent-oriented approach to distributed computing, and it includes the definition of an agent architecture, an agent language, and a programmable model of interaction. The second phase, part of which is described here, is called Test Site Ronneby (TSR). Our goal is to create a full scale test environment in the city of Ronneby for DA/DSM and HA, as described above, and doing this, one essential aspect, not covered in the initial work, is to make the running system tolerant to a number of fault situations.

Outline of the Paper

We start with a short outline of how fault tolerance is traditionally addressed in industrial systems and in other MAS research. Then we try to identify what the core problems are when a MAS approach is used. We give an short description of the DA-SoC agent architecture, and then we introduce the concept of sentinels in this context. Two examples of sentinels in a MAS are given: in negotiation and in inconsistency detection. With the conclusions we also sketch how the concept of sentinels may be generalized for other purposes than robustness.

2 Fault Tolerant Systems

First we identify that the applications we have in mind have some common properties:

- a massive distribution of computing components,
- a heterogeneous underlying infrastructure (in terms of hardware, software and communication methods),
- an emerging configuration,
- possibly different parties in control of sub-systems, and
- real-time demands in parts of the system.

Fault tolerance is especially well studied for real-time systems. From that work we take some important definitions as a background to our study. Here, we mainly follow a standard textbook on real-time systems (Burns and Wellings [2]).

The sources of faults

According to Burns and Wellings there are four main sources of faults:

(1) Inadequate specification of software
(2) Software design error
(3) Processor failure
(4) Communication error

where the former two sources are unanticipated (in terms of their consequences). The latter two sources, on the other hand, can be considered in the design of the system.

The consequences of faults

If the occurrence of a fault results in the system not being able to conform to the specification of the system's behaviour, there is a *system failure*. If, on the other hand, the system can handle the fault situation, it is called *fault tolerant*. Even if all possible measures are taken to prevent faults, the former two sources above imply the difficulty in building fault-free systems. This emphasizes the need for fault tolerance.

Degrees of fault tolerance

We observe the important notion of levels or degrees of fault tolerance:

- *Full fault tolerance*, where the system continues to operate without significant loss of functionality or performance even in the presence of faults.
- *Graceful degradation*, where the system maintains operation with some loss of functionality or performance.
- *Fail-safe*, where vital functions are preserved while others may fail.

Methods for building fault tolerant systems

Perhaps, the most basic principle behind building fault tolerant systems is that of modularity. First, it lets the designers give structure to the system, helping in data model-

ling, implementation, and testing. Second, for the running system, it makes it possible to isolate a fault as it appears, sometimes referred to as *firewalling* against faulty components. Third, it is a basis for other principles, such as information hiding, checkpointing, and redundancy.

Redundancy refers to techniques where several modules are given the same functionality. These can be identical, e.g. mirroring storage systems, or modules backing up the running module in case of fault. There can also be one or more differing modules taking over functionality from a faulty one, though with loss of performance. A somewhat different approach is the concept of N-version programming, proposed by Chen and Avizienis [3]. Here, N individuals or groups independently develop a module from the same specification. Then at run-time, all N modules execute in parallel, and the results are compared and the final result is voted for. This addresses fault sources (1) and (2) above.

The third principle is the choice of implementation language. The language should traditionally allow the system's behaviour to be determined from the code. This demand for *determinism* disregards such features as dynamic binding and operator overloading.

The problem of inconsistency

One approach to handle inconsistency is using formal methods for describing the system. This can be made in the specification phase with specification languages, such as Z, or for the running system, modelling the whole or parts of the system as a Truth Maintenance System. The problem is that all experience say that if the system is of any considerable size or complexity, inconsistencies *will* appear. This is especially the case when, from physical or other reasons, parts of the system is distributed (as is assumed here). Therefore, if it is important to avoid inconsistency entering the system, it is even more important being able to handle inconsistency as it appears (i.e., a fault tolerant system[1]).

Fault Handling Approaches in Multi-Agent Systems Research

Fault handling is not much addressed in MAS or DAI research. Generally, agents are thought of as being both benevolent and fault-free. Some research has addressed non-benevolent agents or agents with conflicting goals, but agent theories and architectures usually avoid the problem with faulty agents. The matter of inconsistency is especially difficult. It is discussed by Boman and Ekenberg [1], and they call a "global inconsistency over a finite number of locally consistent structures ... a *paraconsistency*." However, our use of inconsistency is more restricted than theirs: we do not consider a formal system where inconsistencies automatically can be detected; an inconsistency is defined from the system design as a global state that should not occur, and if it occurs, it is caused by any one of the four sources of faults mentioned above.

1. Here we consider an inconsistency being a consequence of one of the four sources of faults mentioned earlier, and hence, the inconsistency is considered a fault. Another notion for this could be a defect, but for reason of simplicity, we do not distinguish between the two in this paper.

For agent based systems that have been used commercially, fault handling is not much considered, and mostly it takes traditional approaches. For example, in ARCHON [13] (a project with partly the same objectives as ours), the problem of global coordination and coherence is addressed. It is determined to be part of the acquaintance model (and hereby it shows some likeness to our approach), but the issue of fault handling is not given much attention as a problem in its own right. In the PRS system [9] which is used for a number of commercial applications, there is no explicit fault handling techniques. Fault handling is taken care of by the underlying communication and operating system. In case of faults generated in the specification, design, or implementation phase of the system, normal debugging and testing procedures are used.

The conclusion is that for MAS, fault handling is considered difficult to address in detail and often well covered by traditional methods. In the following it is shown that this is not necessarily true, at least not with the assumptions on applications we have made.

3 Identifying the Problem

We started with stating that there are two opposite positions to fault handling in MAS, one saying that a MAS is inherently fault tolerant and the other saying that it is difficult to make a MAS fault tolerant. This is easy to understand, given the background of the previous section. The agent paradigm perfectly conforms to one of the basic principles for building fault tolerant systems, namely modularity. At the same time it corresponds very poorly to another of these principles, namely that an implementation language (or, more generally, an implementation model) should be deterministic.

Figure 1 illustrates the relationship between different programming paradigms and the matter of control. Orthodox real-time systems are totally *deterministic*; the behaviour of the system can, at least in principal, be determined from the program code. Object-oriented systems have introduced some dynamic features, such as dynamic binding of modules and operator overloading. These systems were initially criticised but are today widely used for real-time applications (e.g., in the telecommunication area and for making computer operating systems). Multi-agent systems represent the other extreme. There is no inherent mechanism for synchronization or control; the system is inherently *non-deterministic*. If we shall guarantee a specific behaviour,

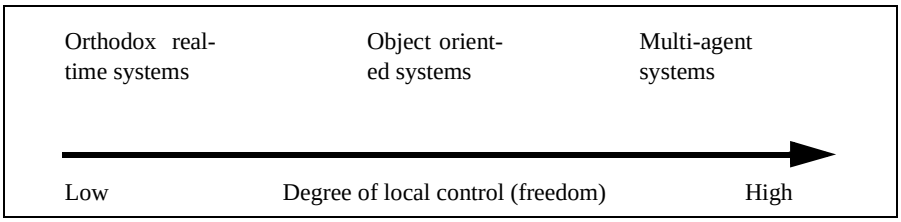

Figure 1. Comparison of local control (freedom) between some types of system.

especially in the presence of faults, then we must find a way to deal with the non-deter-

ministic property of the MAS. The method proposed in this paper does exactly that. Sentinels introduce a supervising and control model for the operating MAS.

4 The DA-SoC Agent Architecture

Before we introduce the concept of sentinels we will shortly outline the DA-SoC agent architecture and interaction model. It is not possible to show all details of the architecture, the language, etc. This is further described in [7].

A DA-SoC agent is procedural and goal-driven, and it holds a world model in declarative form. Its most elaborate feature is a programmable model of interaction, that lets the user tailor its interactive behaviour. A simplified picture of the agent head is shown in Figure 2. The world model consists of a set of beliefs. The reasoner holds agent plans and a goal queue, letting the user program the agent in an event driven fashion. The programmable Model of Interaction (MoI) is realized through a set of interaction plans that are matched and, eventually, executed when a message comes in. By programming the MoI, the user can give agents specific roles in the society.

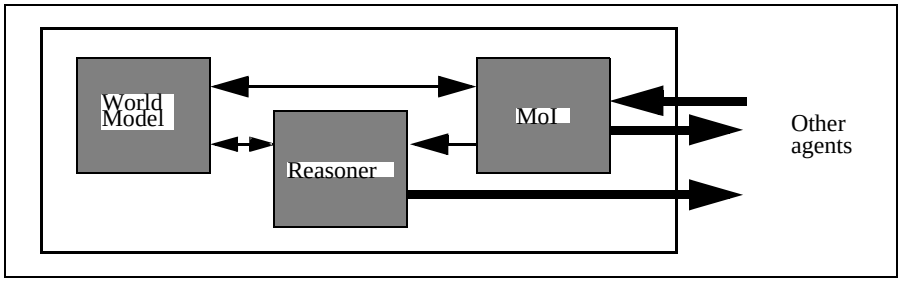

Figure 2. Structure of the DA-SoC agent head

The basic communication mechanism in DA-SoC is a broadcast message passing scheme[1]. When a message is sent by one agent, all other agents receive it, and every agent tries to match it with an interaction plan. The declaration part of an interaction plan has an agent name as the receiver's address and a pattern for what message contents it will accept. Both the agent name (the address) in the message and the message contents must therefore match the declaration part of an interaction plan. Upon a successful match, the plan is executed. During execution, the agent's world model may be altered. There can also be set an internal goal. In this case, the goal is matched with agent plans and, eventually, an agent plan is executed, during which the world model can be altered, goals can be set, and messages can be sent. A special feature is that one agent can hold an interaction plan with the name of another agent as the receiver's address. This enables one agent to play the role of another agent within the society.

1. Fisher discusses in [4] the impact of broadcast communication in MAS.

Semantic addressing

Let us call our plans methods, accessible by other agents. Then the broadcast mechanism, the MoI, and agent plans form consecutive steps in an access model which we call *semantic addressing* (Figure 3). (1) shows an agent executing an agent plan. During execution, a message "request ..." is broadcast (2). The receiving agents try to match the incoming message with interaction plan declarations (3), and in (4) we see that agents 2 and 4 execute successfully matched interaction plans, while agents 1 and 3 failed in matching the message. Finally, (5) shows that agent 2 set a goal during execution of the interaction plan, and the matched agent plan is now executed. Here, agent 4 either did not set a goal or failed in matching the goal with an agent plan.

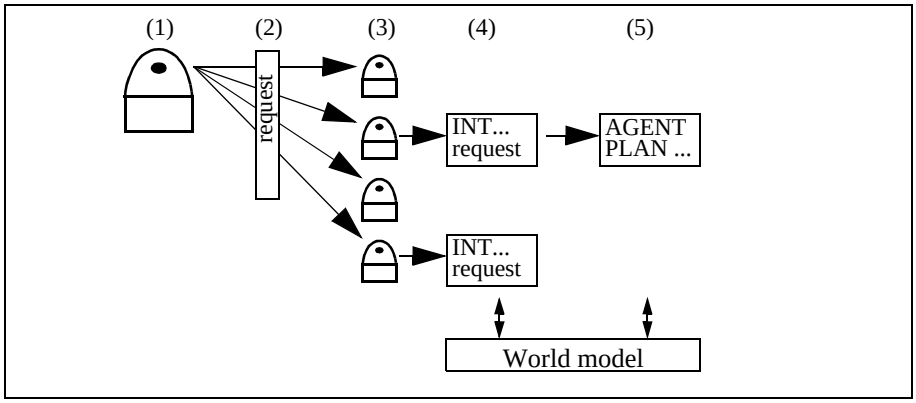

Figure 3. Semantic addressing in DA-SoC

Semantic addressing allows agents to access methods within other agents in the society without knowing where they should be found. But moreover, a method is accessed wherever it is semantically meaningful according to the application definitions, and where it is consistent with the agent's world model (which can change). The world model may be altered due to changes in the surrounding world or in agents' capabilities. Thus, the next time the same message is sent, there may be a goal set and a plan executed within another agent than the first time.

Semantic addressing is outlined in length in [8]. There it is shown how this access scheme allows for agents to play different roles in a society (e.g., backing up each other, or adapting communication patterns to changes in the environment). Here we lay down semantic addressing as a basis for the concept of sentinels.

5 Sentinels in a Multi-Agent System

In our approach we first conclude that if the total system is of considerable size and complexity, involving unreliable communication media, a fully fault tolerant system is not realistic. Therefore, the designer must decide on which functions are most vital for the system's integrity. The system must also have good support for both fault handling and fault reporting. Second, the technique we choose, actually decreases the freedom

for agents where it is necessary for preserving the system's integrity[1]. And third, this control must be easy to maintain and change (e.g., in case of version changes, during repair of faulty components, or as the system emerges over time).

Sentinels

A sentinel is an agent, and its mission is to guard specific functions or to guard against specific states in the society of agents. The sentinel does not partake in problem solving, but it can intervene if necessary, choosing alternative problem solving methods for agents, excluding faulty agents, altering parameters for agents, and reporting to human operators.

Being an agent, the sentinel interacts with other agents using semantic addressing. Thereby it can, by monitoring agent communication and by interaction (asking), build models of other agents. It can also use timers to detect crashed agents (or a faulty communication link).

Checkpoints

Given a set of agents that cooperate in realizing a system function, a sentinel is set to guard the operation of that function. The sentinel now maintains models of these agents, as mentioned above. Some items in such a model (expressed as beliefs) are directly copied from the world model of the agent in concern, becoming parts of the world model of the sentinel. We call these items *checkpoints* (Figure 4). These check-

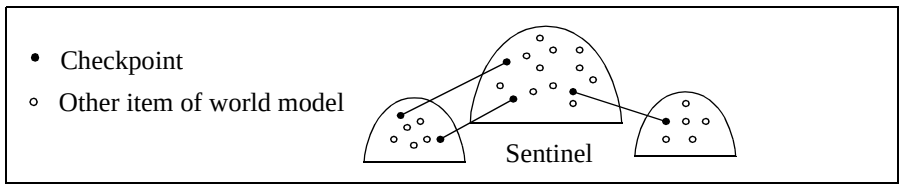

Figure 4. Checkpoints and world models

points let sentinels judge on the status of an agent, not only from its behaviour, but from its internal state. This is a means for early detection of a faulty agent and of inconsistency between agents (which is considered a system fault).

System development

The DA-SoC architecture supports incremental system development. In a typical development process, the developer iterates between design and implementation (and even back to specification). With DA-SoC, some agents, realizing a certain function, may be implemented and the function can be tested. Other agents can thereafter be entered to the running system, and the total system emerges over time.

1. This may be a controversial result. Many researchers prefer to view agents as totally independent. We find it, however, essential to impose this kind of restriction to agents freedom in order to cope with the non-determinism, mentioned earlier. This is further discussed in the conclusions of the paper.

For our purpose this means that the whole system can be developed and its operation can be tested before the control system (the sentinels) is put on top of the fully operating system, Figure 5. The decision on checkpoints can be made as the sentinels are designed, as well as communication between the sentinels and between sentinels and operator terminals. It also means that the control system can be modified later without disturbing the running system.

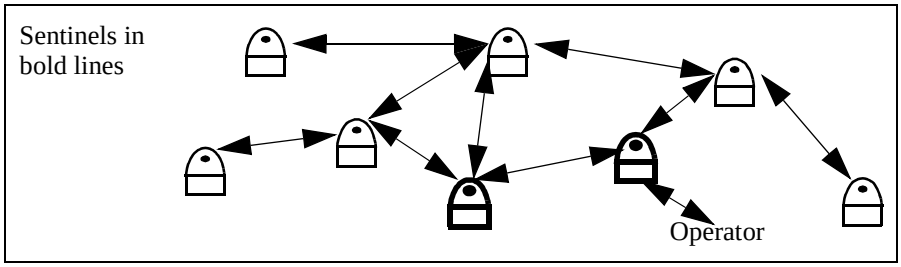

Figure 5. Sentinels in a society of agents

6 Example: Sentinels in Negotiation

Hägg and Ygge [7] present an example where DA-SoC agents are used for *load balancing* in power distribution. Distributor and customers operate on a spot market, where power is sold and bought[1]. The goal is to optimize the utilization of distribution lines, reducing load peaks, and avoiding overload in bottlenecks. Customers and distributor are represented by agents, negotiating about prices and the amount of power to sell. Suppose a customer's agent constantly underbids in negotiations, winning all contracts and thereby undertaking to reduce its power consumption more and more.

We now enter a sentinel who's task is to monitor agent communication, maintain models of the negotiating parties, and protect the negotiation process according to given guidelines. Soon it will detect that there is something wrong with the overly generous customer's agent. Depending on the guidelines, the sentinel may raise an alarm to an operator, or it can tell the distributor's agents to disregard that customer's agent for the moment.

Monitoring agent communication is quite simple using the semantic addressing scheme. Suppose the agents send bid messages on the following format[2]

```
(bid(<distributor agent> <bid time> sell(<amount> <price>
<duration>)))
```

If the sentinel now has the following interaction plan as part of its MoI

```
INTERACTION PLAN bid($VOID $time $$fact)
     BEL(SENDER $time $$fact)
```

1. A customer "sells" power if he offers to use less power than previously agreed upon in long term contracts.
2. The expressions and plans should be quite intuitive to read, but see [7] for a full outline of the language.

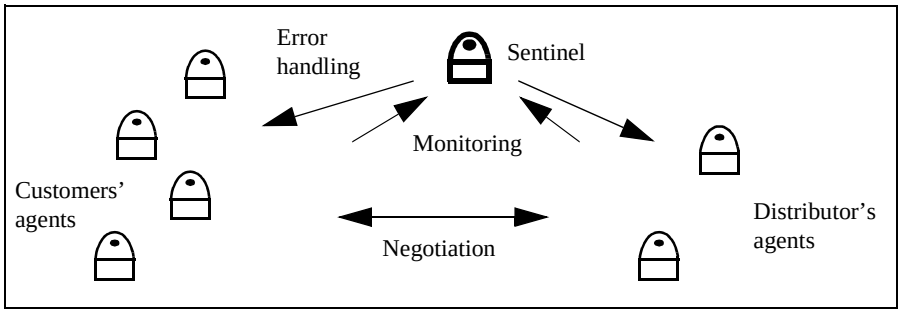

Figure 6. A sentinel supervising negotiation

it will pick up all bid messages in the society, and soon it will have a number of items of its world model as negotiation goes on:

```
BEL(c_ag_1 time_1 sell(am_1 pr_1 dur_1)
BEL(c_ag_2 time_2 sell(am_2 pr_2 dur_2)
BEL(c_ag_3 time_3 sell(am_3 pr_3 dur_3)
...
BEL(c_ag_n time_n sell(am_n pr_n dur_n)
BEL(c_ag_1 time_(n+1) sell(am_(n+1) pr_(n+1) dur_(n+1))
BEL(c_ag_2 time_(n+2) sell(am_(n+2) pr_(n+2) dur_(n+2))
...
```

These items become parts of the sentinels models of the customers' agents, and from them the sentinel can analyse the behaviour of those agents. A simple procedure for detecting underbids is the following agent plan:

```
AGENT PLAN ME check_for_underbids()
BEGIN
     ?BEL(ME $VOID price_limit($lower_limit
               $upper_limit)) // get limit values
     ?BEL($agent $time sell($amount $price
               $duration) // get list of monitored
                    // beliefs
     CHOOSE(MIN($price)) // choose belief with
                    // minimum price
     IF($price<$lower_limit) THEN // price is too low
     BEGIN
          SACT(start_error_check($agent NOW
               price_limit($lower_limit
               $upper_limit))) // ask the agent
                    // to start error checking
          BEL($agent $time underbid($amount $price
                    $duration)) // catalogue the event
          ~BEL($agent $time sell($amount $price
                    $duration)) // remove the bid
          GOAL(ME NOW check_for_underbids())
```

```
                    // check for more underbids
        END
        ELSE // there is no price that is too low
        BEGIN
             ?BEL(ME $VOID check_interval($interval))
                  // get the time interval for checking
             GOAL(ME LATER(NOW $interval)
                  check_for_underbids()) // set next goal
        END
   END
```

While the interaction plan picks up all bids and maintains models of bidding agents, the agent plan periodically checks for underbids, initiates error checking procedures, and catalogues error events. There can of course be other plans (e.g., checking for other limit transgressions). Though, if the sentinel should calculate more elaborately on the retrieved information, it would be nice to have access to, for example, a spreadsheet or a database program. The DA-SoC architecture is designed for this purpose; an agent's body is defined to be any program that can communicate with the agent head using a well defined protocol. To our testbed [7] we have ready made interfaces to MS Excel, Borland C++, and MS Visual Basic, allowing us to create body programs for virtually any purpose.

There are at least two important reasons for not integrating this control function with the distributor's agents. First, supervising the negotiation process is a strategic matter that aims at protecting the system's integrity. If strategies change, the sentinel can be given new guidelines without disturbing the normal operation. Second, normal operation would perform poorer if these and similar decisions should be taken into account during negotiation[1].

7 Example: Sentinels in Inconsistency Detection

Another example from power distribution is shown in Figure 7. The secondary substa-

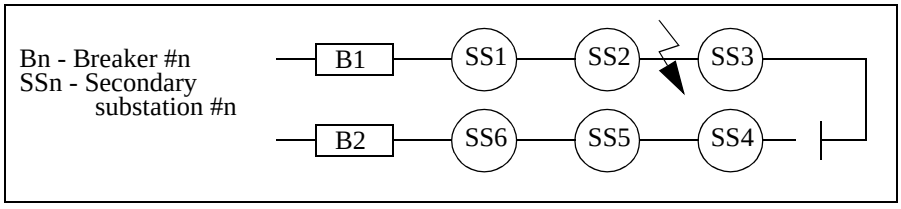

Figure 7. A short circuit between two secondary sub-stations.

tions, SS1 - SS4, get voltage through breaker B1 or breaker B2, depending on where the circuit is open. If, in the example, there is a short circuit between SS2 and SS3, B1 will automatically open and SS1 - SS3 will be without voltage. Agents representing

1. This may be implementation dependent.

the breakers and the secondary substations will now try to solve the problem, resulting in SS2 and SS3 opening local breakers to disconnect from the faulty line, SS3 and SS4 closing local breakers to supply SS3 with electricity from SS4, and B1 closing to supply SS1 and SS2. This is called *grid restoration*, and so far the agents solved the problem (see [6] how it is solved), and the grid has now changed to the situation in Figure 8.

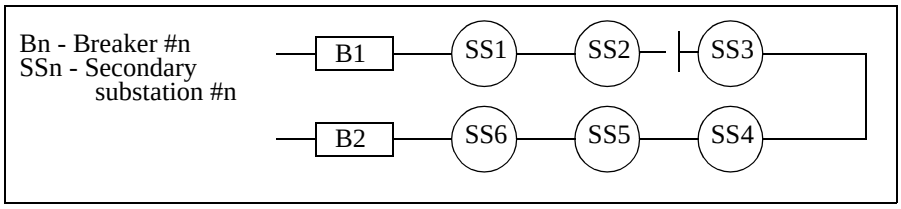

Figure 8. After grid restoration.

Suppose now that after these operations, $Agent_{SS3}$ in its world model holds the belief that its local breaker to SS4 is closed (which it should be) and the belief that its voltage level is high (which it also should be). At the same time $Agent_{SS4}$ believes that its local breaker to SS3 is open (which it should not be). Obviously there is something wrong. Either SS4 has a faulty sensor incorrectly indicating an open breaker to SS3, or SS4's breaker did not really close during the restoration and at the same time SS3 has a faulty sensor incorrectly indicating a high voltage level (or possibly a third explanation, maybe a programming error).

The problem is that neither $Agent_{SS3}$ nor $Agent_{SS4}$ has the information needed to detect the fault, less the nature of the fault; there is an inconsistency between the agents that requires global information to detect. If the second explanation is correct, SS3 is out of voltage, and customers will eventually call the distributor about it. According to the first explanation (and perhaps the most likely from what we know), the grid is operating correctly, but SS4's faulty sensor may cause failures later.

Observe the two levels of fault handling: Restoring the grid is the normal operation of the MAS, while taking care of the inconsistency is to make the restoration system fault tolerant. If we introduce a sentinel to the MAS and let it through agent communication use essential beliefs of the other agents as checkpoints, it can detect inconsistencies of this kind.

If the sentinel through monitoring, as in the previous example, or through direct agent communication, has the following items of its world model

```
BEL(SS3 time_1 local_breaker(SS4 closed))
BEL(SS3 time_2 voltage(high))
BEL(ME time_3 voltage_from(SS3 SS4)
```

but not the item

```
BEL(SS4 time_4 local_breaker(SS3 closed))
```

then the following agent plan will detect the inconsistency and start the error handling procedure.

```
AGENT PLAN ME check_local_breaker($source $dependant)
     // testing $source's local breaker to $dependant
BEGIN
     ?BEL(ME $VOID voltage_from($dependant $source))
               // does $dependant get
               // voltage from $source?
     IF SUCCESS THEN // yes
     BEGIN
          ?BEL($dependant $VOID voltage(high) // has
               // $dependant a high voltage level?
          IF SUCCESS THEN // yes
          BEGIN
               ?BEL($source $VOID local_breaker(
                    $dependant closed)) // is the
                              // breaker closed?
               IF NOT SUCCESS THEN // no
                    GOAL(ME NOW error_local_breaker(
                         $source $dependant)) // start
                              // error handling
          END
     END
END
```

The plan is invoked by the goal

```
GOAL(ME <time> check_local_breaker(SS4 SS3))
```

to check for the correctness of SS4's local breaker to SS3. This should be done periodically, and for all local breakers. Possibly the error handling procedure can, by retrieving more information, identify the nature of the fault, make the involved agents observant of it, and, in any case, report it.

The arguments for not letting the operating agents do the consistency checking themselves are the same as in the previous example. Yet another argument is that it would surely take more computing and communication resources to use distributed algorithms for this.

8 Conclusions and Future Activities

Conclusions

We have already stated that the problem of fault handling in MAS concerns the question of determinism. If a MAS should be tolerant to faults, the inherent property of non-determinism must be restricted. The philosophical question, indicated earlier, whether this restriction not violates the very concept of agenthood builds, as we see it, on a misconception: Agents' freedom and non-determinism from a system's perspective is an architectural issue. Even DA-SoC agents are modelled independently, and there are no limitations within the architecture that prevent agents from changing its behaviour or taking new roles in the society from time to time. On the other hand, the restrictions laid upon agents is a structural (or design) issue. By imposing a certain

structure onto the agent society, some specific behaviour can be guaranteed, and other behaviour can be prevented from.

Distributed AI often takes its metaphors from social sciences. Many human activities require cooperation. If the cooperation by any reason goes wrong, some "fault handling routines" must be executed. If the society is of any size, it is necessary to have some control structure, rules, etc. and supervisors to secure it. Here, the supervisors execute the "fault handling routines". This does not conflict with the concept of humanness (perhaps with the exception of extreme dictatorship). On the contrary, it is natural to humans, and it actually helps them in fulfilling their common tasks.

A similar discussion is that of centralized vs. decentralized control in distributed computer systems. It is sometimes assumed that decentralized control makes the system more flexible. It is, however, often difficult to find decentralized solutions that are both efficient and fault tolerant (Tanenbaum [12], pp. 27-29). In this respect, our sentinel approach means that we enter a centralized control structure. But, again, there are limitations on the centralized aspects: agents can change roles, and the control system may change over time.

The sentinel approach to fault handling indicate that total control of possible fault situations and global consistency are not always realistic. For the applications we consider, it would be far too expensive in terms of computation and communication, if achievable at all. Sentinels are a means for designers to guarantee partial consistency and graceful degradation in case of faults, given the priorities of the application.

Our approach allows the system implementer to program the functionality of the MAS and then design and implement the control system. The control system can also be changed with minimal disturbance to the application.

Abstraction levels

Finally, we will very briefly generalize the sentinel concept from mere fault handling to a generic way of entering abstraction levels in a MAS. This is subject for future research, but it is a natural extrapolation of the present work. Figure 9 shows five groups of agents. Each group can be defined from its global role (e.g., representing a certain functionality). Agent group C, D, and E may realize different operational proc-

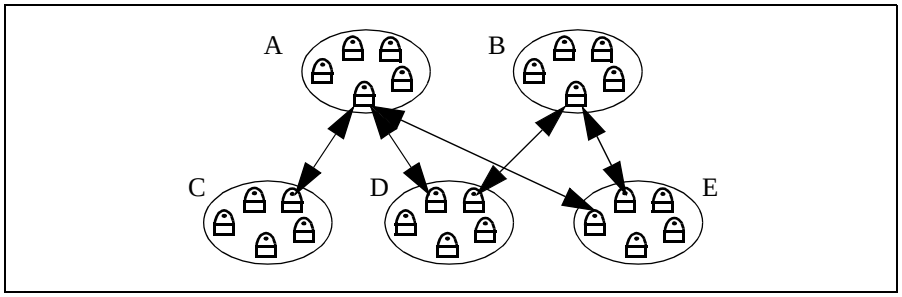

Figure 9. Five groups of agents representing abstraction levels

esses, while groups A and B may realize strategic functions, setting limits for the operational groups. If the application is telecommunication management, the operational

groups can control communication lines. Group A can be responsible for economic transactions, and group B can handle customer demands. There may also be other groups taking care of fault handling, security, etc. There may also be additional levels in the MAS. A similar approach is shown by Somers [11], developing HYBRID, a Multi-Agent System for telecommunication management.

Using the agent-oriented approach, these groups, as well as individual agents, may be developed or modified on the fly, with no or minimal disturbance to the rest of the system. With DA-SoC agents, programs written in traditional languages, or even legacy programs, can be used to realize basic functionality, glued together with the interaction of agent heads.

References

[1] Boman M., and Ekenberg L., Eliminating Paraconsistencies in 4-valued Cooperative Deductive Multidatabase Systems with Classical Negation, in *Proceedings of the Second International Working Conference on Cooperating Knowledge Based Systems*, University of Keele, England, June 1994, ISBN 0-952-17892-3.

[2] Burns A., and Wellings A., *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1990, ISBN 0-201-17529-0.

[3] Chen L., and Avizienis A., N-version programming: a fault-tolerance approach to reliability of software operation. In *Digest of Papers, The Eight Annual International Conference on Fault-Tolerant Computing*, Toulouse, France, 1978.

[4] Fisher M., Representing and Executing Agent-Based Systems, *Proceedings of the ECAI'94 Workshop on Agent Theories, Architectures, and Languages*, August 8-12, 1994, Amsterdam, The Netherlands.

[5] Gustavsson R., Akkermans H., Hägg S., Ygge F., Kozbe B., Lundberg C., and Carlsson B., *Societies of Computation (SoC) - A Framework for Open Distributed Systems, Phase II: 1995-98*, University of Karlskrona/Ronneby, Research Report 8/95, ISSN 1103-1581.

[6] Hägg S., and Ygge F., An Architecture for Agent-Oriented Programming with a Programmable Model of Interaction, in *Proceedings of the Seventh Annual Conference on AI and Cognitive Science '94*, Trinity College, Dublin, Ireland, September 8 - 9, 1994.

[7] Hägg S., and Ygge F., *Agent-Oriented Programming in Power Distribution Automation - An Architecture, a Language, and their Applicability*, Ph.L. Thesis, LUNFD6/(NFCS-3094)/1-180/(1995), Lund University, Sweden, May 1995.

[8] Hägg S., Adaptation in a Multi-Agent System through Semantic Addressing, in *Proceedings of the Workshop on Decentralized Intelligent and Multi-Agent Systems (DIMAS'95)*, Krakow, Poland, November 1995.

[9] Ingrand F. F., and Georgeff M., *Procedural Reasoning System, User Guide*, 1991, available from the Australian Artificial Intelligence Institute, 1 Grattan Street, Carlton, Victoria 3053, Australia.

[10] Societies of Computation (SoC), *Internet Home Page*, http://www.sikt.hk-r.se/~soc.

[11] Somers F., Intelligent Agents for High-Speed Network Management, in *Proceedings of The First International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, London, England, April 22 - 24, 1996.

[12] Tanenbaum A. S., *Distributed Operating Systems*, Prentice-Hall, 1995, ISBN 0-13-143934-0.

[13] Wittig T., ed., *ARCHON an architecture for multi-agent systems*, Ellis Horwood, 1992, ISBN 0-13-044462-6.