Agent Oriented Software Engineering

Yohannes S. (MSc.)

CONTENTS

INTRODUCTION

We need open architectures that continuously change and evolve to accommodate new components and meet new requirements. More and more software must operate on different platforms, without recompilation and with minimal assumptions about its operating systems and users. It must be robust, autonomous and proactive. These circumstances motivated the development of Agent Oriented Programming.

The objective of Agent Oriented (AO) Technology is to build systems applicable to real world that can observe and act on changes in the environment. Such systems must be able to behave rationally and autonomously in completion of their designated tasks. AO technology is an approach for building complex real time distributed applications. This technology is build on belief that a computer system must be designed to exhibit rational goal directed behaviour similar to that of a human being. AO technology achieves this by building entities called agents which are purposeful reactive and communication based and sometimes team oriented.

There are different programming methods. Object Oriented Programming is the successor of Structured programming. Agent oriented programming can be seen as an improvement and extension of object oriented programming. Since the word “Programming” is attached it means that both concepts are close to the programming language and implementation level. The term “Agent-Oriented Programming” was introduced by Shoham. So this AOP is a fairly new programming paradigm that supports societal view of computation. In AOP objects known as agents interact to achieve individual goals. Agents can be autonomous entities, deciding their next step without the interference of a user, or they can be controllable, serving as mediatory between user and another agent.In AOP programming is performed at abstract level. *Agent-Oriented Software Engineering* is being described as a new paradigm for the research field of *Software Engineering*. But in order to become a new paradigm for the software industry, robust and easy-to-use methodologies and tools have to be developed. The term AOP was suggested by Shoham.

ESSENTIALS OF AN AOP

Shoham suggests that AOP system needs each of three elements to be complete

1. A formal language with clear syntax for describing the mental state. This would likely include structure for stating beliefs, passing messages etc.
2. A programming language in which to define agents. The semantics of this language should be closely related to formal language.
3. A method for converting neutral applications into agents. This kind of tool will allow an agent to communicate with a non-agent .

AGENT

Before we go in detail into the programming side of AOP lets have a look into what the agent concept is?

Agent is a software process that meets the conventions of OAA (Open Agent Architecture) society. An agent is an abstraction that enables to model a system more easily. With this paradigm, a complex system can be modeled as a set of entities, the"agents". Agent satisfies this requirement by registering the services it can provide in an acceptable form by being able to speak the Inter agent Communication language (IACL), and by sharing the functionality common to all OAA Agent such as the ability to install triggers, manage data in certain ways etc. An agent according to Shoham is "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices and commitments". This definition is cryptic at best and useless at worst. In agent Oriented programming there is no single agreed definition of entities that are dealt with it.

"What makes any hardware or software component an agent is precisely the fact that one has chosen to analyze and control it in these mental terms." – Yoav Shoham

Generally agent is a hardware or (more usually) software based computer system with the following properties

*Autonomy: The ability to operate without direct intervention of humans or others and have some kind of control over their actions and internal state.

*Social-ability: The ability to interact with other agents through some kind of agent-communication language.

*Reactivity: The ability to understand the environment and respond regularly to changes that occur in it. Environment is usually thought to be physical, unpredictable, and containing other agents

*Pro-activeness: The ability to exhibit goal-directed behaviour by taking the initiative instead of just acting in response. An agent is *pro-active* because it acts according to a goal and not simply in reaction to the environment. It is capable of behaviors (directed by the objective to carry out) taking the initiative rather than to wait for orders stating to it what it must do.

Some other attributes of agency are

*Mobility: The ability to move around an electronic network.

*Veracity: The assumption of not communicating false information knowingly.

*Benevolence: The assumption of not having conflicting goals.

*Rationality: The assumption of acting with a view to achieve its goals, instead of preventing them.

So we can say that an agent is an intelligent agent; i.e. it is a piece of autonomous software. Is there any special meaning in words intelligent and agent? An agent shows intelligent behaviour, which is the ability to select actions based on knowledge. The word agent means one who is authorized to act for or in place of another.

Examples of software agents

1. The animated paper clip in Microsoft office.
2. Computer viruses (example of destructive agent)
3. Artificial players or actors in computer games and simulation.
4. Web spiders:- Collect data to build indexes to used by a software engine.

So the agent needs to have

Belief: This represents the agents current knowledge about the world, including information about the current state of the environment inferred from perception devices and messages from other agents, as well as internal information.

Desire: This represents a state, which the agent is trying to achieve, such as safe landing of all planes currently circling the airport, or the achievement of a certain financial return on the stock market.

Intention: This is the chosen means to achieve the agents desires and is generally implemented as plans. Plan may be thought of as procedures that come with preconditions and intended outcomes.

Plan: The means of achieving desires with the options available to the agents.

In agent systems the agent not only needs to know what has to be achieved , but also needs to be able to take appropriate action inorder to ensure that the desired state of the world is achieved .So an agent system requires not just abductive reasoning, but also a suitable notion of agent.

AGENT CLASSIFICATION

So far we discussed about what an agent is. Now we may go to the different types of agents. The presence and absence of the above mentioned attributes is the basis of classification. There is weak and strong notion of agency. Agent theories depend on these two notions of agency. In the weak notion of agency, agents have their own will (autonomy), they are able to interact with each other (social ability), they respond to stimulus (reactivity) and they take initiative (pro-activity). Weak notion requires only a white box on agents. It defines agents only in terms of observable properties. In strong notion of agency weak notion of agency is preserved, in addition agents can move around (mobility), they are truthful (veracity), they do what they are told to do (benevolence) and they will perform in an optimal manner to achieve goals (rationality). i.e. in strong notion of agency agent is modeled in terms of mental concepts. These mental concepts should have an implicit representation within the implementation of agents. This strong notion forces a white box on agents. . In McCarthy's view, ascribing mental qualities is a means of understanding and of communication between humans, i.e. it is a purely conceptual tool that serves the purpose of expressing existing knowledge about a particular program or its current state.

Shall we go detail into the idea of mental concepts

"All the ... reasons for ascribing belief's are epistemological; i.e. ascribing beliefs is needed to adapt to limitations on our ability to acquire knowledge, use it for prediction, and establish generalizations in terms of the elementary structure of the program. Perhaps this is the general reason for ascribing higher levels of organization to systems."

To illustrate why this point of view is reasonable, McCarthy uses the example of a program that is given in source code form. It is possible to completely determine the programs behavior by simulating the given code, i.e. no mental categories are necessary to describe this behavior.

Why would we still want to use mental categories to talk and reason about the program?

In the original paper, McCarthy discusses several reasons for this. In the following list, I have selected those reasons that seem to be most relevant to me.

(1) The programs state at a particular point in time is usually not directly observable. Therefore, the observable information is better expressed in mental categories.

(2) A complete simulation may be too slow, but a prediction about the behavior on the basis of the ascribed mental qualities may be feasible. Ascribing mental qualities can lead to more general hypothesis about the programs behavior than a finite number of simulations.

(3) The mental categories (e.g. goals) that are ascribed are likely to correspond to the programmers intentions when designing the program. Thus, the program can be understood and changed more easily. The structure of the program is more easily accessible then in the source code form.

(4) Especially the fourth point in the above enumeration is extremely important for AOSE because the task of understanding existing software becomes increasingly important in the software industry and is likely to outrange the development of new software. Thus, if it becomes easier to access the original developers idea (that is eventually manifested in the design) it becomes easier to understand the design and this leads to higher cost efficiency in software maintenance.

Through this tour we are also getting introduced into Agent-0 based languages.

Describing Mental State

Here I will clearly define some of the psychological terms we have been using and lay out the semantics behind the coming languages. Some of the terms we need to solidify are decisions, which are partially determined by beliefs, That in turn relate to the state of the world and the capabilities of the agent. The Agent-0 based languages adopt a specific form of decisions that remove some functionality but simplify the problem of language definition. Rather than have an agent decide to perform an action as some function of its beliefs and the state of the world, Shoham introduces the notion of commitments and defines a decision to be

a commitment to oneself to perform an action. The task of the language designer is now simplified in a number of ways:

No new mechanism need be introduced to perform action selection. With commitments, the agent can assume that it magically attains some obligations represented as beliefs that it will perform an action. For example, the Moon-Rover might posses a belief that it will turn left at time t. The actual language specification must have a mechanism for deciding when to adopt these commitments; this is in some sense an action selection mechanism.

- Designing a large system becomes a challenge. With a dynamic action-selection mechanism, decisions can be made according to some (possibly) pre-specified rules. With a commitment system, the rules must be specified when the program is created and must cover a tremendous amount of possible scenarios.

The three key categories of Shoham's formal language are :

Belief

Sentences in the formal language are point-based time statements; they describe the state of something in the world at a particular point in time. For example,
$turn(me, left)^t$

would mean that I will turn / have turned left at time t. Note that this is not only a statement but an action. This introduces the limitation that actions are viewed as "instantaneous".

To express that I currently believe that I will turn left at time t, we would write : B^{*now*}_{me} $turn(me, left)^t$

With this formalism, it is possible to express ideas such as existential belief.

Obligation (Commitment) and Decision

An obligation is introduced as the belief that one agent will create the truth of a statement (for another agent) at a point in time. In other words,

$OBL^{t}_{me,you}$ turn(me, left)$^{t+1}$

means that at time t, I am obligated to you to turn left at time t+1. Note that the statement need not be an action; I could have just as easily entered into the following commitment :

$OBL^{t}_{me,you}$ on_pedestal(you)$^{t+1}$

which means that I am obligated to, somehow, make it true that you are on a pedestal at time t+1.

Again, a decision is simply expressed as an obligation to oneself.

Capability

An agent is said to be capable of a statement if it has the ability to see that that statement hold at the specified time. For example,

CAN^{*now*}_{me} on_pedestal(you)^{t}

means that I am currently capable of seeing to it that you are on a pedestal at time t. Note that this does not assure that, when the time comes, I am still capable of this. An agent must be able to handle the possibility that its capabilities may have changed since it entered into a commitment.

There are four key properties that should hold in any reasonable agent language -- they are required for reasonable performance of the programs written in the language. They are :

1.Internal Consistency

Any agent written in the language should have consistent beliefs and desires. For example, to believe x and believe not(x) simultaneously would violate this constraint. A mechanism is needed for arbitration should these situations arise.

2.Good Faith

An agent should not commit to a statement unless it believes it is capable of the statement.

3.***Introspection***

Agents must be aware of their obligations :

OBL $^{t}_{a,b}$statement -> B^{t}_{a} OBL $^{t}_{a,b}$statement

not(OBL $^{t}_{a,b}$statement) -> B^{t}_{a} not(OBL $^{t}_{a,b}$statement)

4. Persistence of Mental State

AOP requires that beliefs and obligations should persist by default. In other words, what an agent believes at time t it should believe at time t+1 unless some external force has affected its perspective.

Finally, capabilities are assumed to be fairly consistent; this is the result of fairly dependent action-selection and the awkward definition of decisions. If the things that an agent were capable of fluctuated wildly, its entering into commitments would be meaningless. It would not be capable of fulfilling its commitments.

LANGUAGES

We need to have a brief look into the agent oriented languages .lets have looked into the agent-0 language.

4.1 Agent-0

This is one of the languages that can be used for agent oriented programming.Agent-0, like all of its counterparts surveyed here, follows a simple control loop when executing a program.
at each time step

1) Gather incoming messages and update the mental state accordingly
2) Execute commitments (using capabilities)

The cycle of control of an Agent-0 program is shown in Figure 1. Observe that, while it might aid the author to think of an Agent-0 program as a logic program, it is not one. Nothing like proposition matching is involved; simply straight-forward command execution.

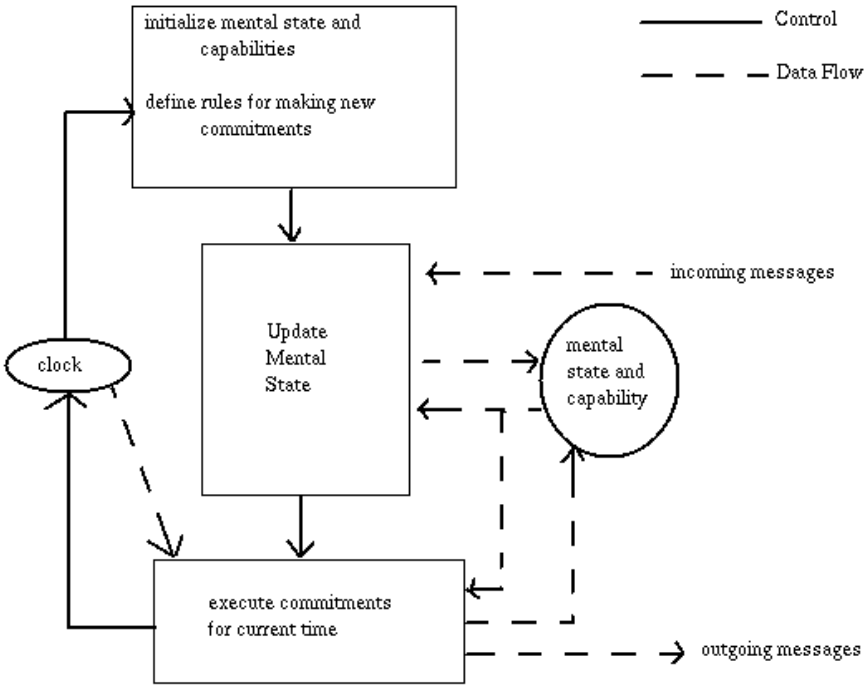

Figure 1 : Control/Data Flow Of The Agent–0 Language Interpreter (Shoham, 93)

A program in Agent-0 is made up of rules for entering commitment, rules for executing capabilities and a set of initial beliefs. In Agent-0, commitments can be made only to execute primitive actions; any activity that would require planning must be committed to as the anticipated elements of its plan. For example, the Rover could not commit to 'go-to-MAP-POINT(x)' unless that was a primitive action. More likely, it would have to commit to the series of 'left-turn', 'right-turn', 'go-forward' actions that would bring it to point x. Note that this is part of the reason for the rigid nature of actions; if for some reason an activity failed, the agent would have no recourse to find a new plan or drop the goal. Beginning with the most elementary details, the syntax of an Agent-0 program is as follows :

Statements

Statements of fact are the foundation of the concept of AOP. Facts in Agent-0, for reasons of implementation complexity, are simple atomic sentences; no conjunction or

disjunction operators are included. Some sample statements : (t (rock west 65)) (which could mean that it is true at time t that there is a rock 65 meters west) and (NOT (t (empty gasTank))) (it is Not true at time t that the gas tank is empty). Of course the interpretation of the sentences is left to the programmer; the intentional point-of-view suggests that these sentences only have meaning from the results they generate.

Unconditional Action Statements

Agents can engage in any of two classes of actions: private and communicative. Private actions are handled by a separate entity (something other than the AOP interpreter) when they are encountered. No mechanism is in place in Agent-0 for private actions to do anything but notify the user that they are to be executed. For example, the turn-right private action of the Rover would do nothing but perhaps print turn-right with the current Agent-0 specification. It is not difficult to imagine a mechanism for sending messages to other (say, Java) programs telling them to execute these primitives. Shoham, however, does not address this deficiency and may have other plans for it.

The syntax of a private-action execution statement is :
(DO t p-action)
where p-action is the private action to execute.
Communicative actions use the speech-act commands to converse with other agents. The four classes of messages in Agent-0 are :
(INFORM t a fact)
(REQUEST t a action)
(UNREQUEST t a action)
(REFRAIN action)
where t specifies the time the message is to take place, a is the agent that receive the message, and action is any action statement. An INFORM action sends the fact to agent a, a REQUEST notifies agent a that the requester would like the action to be realized. An UNREQUEST is the inverse of a REQUEST. A REFRAIN message asks that an action not be committed to by the receiving agent.

Conditional Action Statements

Conditional actions are actions that are initiated only if some keys of the mental state hold. The syntax for specifying a conditional action is :
(IF mntlcond action)
where the action is executed only if the mental condition is true. Mental conditions are queries of the belief of a statement or whether an action is committed to. Conditions, therefore, take the form (B fact) (true if the agent believes the fact) or ((CMT a) action) (true if the agent is committed to agent a to perform the action). Mental conditions may be connected conjunctively or disjunctively using the AND and OR operators. For example :
(AND (B x) (B y) (NOT ((CMT a) z)))
is true if the agent believes x and y but is not committed to agent a to perform z.

Variables

Agent-0 has limited support for variables. Existentially qualified variables (?x) and universally qualified variables (?!x) are supported. The scope of a bound variable is the entire formula in which they are instantiated.

Commitment Rules

As was stated earlier, an Agent-0 program is basically a collection of commitment rules. The agent simply reacts to its environment; bases its actions on what are called message and mental conditions. Mental conditions were discussed earlier. A message condition is matched if the agent just received a message of the specified type. The format of a message condition is (From Type Content) where From is an agent name, Type is a message type (REQUEST, etc), and Content matches the actual message. For example,
(?x REQUEST ?y)
matches every request from every agent.
The syntax for a complete commitment rule is :
(COMMIT msgcond mntlcond (agent action))
which commits the agent to perform action for agent in the future.

Capabilities

Capabilities are stored in a database as (private-action mntlcond) pairs. The mental conditions allow the system to dispose of commitments to execute capabilities that would not behave properly due to a change in the state of the world. What should be done in this case is not immediately clear; Agent-0 simply discards any commitments that cannot be executed without even notifying the agent responsible for requesting the action take place.

So what makes an agent do anything anyway? No agent would ever commit to anything unless asked to by another agent. In actuality, an agent can be written to hold commitments initially and build on those; for example the Rover might be designed to drive around chaotically until it found something interesting (when mental conditions would spawn other commitments). This is not unlike the concept of Event-Driven programming, where behavior is defined in relation to external occurrences.

Comparison of OOP and AOP

An early computational theory that was meant as the foundation of a "natural" way of programming is declarative programming but it has been demonstrated by empirical investigations in cognitive psychology that this claim does not necessarily hold true.

(1) Agent-oriented programming (AOP) can be seen as an extension of object-oriented programming (OOP), OOP on the other hand can be seen as a successor of structured programming. (2)In OOP the main entity is the object. An object is a logical combination of data structures and their corresponding methods (functions). An object can be anything ranging from a concrete entity from the real world to a conceptual entity that only exists in the designers head. Each object within the system is associated with a particular *class* that determines the objects basic properties. Classes can be linked with each other in several ways. Probably the best known relation between two classes is *inheritance* that models a conceptual extension of a common base specification. During their lifetime, objects communicate by sending *messages* to each other. These messages can be used to request services from the receiving object such as to provide internal information or to change the current state.(3) Objects are successfully being used as abstractions for *passive* entities (e.g. a house) in the real-world, and agents are regarded as a possible successor of objects since they can improve the abstractions of *active* entities. (4)Agents are similar to objects, but they also support structures for representing mental components, i.e. beliefs and commitments. (5)In addition, agents support high-level interaction (using agent-communication languages) between agents based on the ``speech act'' theory as opposed to ad-hoc messages frequently used between objects, examples of such languages are ACL and KQML.

(6) The property of autonomy of an agent is a major difference between the two paradigms. An object is passive by nature and is activated on reception of a message (typically reception of an invocation of one of its methods). It does not encapsulate any capacity of choice of actions. The agent, as for it, has a role to play within the system, which justifies that it can be brought to carryout actions for it, even if it is already in direct relation

with other agents. It is always active. This amounts by the henceforth famous sentence: “The objects do it for free; the agents do it because they want to”. Thus the traditional view of an object is the one of a passive entity that acts only on demand. However, this idea of passivity of the object is called into question with the introduction of the active object concept . Therefore the question arises: “is an active object an agent?”. It is true that the concepts get closer, if one disregards pro-activeness and interpretation of the characteristics of autonomy, social ability and reactivity.

(7) Another important difference between AOP and OOP is that objects are controlled from the outside (whitebox control), as opposed to agents that have autonomous behavior which can't be directly controllable from the outside (blackbox control). In other words, agents have the right to say ``no".

In summary, the collection of object-oriented concepts is clear and manageable in size and does not vary greatly in different object-oriented approaches. In the agent-oriented universe, on the other hand, we are faced with the first serious problem as there is no single agreed definition of the entities that are dealt with. Earlier we have mentioned about the various notions of agency.

An object is a logical combination of data structures and their corresponding methods (functions). Moreover agents support high level interaction between agents based on speech act theory as opposed to ad-hoc messages frequently used between objects.

(9) The main point is that OOP has nothing to say about who has access to the methods of an object apart from in terms of setting a global private/public distinction. In AO programming the designer should be forced to think in terms of a messaging interface between agents, such that any accessing of a method subject to security constraints. An AO language would try to pressure developers into providing higher level interfaces, and thus move the locus of control away from the method being accessed, and towards making the entity a self contained entity that checks who is making the request before acting upon it. This

whole approach is supposed to be suited to environments where lots of different people from different organizations have built the components, rather than where one might have lots of different objects built by a group of people that we assume is trying to co-operate.

These concepts correspond to the agent-oriented world by replacing class with role, state variable with belief/knowledge and method with message. Thus a role definition describes the agent's capabilities, the data that is needed to produce the desired results and the requests that trigger a particular service. Besides this fundamental relation, there are many other conceptual similarities between object-orientation and agent-orientation that can be mapped onto each other as summarized in the following table.

	OOP	AOP
Structural Elements		
	abstract class	generic role
	Class	domain specific role
	member variable	Knowledge, belief
	Method	Capability
Relations		
	collaboration (uses)	Negotiation
	composition (has)	holonic agents
	inheritance (is)	role multiplicity
	instantiation	domain specific role + individual knowledge
	polymorphism	service matchmaking

(10) In an object-oriented runtime system, the objects are statically represented by the *object architecture*. This architecture is usually quite simple as it only contains the current state of the object and the relation to the objects class (which determines the operations that can be performed on the object). An object is usually represented as arbitrary collection of data elements with associated functions and the granularity of objects is potentially not limited. However, efficiency issues dictate that not every entity is modeled as an object and so in reality this conceptual benefit is slightly weakened. The *object management system* is responsible for representing the relations such as inheritance between the defined classes and object manipulation such as creating or destroying objects. Furthermore, the object management system is also responsible for dynamic aspects such as method selection of polymorphous objects, exception handling or garbage collection.

In an agent-oriented runtime system, things are distinctly more complicated although similar in their general structure. The basic entities are the agents that are implemented by their agent architecture. However, agent architectures are far more complex then the object architecture, especially because of the dynamic aspects that must be dealt with. Because of the richness of the agent-oriented world, there exists a large number of different agent architectures. Due to the vast number of approaches, it is impossible to identify *the* best or most general architecture. In each iteration, the agent perceives the state of its environment, integrates the perception in its knowledge base that is used to derive the next action which is then executed. This generic cycle is a useful abstraction as it provides a black-box view on the agent architecture and encapsulates specific aspects.

So we can reach the following conclusions about difference between AOP And OOP that The name agent oriented programming was chosen to stress the view of an entity that sends the processes messages-this is the name agent-oriented programming was chosen to stress the view of an agent as an entity that sends and processes messages -- this is the definition of an object in Hewitt's Actors framework. An agent differs from the objects that we know from C++ or Java in a number of significant ways :

- The fields of an agent are restricted. The state of an agent is described in terms of beliefs, capabilities, and decisions. These ideas are built into the syntax of the language.
- Each message is also defined in terms of mental activities. An agent may engage another (or itself) with messaging activities from a restricted class of categories. In Shoham's formalism, the categories of messages are taken from speech-act theory; they are: informing, requesting, offering, accepting, rejecting, competing, and assisting.

I do not believe that it benefits the researcher to think of agents in the framework of OOP. While an agent may be an object in the strictest sense, many ideas considered fundamental to the theory of objects are missing from AOP.

So in shoham’s words

“AOP can be viewed as an specialization of object-oriented programming.” -- Shoham

Can agents solve all software problems?

Since this is a new and rapidly growing filed, there is a danger that researchers become *overly optimistic* regarding the abilities of agent-oriented software engineering. Wooldridge and Jennings discuss the potential pitfalls of agent-oriented software engineering. They have classified pitfalls in five groups: political, conceptual, analysis and design, agent-level, and society-level pitfalls. *Political pitfalls* can occur if the concept of agents is oversold or sought applied as a *the* universal solution. *Conceptual pitfalls* may occur if the developer forgets that agents are software, in fact multithreaded software. *Analysis and design pitfalls* may occur if the developer ignores related technology, e.g. other software engineering methodologies. *Agent-level pitfalls* may occur if the developer tries to use too much or too little artificial intelligence in the agent-system. And finally, *society-level pitfalls* can occur if the developer sees agents everywhere or applies too few agents in the agent-system.

METHODOLOGIES

Several approaches to agent-oriented software engineering have been developed, ranging from structured, informal methodologies, to formal ones, as described in a recent overview and in most of them focusing basically on architectural design. We propose a software development methodology, called Tropos, which will allow us to exploit all the flexibility provided by agent oriented programming. In a nutshell, the three key features of Tropos are the following:

1. The notion of agent and all the related mentalistic notions are used in all phases of software development, from the first phases of early analysis down to the actual implementation.
2. A crucial role is given to the earlier analysis of requirements that precedes prescriptive requirements specification. We consider therefore much earlier phases with respect to standard object oriented methodologies as, for instance, those based on the Unified Modeling Language (UML) , where use case analysis is proposed as an early activity, followed by architectural design.
3. The methodology rests on the idea of building a model of the system-to-be that is incrementally refined and extended from a conceptual level to executable artifacts. This process adopts a transformational approach: a set of transformation operators which allow the engineer to progressively detail the higher level notions introduced in the earlier phases are proposed. It must be noticed that, contrarily to what happens in most other approaches, e.g., UML based methodologies, there is no change of graphical notation from one step to the next (e.g., from use cases to class diagrams).
   The refinement process is performed in a more uniform way.
   In the following section we give an overview of the Tropos methodology,

The Tropos Methodology

The Tropos methodology is intended to support all the analysis and design activities from the very early phases of requirements engineering down to implementation, and organizes them

into five main development phases: early requirement analysis, late requirements analysis, architectural design, detailed design and implementation2.The Tropos modeling language is derived from the Eric Yu's _* paradigm [18] which offers actors, goals, and actor dependencies as primitive concepts for modeling an application during early requirements analysis

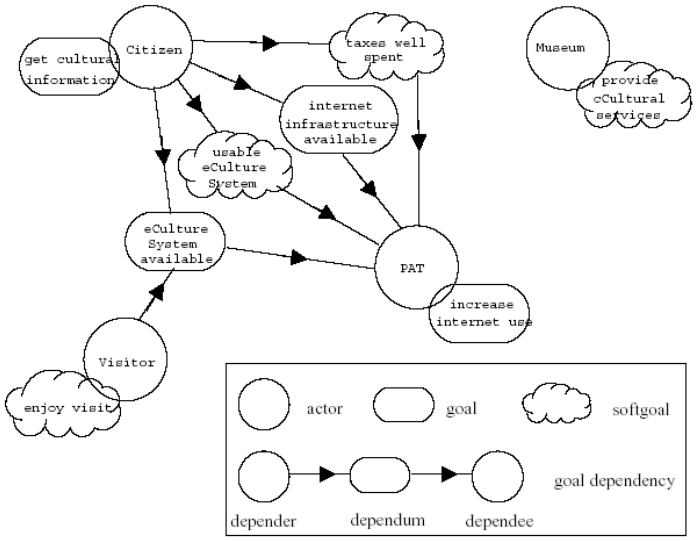

Figure 1. An actor diagram specifying the project stakeholders and their main goal dependencies.

Tropos language is intended to support both an formal model specification and a formal one, allowing for automatic checking of model properties . A set of diagrammatic representation of the model are provided. Each element in the model has its own graphical representation, taken from the* framework. Two main types of diagrams are provided for visualizing the model: the actor diagram3, where the nodes (the actors) are connected through dependencies (labeled arcs), and the goal diagram4, represented as a balloon labeled with a specific actor name and containing goal and plan analysis, conducted from the point of view of the actor. In the rest of this section we briefly describe the five Tropos phases, specifying the main activities of each phase and the process artifacts. The example are extracted from a case-study which refers to the development of a web-based broker of cultural information and services (herein eCulturesystem) for the Province of Trentino, including information obtained from museums, exhibitions, and other cultural organizations. It is the government's intention

that the system be usable by a variety of users, including citizens of Trentino and tourists looking for things to do, or scholars and students looking for material relevant to their studies.

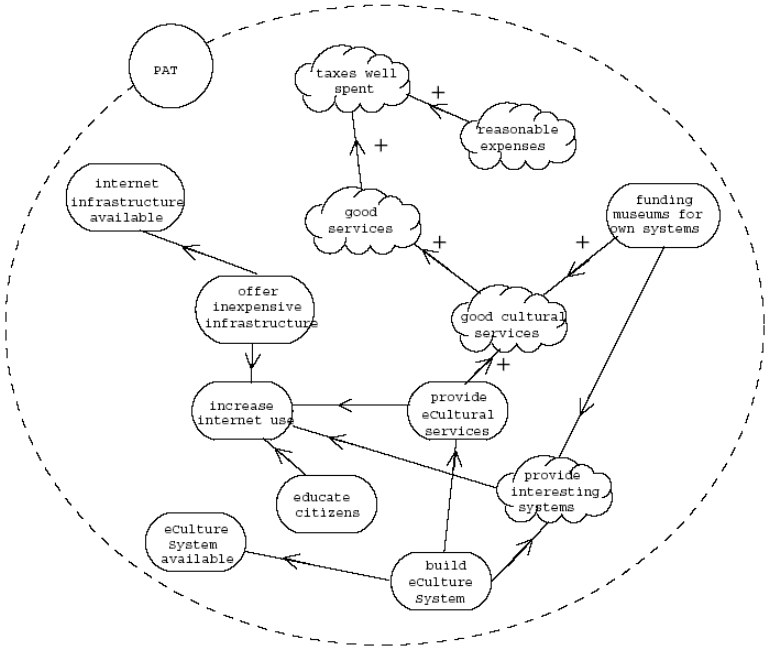

Figure 2. A goal diagram for PAT. The analysis shows goal decomposition and softgoal (positive) contribution.

Early Requirements

The main objective of the early requirement analysis inTropos is the understanding of a problem by studying an existing organizational setting. During this phase, the requirement engineer models the stakeholders as actors and analyzes their intentions, that are modeled as goals which, through a goal-oriented analysis, are then decomposed into finer goals, that eventually can support evaluation of alternatives.

Goal analysis can be concluded by identifyingplans that, if performed by the actor, allow for goal achievement. The analysis can also lead to the identification of further dependencies with other actors. When necessary, we distinguish between hard and soft goals, the latter lacking a clear-cut definition and/or criteria for deciding whether they are satisfied or not. Softgoals are amenable to a more qualitative kind of analysis that, when moving to

later phases concerning the system definition, may lead to the identification of non-functional requirements. The resulting model can be depicted as an actor diagram. Figure 1 shows the actors involved in the eCulture project and their respective goals.

In particular, the actor PAT represents the local government and has been represented with a single relevant goal: increase internet use. The actors Visitor and Museum have associated softgoals, enjoy visit and provide cultural services respectively. The actor Citizen wants to get cultural information and depends on PAT to fulfill the softgoal taxes well spent, a high level goal that motivates more specific PAT's responsibilities, namely to provide an Internet infrastructure, to deliver on the eCulture system and make it usable too. Some of the dependencies in Figure 1 arise from a refinement of the preliminary model obtained by performing goal analysis, as depicted, for instance, in Figure 2.

Late Requirements

The late requirement analysis aims at specifying the system-to-be within its operating environment, along with relevant functions and qualities. The system is represented as an actor which have a number of dependencies with the actors already described during the early requirements phase. These dependencies define all functional and non-functional requirements for the system-tobe. The actor diagram in Figure 3 includes the eCulture System and shows a set of goals that PAT delegates

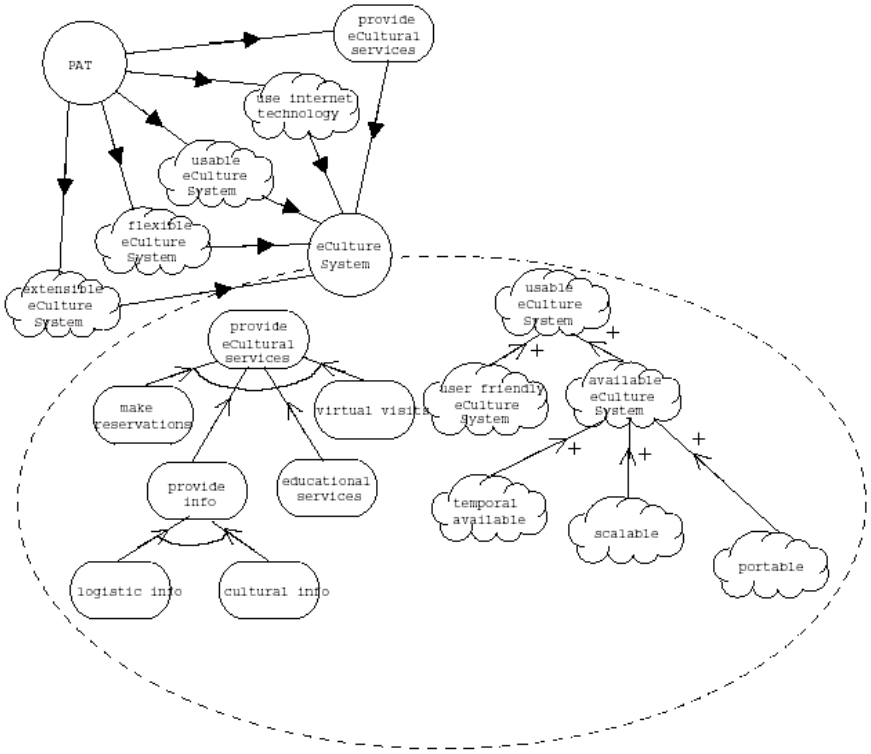

Figure 3. A fragment of the actor diagram including the `PAT` and the `eCulture System` and the goal diagram for the `eCulture System`.

to it through goal dependencies. These goals are then analyzed from the point of view of the eCulture System and are shown in the goal diagram depicted in the lower part of Figure 3. In the example we concentrate on the analysis for the goal provide eCultural services and the softgoal usable eCulture System.The goal provide eCultural services is decomposed (AND decomposition) into four subgoals: make reservations, provide info, educational services and virtual visits. As basic eCultural service, the eCulture System must provide information (provide info), which can be logistic info, and cultural info. Softgoal contributions are than identified. So for instance, the softgoal usable eCulture System has two positive (+) contributions from softgoals user friendly eCulture System and available eCulture System. The former contributes positively because a system must be user friendly to be usable, whereas the latter contributes positively because it makes the system portable, scalable, and available over time (temporal available).

Architectural Design

The main objective of the architectural design phase is the definition of the system's global architecture in terms of subsystems (actors), interconnected through data and control flows (dependencies). Basically, this phase consists of three steps: refining the system actor diagram introducing subactors upon analysis of functional and non functional requirements and taking into account design patterns (step 1); capturing actor capabilities from the analysis of the tasksthat actors and sub-actors will carry on in order to fulfill functional requirements (step 2); defining a set of agent types (components) and in assigning to each component one or more different capabilities (step 3). A portion of the architectural design model of the eCulture project, resulting from the first step, is represented by the actor diagram in

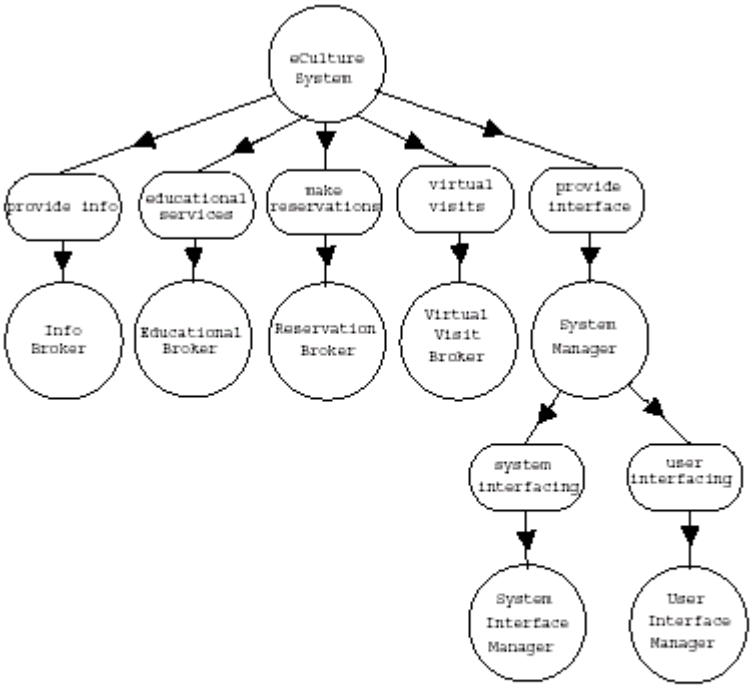

Figure 4. Actor diagram of the architecture of the `eCulture System` (step 1)

Detailed Design

The detailed design phase aims at specifying the agent (component) capabilities and interactions. At this point, usually, the implementation platform has already been chosen and this can be taken into account in order to perform a detailed design that will map directly to the code5. So, for instance, choosing a BDI (Belief Desire Intention) multiagent platform will require the specification of agent capabilities in terms of external and internal events that trigger plans, and the beliefs involved in agent reasoning. These properties are specified through a set of diagrams. A subset of the AUMLdiagrams proposed in are used: the activity diagrams (capability diagram) to model a capability (or a set of correlated capabilities) from the point of view of a specific actor; activity diagrams (plan diagram) to specify each plan node of a capability diagram; and sequence diagrams (agent interaction diagram) to model agents interaction in terms of communication acts.

Implementation

The implementation activity follows step by step the detailed design specification, according to the established mapping between the implementation platform constructs and the detailed design notions. In our case-study, the JACK Intelligent Agents platform has been chosen for implementation. JACK is a BDI agent-oriented development environment built on top and fully integrated with Java, where agents are autonomous software components that have explicit goals (desires) to achieve or events to handle. Agents are programmed with a set of plans in order to make them capable of achieving goals.

Agents in the real-world

The agent-oriented approach is increasingly being applied in industrial applications, but it is far from as widespread as the object-oriented approach. This section describes where and how agents have been applied with success in the manufacturing industry.

Parunak defines agenthood, a taxonomy and a maturity metric in an industrial context. His purpose is to improve the understanding and utilization of agent-oriented software engineering in industry. *Agenthood*, i.e. agent-oriented programming, is explained as an iterative improvement of the industry-strength methodology of object-oriented programming.

The *taxonomy* classifies agent systems as belonging to one of the following *environments*: digital (i.e. software and digital hardware), social (involving human users) or electromechanical (non-digital hardware, e.g. a motor). Thereafter the taxonomy classifies agents according to the *interface* they support. Interface types are similar to the environments: digital (e.g. communication protocols), social (e.g. user interfaces) and electromechanical (e.g. motor control interfaces).

Few business users, as opposed to researchers, are early-adapters of new and immature technology, as a result of this a *maturity metric* of agent-based systems is developed to be able to measure the level of agent technology and systems. The maturity metric has six degrees ranging from modeled applications to products. *Modeled applications*, the least mature, are theoretical applications in the form of architectural descriptions or analyses. The metric continues with *emulated applications* that are relatively immature due to the fact that they are simulations in a lab environment. *Prototype applications* represent the next maturity degree, they run in a non-commercial environment but on real hardware. *Pilot applications* are relatively mature applications, however they are not expected to be completely bug-free, and after a certain period they usually become more mature and become *production applications*. A production application is being applied in several businesses, but they require support for installation and maintenance. The most mature applications are *products*, they are

usually shrink-wrapped and sold over desk, and they can usually be installed and maintained by the non-expert user.

Agents in the industry - where and how?

Parunak presents a review of industrial agent applications. Application areas considered are: manufacturing scheduling, control, collaboration and agent simulation. Thereafter tools, methodologies, insights and problems for development of agent systems are presented and discussed.

Manufacturing scheduling is the ordering and timing of processes in a production facility. The purpose is to optimize the production by maximizing the number of units produced per time slot and keep good quality of the product, and minimize the resource requirements per unit and the risk of failures. Processes and machinery has to be *controlled* in order to operate as scheduled. The control can range from simple regulation of the power level for a piece of machinery to advanced real-time cybernetic control of processes. For many industries, human *collaboration* is needed to solve complex problems, e.g. in a design process engineers and designers have to collaborate in order to guarantee that products are pleasent to look in addition to being safe. In industries such as electronics production, there are tremendous setup costs for production facilities, consequently there is a need for cost-efficient *simulation* of the manufacturing processes.

Typical applications of agent programming

mobile computing

The following distinguishing characteristics of an agent are suitable for a wide area mobile networking environment.

Mobility: Mobile computing exhibits an inherently transient nature i.e. host locations constantly change. Transactions span many nodes and are short lived. The ability of agents to carry code and data across network nodes underlines their suitability to a transient environment.

Concurrent problem solving: Agents provide a clear, natural paradigm for performing tasks, which may have several concurrent aspects. Autonomous security model: Each agent is assigned a privilege level. The agent is only granted as much privilege as it needs. Agent may also transfer ,user or system privileges.

Proxy Handling: Agents can behave simultaneously as server proxies or client proxies at any node. Server proxies emulate the functionality or presence of a server while client proxies emulate the same for client.

Communication traffic routing: Determining the optimum network path for network communication has always as been near impossible task as no model of the existing network traffic can take into account the variable nature of the data traffic. An autonomous agent can be programmed to take into account the variable nature of such traffic and can dynamically optimize the network path of such large networks.

Information scouts: Intelligent agents can be used for collecting data. The information scouting agents go through the network, communicating with the user and other agents for achieving their goals. example. Segue represents pages in browsing history using a series of ‘skeins’ which represent changes in interest over time.

Butterfly: Samples thousands of real time conversational group & recommends those that are of interest.

Air traffic control : This is a very demanding task, getting all circling planes to land safely. An agent can be designed to do the task efficiently, taking care of the weather conditions and other factors. Other examples of agent are KidSim agent, IBM agent, Hayes- Roth agent etc.

CONCLUSION

We need to program an agent which meets all the required abilities can be quiet a daunting task as qualities such as desire and belief have to be incorporated within the agent and a framework for sharing with other agents has to be designed thereby requiring elements of AAI to be incorporated into the agent.

At times, adapted neural network processes have been used to build an agent. The agent oriented programming paradigm holds much promise for the future and once used, will pave the way for more self aware, efficient agents that could be deployed in almost every conceivable situation that demands complex reactions and actions.