Demonstrating Social Error Recovery with AgentFactory

Robert Ross
Universitat Bremen
Germany
robertr@tzi.de

Rem Collier
University College Dublin
Ireland
rem.collier@ucd.ie

G.M.P. O'Hare
University College Dublin
Ireland
gregory.ohare@ucd.ie

Abstract

Exception handling is a well established method of error recovery through the alteration of plans in situ . *This method relies on recovery routines existing in advance, which - we argue - is simply shorthand for more detailed plan descriptions. However, in practice, agents rarely act alone in their environment - other agents may exist, and potentially provide help in times of need. We argue that* social error recovery *is a particular class of exception handling that allows agents to resolve erroneous situations that are beyond their direct control. In our opinion, agent oriented programming languages must directly provide agents with abilities like social error recovery. Consequently, we introduce revisions to the AgentFactory framework, and more specifically, the programming language (AF-APL), which facilitate the rapid development of agents with in-built social error recovery. The use of these abilities are illustrated via an example of a social error recovery scenario for a mobile robot working as an office assistant.*

1. Introduction

In real world applications, agents - be them software agents or autonomous robots - inevitably face erroneous situations that had not been planned for. Take for example an autonomous wheelchair that has been instructed to bring a patient to their room. If the wheelchair were to encounter a partially blocked corridor, then backtracking or re-planning would be necessary to circumvent the blockage; however, both of these options are expensive.

Exception handling provides a method for recovering plans like this *in situ* . However, exception catching typically requires that a plan has been conceived for dealing with the erroneous situation. We argue that this exception handling is simply a simply shorthand for more detailed plan descriptions, and reflects a single agent view of the world. A more powerful form of exception handling - used by humans all the time - is to take advantage of the social environment to recover from errors. Social error recovery acknowledges the presence of other agents, and their skills, to perform tasks. In Section 3 we look at how social error recovery as a particular class of error handling for agents.

To increase the value of agent oriented programming - and indeed to distinguish it from *Middleware* or *Component Based Software Engineering* - it is our belief that agents must be endowed with default reasoning and social skills. Social error recovery is one of these key skills, and in this paper we show how it can be provided through appropriate language constructs and the design of suitable roles. We detail the approach with the AgentFactory Framework [2]; first, introducing relevant features of the framework in Section 2, before outlining the implementation of social error recovery in Section 4. To demonstrate the use of the mechanisms, Section 5 describes how a office assistant robot used these default skills to solicit help from a human user while performing a simple delivery task.

2. The AgentFactory Framework

The AgentFactory Framework [2] is a complete agent prototyping environment, providing development facilities similar to those of OAA [6] and JACk [1]; but unlike these environments AgentFactory is based on a formal agent oriented programming language similar to Shoham's AgentO [8], Goal Directed 3APL [3], and Wooldridge's formulations for intentional rational agents [9]. In the following two sections, we focus on two parts of AgentFactory which are central to the implementation of social recovery - namely the AF-APL language constructs, and AgentFactory's roles as a means of inheritance. As mentioned, the framework also provides a large suite of tools for the development and deployment of AF-APL agents, including: prototype agent designs, language interpreters and compilers, a fully functional IDE and agent and platform viewers. (See Figure 1). These tools are beyond the scope of this paper, but have previously been addressed in [**?**].

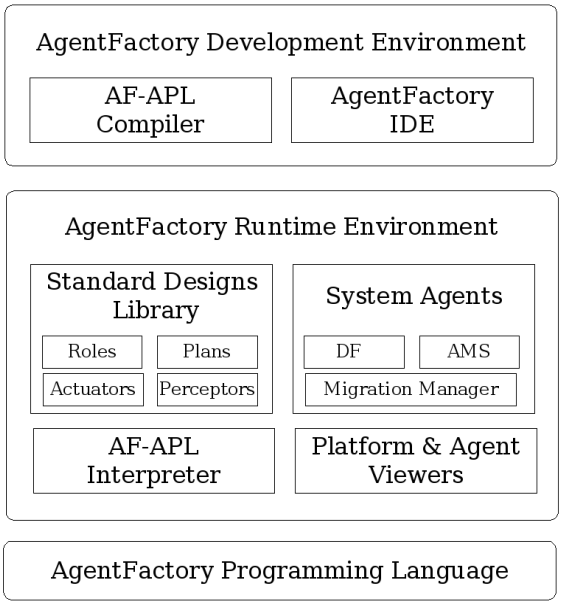

Figure 1. The AgentFactory Framework

2.1. The AgentFactory Programming Language

At the core of AgentFactory lies the AgentFactory Programming Languages (AF-APL), which has been formally presented elsewhere [2]. Here we provide an informal review of the main language constructs to aid the explanation of the social error recovery implementation.

2.1.1. Beliefs In AF-APL a belief is represented with the BELIEF construct. For example, the belief that Rem is happy may be represented with the statement:

```
BELIEF(happy(Rem))
```

By default, the beliefs of an agent only persist in the agent's belief set for one iteration of the agent's execution cycle. This behaviour can be changed through the use of the NEXT, UNTIL, or ALWAYS constructs as required. For example, we can represent the belief that Rem always likes beer, as follows:

```
ALWAYS(BELIEF(likes(Rem, beer)))
```

2.1.2. Belief Rules Belief Rules can be used to allow an agent to infer new beliefs from already existing beliefs. Therefore, the agent can decide if a friend is allowed to drink beer with the following Belief Rule:

```
BELIEF(friend_age(?friend,?years))
& BELIEF(greaterThan(?years,18))
   => BELIEF(canDrinkAlcohol(?friend));
```

The left hand side of a belief rule is referred to as a belief query, and is a conjunctions of positive or negative current beliefs containing free or bound variables. Belief queries can reference either the agent's belief set directly, or can reference three mental functions - equals, greaterThan, and lessThan - which have been defined to allow basic equality testing. Users are not allowed to define new mental functions, since poorly designed functions could seriously impact on the agent's processing model.

2.1.3. Perceptors An AF-APL perceptor enables an agent to acquire beliefs about its environment. The PERCEPTOR construct is used to declare a perceptor with a particular identifier and external code implementation. Each perceptor is fired in its own thread once per agent execution cycle, and any resultant beliefs are then added to the agent's belief set. For example to define a perceptor for receiving FIPA messages with a Java implementation of the low level perceptor code, one might use this construction:

```
BEGIN{PERCEPTOR}
IDENTIFIER ie.ucd.af.fipa.fipaReceive();
CODE ie.ucd.af.fipa.Receiver.class;
END{PERCEPTOR}
```

2.1.4. Actuators An actuator represents the most basic action an agent can perform. As with perceptors, actuators are defined in terms of a particular identifier, with an implementation provided in an external programming language such as Java. In addition, perceptors have pre and post conditions defined; these conditions determining what must be true for an actuator to be invoked, and what should be true when an actuator has completed. To avoid locking of the agent execution cycle, actuators are fired asynchronously. An actuator for sending FIPA messages might be defined as follows:

```
BEGIN{ACTUATOR}
IDENTIFIER ie.ucd.af.fipa.fipaSend(?fipa_msg);
PRE BELIEF(TRUE);
POST BELIEF(send_success(?fipa_msg))
     | BELIEF(send_failure(?fipa_msg));
CODE ie.ucd.af.fipa.Sender.class;
END{ACTUATOR}
```

The above example presents an actuator from a *procedural* viewpoint. AF-APL also allows actuators to be defined from a behavioural viewpoint - where an actuator is not expected to return by itself, but which must be explicitly stopped when some belief is entailed by the agent's belief set. The BEHAVIOUR keyword is used to declare such actuators, which can then only be used within the DO_BEHAVIOUR_UNTIL plan construct.

2.1.5. Plans Plans provide a mechanism for actuators to be sequenced to perform complex actions. Plans are built out of actuators and other plans through the application of the AF-APL planning constructs which follow: SEQ, the sequence construct; PAR, the parallel construct; AND, the random order construct; OR, the

non-deterministic choice construct; XOR, the deterministic choice construct; TEST, the belief query construct; IF ELSE, the cognitive choice construct; FOREACH, the universal quantification construct; DO_BEHAVIOUR_UNTIL, the behaviour controller construct; COMMIT, the commitment description construct; ADOPT_GOAL, the goal declaration construct; ACHIEVE_GOAL, the goal achievement construct; and TRY_RECOVERY, the error recovery construct.

Plans can be defined in AF-APL using the plan construct, For example, a (naive) plan to make a cup of tea can be represented as follows:

```
BEGIN{PLAN}
IDENTIFIER ie.ucd.assitant.makeTea();
PRE BELIEF(TRUE);
POST BELIEF(made_tea);
BODY SEQ( AND(getTea,getCup,getMilk),
          XOR(SEQ(getKettle,fillKettle,
                                boilKettle),
              SEQ(getPot,fillPot,boilPot)
              ),
          AND(addTea,pourWate),
          PAR(stir,addMilk),
          tasteTea
        );
END{PLAN}
```

2.1.6. Commitments A commitment is a promise made by an AF-APL agent (to itself or other agent) to attempt an action. In other words, a commitment is the mental equivalent of a contract. As such, it specifies the course of action that the agent has agreed to; to whom this agreement has been made; when it must be fulfilled; and under what conditions the agreed course of action becomes invalid (i.e. under what conditions the contract can be breached). For example, we can represent that Rem has committed himself to eat biscuits at 11am (as long as he believes that he has no lunch plans) as follows:

```
COMMIT(Rem,
       11:00,
       !BELIEF(lunch_plans),
       eat(biscuits)
       )
```

The rational realisation of commitments is at the heart of the AF-APL execution and reasoning model. An AF-APL agent may have any number of commitments at any time. These commitments, stored in the agent's commitment set, are normally acquired by the agent in two ways: first, through Commitment Rules, which allow an agent to adopt (primary) commitments if its belief set entails a condition; second, Commitment Management, where commitments are refined to structures of more primitive commitments through the application of plans and reasoning.

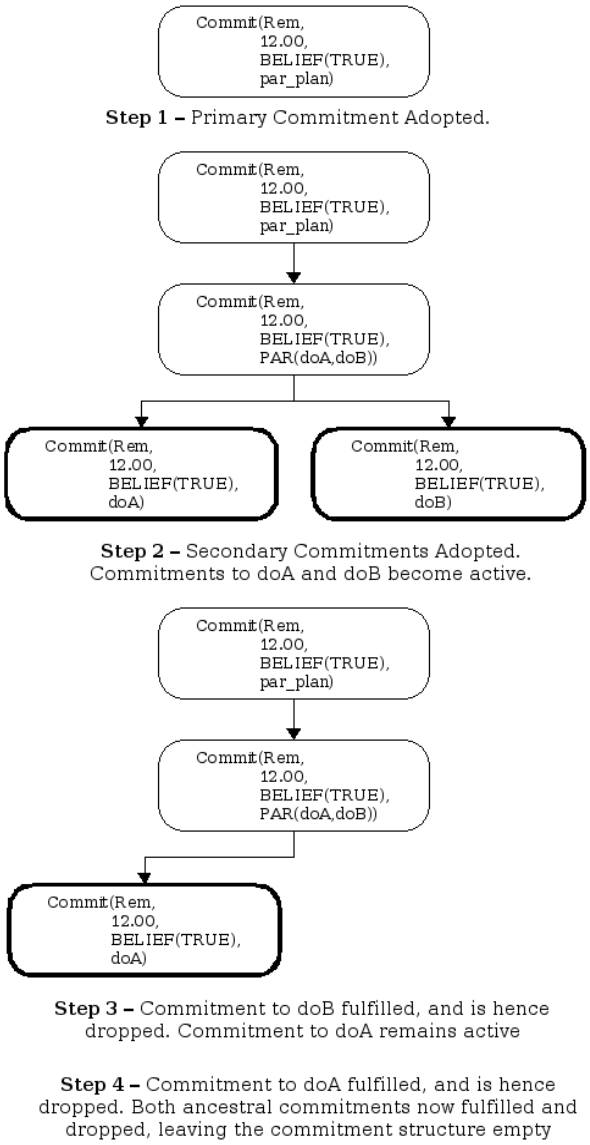

Figure 2. Commitment Structure Management

The commitment management system takes these commitment structures and attempts to achieve the commitments through the appropriate use of actuators and reasoning. Figure 2 shows the most simple example of a commitment structure's generation and refinement, Commitment structures can be directly reasoned about by an agent, thus allowing them to rationally choose courses of actions for the future. This, plus commitment managements treatment of the FOREACH, DO_BEHAVIOUR_UNTIL, COMMIT, and TEST constructs makes commitment management quite different from a simple transversal of a tree based plan structure.

2.1.7. Commitment Rules As mentioned above, the commitment rule allows an agent to rationally decide on a course of action based on its mental state. Take for example an agent who is hungry and has a piece of fruit. Under these circumstances, a rational action for the agent to take, would be to eat the fruit. We can specify this as a commitment rule in AF-APL with a commitment rule construct:

```
BELIEF(hungry(me))
& BELIEF(haveFruite(?someFruit))
  => COMMIT(?Self,
            ?now,
            BELIEF(TRUE),
            eat(?fruit)
            );
```

2.1.8. Reactive Rules The Reactive Rule allows an agent to invoke an action directly based on what it believes to be true. The reactive rule is simpler than a commitment rule, but is hence less robust. A reactive rule can be used to code basic reflective behaviours into the agent's design. For example, A reactive rule to dodge an obstacle blocking a robot's progress could be encoded with the REACTIVE_RULE construct as follows:

```
BELIEF(blocked(ahead))
& BELIEF(moving(forward))
     => EXEC(dodgeObstacle);
```

Reactive rules are different from their big brothers, the commitment rule, in a number of ways - including: Static binding, reactive rules are bound at runtime to a specific plan implementation, whereas choices between plans and actuators can be made by commitment management at runtime; No deliberation, the agent cannot reason about its reactive rules; Priority Execution, reactive rules take precedence over commitment rules; More basic formulation, reactive rules may only contain actuator identifiers or the SEQ, XOR, OR, PAR, and AND plan constructs.

2.1.9. Goals In AF-APL, a goal is a state of the world - or set of beliefs - that an agent wishes to bring about. Agents may adopt goals using either the ADOPT_GOAL or ACHIEVE_GOAL constructs. ADOPT_GOAL adds a goal to the agent's goal set, returning immediately. ACHIEVE_GOAL also adds a goal to the goal set, but does not return until the goal is achieved. To illustrate the use of ADOPT_GOAL, let us consider a robot which has been requested to close the door. The agent can adopt the general goal of causing the door to be closed, with the following commitment rule:

```
BELIEF(requested(?someAgent,closeDoor)) =>
        COMMIT(?someAgent,
               Now,
               BELIEF(TRUE),
               ADOPT_GOAL(door(closed)));
```

Once adopted into the goal set, an agent will use means-end reasoning to attempt to determine a plan that can achieve the goal. If such a plan can be determined, a secondary commitment to that plan structure will form as a child of the original goal; fulfilling the plan then causes the original commitment to be dropped. The means-end reasoning process is, of course, performed asynchronously to guarantee the continued reactivity of the agent.

2.1.10. Roles AF-APL provides a form of inheritance in agent design through the use of explicitly defined roles. AF-APL Roles allow a collection of actuators, perceptors and other agent components to be grouped together into an agent prototype. These agent prototypes can then either be instantiated directly into agents, or included in other agent designs. For example, the AgentFactory runtime provides a number of pre-defined roles, including one that implements basic functionality for FIPA compliant communication. This role can be included into a new agent design with the INCLUDE_ROLE construct:

```
INCLUDE_ROLE ie.ucd.core.fipa.role.FIPARole;
```

In addition to providing basic inheritance, roles can also be switched on and off at runtime, thus allowing an agent to only use certain commitment or reactive rules in certain situations. The FIPA role is essential to allowing a group of agents to work together as a whole. But to allow the explicit error recovery a sociallyAware role was also necessary, this role is introduced in Section 4.2.

3. Error Handling for Intelligent Agents

In order to achieve complex goals, Intentional Agents execute plans which essentially break a main goal into a structure of sub-goals which can be achieved through action. However in the real world, the actions performed by any agent can fail for a multitude of reasons. The impact of this action failure is traditionally viewed as having three levels of severity:

- **Anticipated Error** An action has failed, but the agent can backtrack through the plan to attempt to achieve the same goal through an alternate course of action.
- **Exceptional Error** An action has failed, but the current plan was not expressive enough to cope with this failure. Re-planning is necessary to formulate a strategy to achieve the original goal.
- **Unrecoverable Error** An action has failed. But the current plan is not expressive enough to cope with the error, nor is re-planning capable of formulating a strategy to achieve the original goal.

Although this three tier fault model does present a great deal of flexibility in the ways goals may be achieved, it reflects an old single entity view of the world. In reality other

agents may exist in the environment and may be capable of offering help in times of need. An agent encountering a problem should attempt to solicit help from those in it's proximity. To this end some agent execution models have attempted to introduce other agents in the community to re-plan under unusual circumstances [4]. However it is often too time consuming or computationally expensive to make a complete plan revision. We introduce social error recovery as a new level of error recovery:

- **Socially Recoverable Error** An action has failed. But, through appropriate assistance from other agents within its environment, the agent can continue on with the original plan.

This *Error Recovery Through Social Interaction* can be attempted before backtracking or re-planning is necessary, thereby reducing the cost of the agent repairing its plan.

4. Social Error Recovery in AgentFactory

We now show how social error recovery was provided for in AgentFactory. Before detailing the *SociallyAware* role that was created, we describe AF-APL's TRY_RECOVER construct that is used as the basis for general error recovery situations.

4.1. The TRY_RECOVER Construct

The TRY_RECOVER plan construct is intended for more than simply graceful degradation of plans - recovery plans used with the operator should attempt to remedy a plan in situ. This approach reflects our belief that it is often very wasteful for an agent to give up on a plan, and that agents should put more effort into recovering from unpredicted situations. The approach taken here is similar to the *Interrupt Mechanism* introduced to 3APL in [5], but produces a simpler implementation model which does not effect the semantics of the underlying logic.

TRY_RECOVERY is used to indicate whether an action - or indeed a complete plan structure - should have error recovery mechanisms associated. The construct takes two arguments: first, the action or plan body to be monitored for failure; second, the plan which is to be used for recovery. For example, the plan body SEQ(doA,doB) can be tagged for recovery by the *social_recovery* plan as follows:

```
TRY_RECOVER(SEQ(doA,doB),
            social_recovery)
```

Unlike the *try catch* exception handling mechanisms in Java, TRY_RECOVER attempts to repair an erroneous situation and return the agent to finish the original plan. Therefore it is possible to re-write the construct in a more basic form by making the following substitution for each action scoped by the construct:

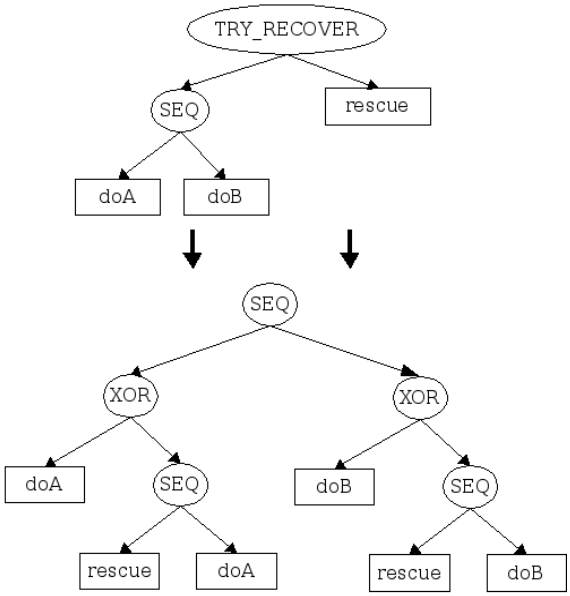

Figure 3. Equivalence of TRY_RECOVER to a XOR SEQ construction.

```
act_ident -->
   XOR(?act_ident,
       SEQ(?recovery_plan(?act_ident),
           ?act_ident))
```

This replacement allows an agent to attempt an action. If the action fails, the agent will then try the recovery of the situation; this failing, the action in general fails. However, if the recovery plan succeeds, then the agent will attempt the original action once again. Failure on this occasion leads to complete failure of the action. To illustrate Figure 3 shows, in a tree representation, how applying TRY_RECOVER construct effectively alters the simple plan body SEQ(doA,doB).

This AF-APL construct allows any number of recovery plans to be defined. Since AgentFactory is intended for the actual development of agents, the Runtime Environment provides recovery plans for use by agents. One of these is the social_recovery(?action) plan, and it forms part of the SociallyAware role described in the next section.

4.2. The SociallyAware Role

Where AF-APL and all of its construct give a logical model for the execution of agents, useful agents often need far more than a well defined model. We now present the skills that were provided to agents care of the SociallyAware role. It should be noted that most of the skills mentioned be-

low have been discussed in greater detail, but a more primitive state, in [7].

The SociallyAware role defines a number of primitive actions, plans, and commitment rules which can be used to give an agent a notion of social awareness. The role made use of a FIPA core role that provided basic FIPA compliant communication mechanisms. It is our belief that software agents operating in an open society should never be staticly linked; therefore, AgentFactory provides FIPA compliant Directory Facilitation and Agent Management Services to allow agents to register and request information about other agents. For a truly open society this process must be recurrent; therefore, by default, all SociallyAware agents perform a registration/inquiry cycle every two minutes. Even when agents have been successfully introduced to relevant partners, there is still a need for well defined communication protocols. FIPA provides a defined semantics for speech acts, which has previously been hard-coded into agent programming languages by others. Rather then performing such hard coding, we provide basic *dialog plans* which can be used by an agent to comply with FIPA semantics.

With the basic skills for operation in an open agent society in place, it was then necessary to construct actuators, plans, and commitment rules which could allow agents to negotiate about their own abilities, needs, and commitments. Hard-coding these skills for particular cases would haven been easy, but making the abilities more general - such that they can be inherited and used by any agent in any domain - was a more difficult process. To aid the negotiation process, an introspection actuator, which allows an agent to decide what decisions it would make *if* a particular request was made, was created. This actuator uses a partial clone of the agent's mental state to make a guess at its future actions. This is of course only a guess, since the agent cannot be held responsible for external changes which might effect its future decision process. Introspection could then be used in a general negotiation process between agents, where one agent needed a task performed by another agent. A body of plans and commitment rules were then created to be used by agents in the negotiation process. Some of these mental objects are used by those who need help, others used by those who can give help. For illustration, Figure 4 shows the basic form of the social_recovery plan.

```
BEGIN{PLAN}
IDENTIFIER social_recovery(?act);
PRE  BELIEF(TRUE);
POST BELIEF(TRUE);
BODY
FOREACH(BELIEF(friend(?agent)),
     XOR(SEQ(request(?agent,
                    capable(?act)),
           awaitMsg(inform(capable(?act)),
                            ?agent,50),
           request(?agent,?act),
           awaitMsg(inform(complete(?act)),
                             ?agent,50)
            )));
END{PLAN}
```

Figure 4. The social_recovery(?act) plan.

5. Demonstration : Social Recovery in an Office Environment

The demonstrator looks at the provision of an office assistant. The scenario sees Bodan the robot as a simple office assistant working in the Eurocom Building in University College Dublin. Bodan needs to take care of typical *gopher* jobs, including the delivery of packages and cups of tea. To keep costs to a minimum, Bodan is a simple robot, with minimal sensing and actuation capabilities. Specifically, Bodan can only sense through the use of sonar, bumper, sound and direct command messages via a wireless TCP/IP connection.

In the example, Robert has made Anna a cup of tea, but being lazy, Robert places the tea on Bodan, and asks him to bring the tea to Anna in her office. Having nothing better to do, Bodan constructs a movement plan and starts making his way from the tea area to Anna's office. Unfortunately Bodan finds something blocking his path, namely the door to Anna's office is closed. Bodan's movement plan made no contingencies for such an eventuality, and being a simple robot, he is not able to open the door by himself. To a robot running a simple plan sequencer, this would lead to a failure of the plan, and (perhaps) a notification to Robert that Bodan cannot meet the original request. We will now examine show how social error recovery in the AgentFactory agents, allowed Bodan to recover from this situation.

5.1. Demonstrator Setup

The environment for the demonstrator was the top floor of the Eurocom Building, University College Dublin. A simple schematic of this floor, along with Bodan's trajectory, is presented in Figure 6. It should be noted, that this schematic is intended for documentation purposes only, and is not derived from an internal representation used by Bodan.

The demonstrator was implemented with a Nomadic Technologies Scout II robot as shown in Figure 5. The robot comes with integrated sonar and bumper sensors, and is controlled on a low level by a Motorola 68332 based board. High level control for the robot was provided by two laptop computers which sat directly on the Scout. The first of these computers was a Dell Latitude C-840. The C-840 ran

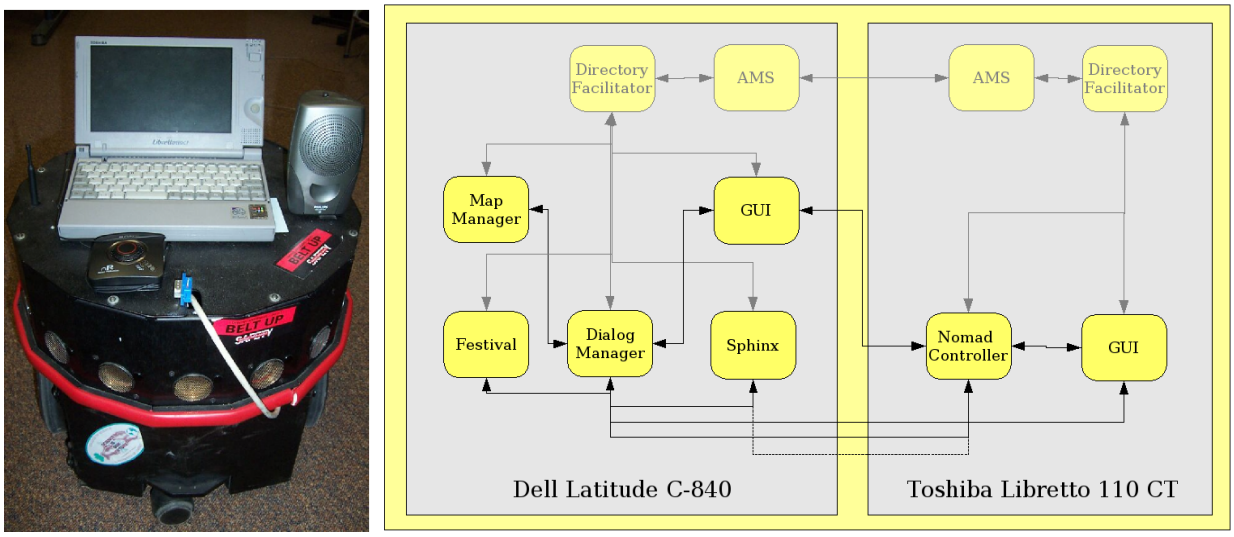

Figure 5. ***On the left we see Bodan, outfitted with sound equipment and off-board computers. On the right, Bodan's MARC based Architecture, showing agents which worked on two separately PCs, connected via wireless lan.***

SuSE Linux 8.1 and had in-built wireless lan support. The Dell's basic audio capabilities were enhanced through the use of external speakers and a Sony ECM 500 long range microphone. The second computer was a Toshiba Libretto 110 CT. This more basic model, ran Red Hat Linux 7.1, and was given wireless lan support with a Cisco Systems PCM350 PCMCIA card.

In addition to the system agent's provided by AgentFactory for Directory Facilitation and Agent Management Services, the demonstrator required a number of control agents to be developed. These agents, shown in Figure 5, all inherited the SociallyAware role, and provided everything from motor control to dialog management support.

- **Festival Speech Generation Agent (Festival)** - Using the Festival Speech Generator from the University of Edinbrough, this agent gave Bodan a voice.
- **Sphinx Speech Recognition Agent (Sphinx)** - Based on the Sphinx-2 speech recognition toolkit, and using modified language model generation scripts from the XVoice project for Linux, this was the most stable and versatile of the speech recognition agents created.
- **Dialog Management Agent (DMA)** - Although only one agent within the complete MARC architecture, this agent does provide the main dialog management between the robotic agent and users in the demonstrator. This agent was wrote solely in AF-APL. The DMA acted as a proxy agent for human users, thus allowing the social recovery plan to be used with humans as if it was being used with other software agents.
- **Nomad Controller Agent (NCA)** - Using a core provided by Nomadic Technologies, this agent took care of basic movement and sensor control on a Nomadic Scout II robot.
- **GUI Interface Agent (GUI)** - Noisy environments can often render speech recognition useless; for this reason a GUI interface agent was also created.
- **Map Management Agent (MMA)** - A rudimentary map management agent was also constructed for the demonstrator.

5.2. Operational Discussion

When receiving the request to deliver the cup of tea through speech recognition, the DMA uses a pre-compiled static plan to: confirm the robot's willingness to undertake the task; find the route to the office; and request the NCA to commit to reaching the route. The route was obtained from the MMA, which uses an internal representation of the office environment. With a route available, the DMA requested the NCA to commit to the movement plan. Since the NCA does not have any conflicting requests for movement, it immediately constructs a complete plan, which includes the use of the social_recovery mechanisms.

Bodan turns and begins moving towards his destination. The NCA manages commitments to each step in the movement plan, until its sonar perceptor detects that there is an obstacle directly blocking its path. This perception triggers a reactive behaviour which immediately stops Bodan in his

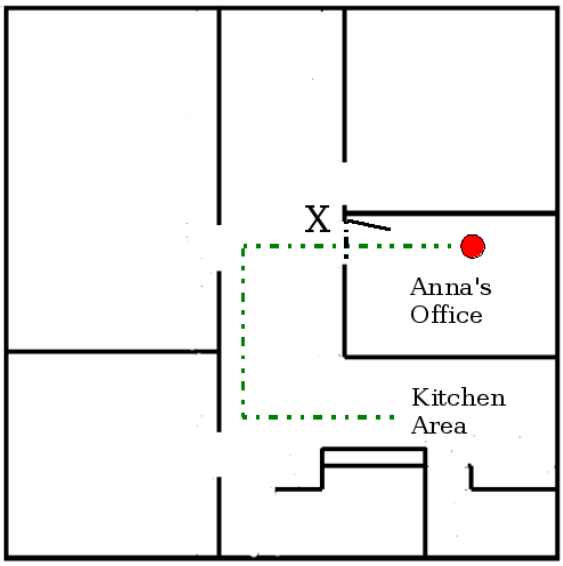

Figure 6. ***Bodan's trajectory around the top floor of the Eurocom Building, UCD, showing the initially closed door at X.***

tracks. Unfortunately this means that the commitment to move forward - which is now being managed - is at risk of failing. Since Bodan's movement plan does not include any other way of moving into the office, the plan to deliver the tea would normally fail at this point. Instead the social_recovery plan is triggered, causing the NCA to request help from all other agents on the platform.

None of these can give direct help, but the DMA, acting as a proxy for any external agents, relays the help request on to any human users present. This leads Bodan to utter a request for help. If a confirmation of help is not received within a given timeout period, the request would have been interprter as a failure. However, in this case, a passing person hears the request for help, and opens the door to the office, confirming to the robot that the door is now open. This confirmation, relayed back through the DMA to the NCA, allows the robot to continue along with its original plan, achieving his goal original commitment.

6. Conclusions & Future Work

We believe that for agent oriented approaches to become popular, language support and tools must be available to give agents inherent intelligence to separate them from other computational paradigms. The ability to recover from erroneous situations by asking other agents for help, is one skill that can be provided directly in AO frameworks. Furthermore, as we saw in the demonstrator, it is possible to allow artificial agents with these skills to interact with humans to recover plans that are going wrong - although, in such cases it is essential that the plan description used by the agent be such that natural language translation is possible. AgentFactory, and in particular, its programming language (AF-APL) and set of default roles were easily extended to provide these behaviours.

There is a large amount of work which can continue from this starting point. First, in particular with regard to the robot scenario, we will be investigating how linguistic ontologies can be used to allow a representation of actions performable by the robot, to be translated to natural language. Second, would be the testing of social error recovery in groups of robotic agents. With results from these extension, we hope to further develop the default skills available to AgentFactory agents, thus allowing them to truly act intelligently in real world scenarios.

Acknowledgements - We gratefully acknowledge the support of Enterprise Ireland through grant No. IF/2001/02, SAID. We also gratefully acknowledge the support of the DFG through the SFB/TR 6023 project on Spatial Cognition - Subproject I3-SharC.

References

[1] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Ack intelligent agents components for intelligent agents in java. *AgentLink Newsletter*, 1, 1999.

[2] R. W. Collier. *Agent Factory: A Framework for the Engineering of Agent Oriented Applications*. PhD thesis, University College Dublin, 2001.

[3] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A programming language for cognitive agents: Goal directed 3apl. In *Proceedings of AAMAS 03*, 2003.

[4] A. Haddadi. *Communciation and Cooperation in Agent Systems: A Pragmatic Theory*. Number 1056 in Lecture Notes in Computer Science. Springer-Verlag: Heidelberg, Germany, 1996.

[5] K. V. Hindrikis, F. de Boer, W. van der Hoek, and J. J. Meyer. Failure, monitoring and recovery in the agent langauge 3apl. In *Proceedings of AAAI 1998 Fall Symposium on Cognitive Robotics*, 1998.

[6] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[7] R. Ross. Marc - applying multiagent systems to service robot control. Master's thesis, University College Dublin, 2004.

[8] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[9] M. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robots and Autonomous Agents. The MIT Press, Cambridge, Massachusetts, 2000.