Lesson Proper for Week 10

INTRODUCTION

Open architectures are essential because they allow for the addition of new components and the fulfillment of ever-changing demands. More and more software must be able to run across a variety of platforms without the need for recompilation and with only the barest of assumptions made about the underlying operating system and its intended audience. Strong, self-reliant, and proactive are all prerequisites. That is why Agent-Oriented Programming was developed.

The goal of Agent Oriented (AO) Technology is to develop systems that apply to the real world and capable of observing and reacting to environmental changes. These systems must be capable of acting rationally and autonomously to accomplish their assigned duties. AO technology is a methodology for developing large-scale real-time distributed applications. This technology is predicated on the assumption that a computer system should demonstrate logical goal-directed behavior comparable to that of a human being. The AO technology accomplishes this by developing beings called agents that are intentional, reactive, communicative, and occasionally team oriented.

Various programming methods exist. This is the successor of Structured Programming. Agent-oriented programming is an evolution of object-oriented programming. This implies that both notions are close to the programming language and implementation level. Shoham coined the term "Agent-Oriented Programming." So AOP is a new programming paradigm that supports societal computation. Agents interact in AOP to pursue individual goals. Agents can be autonomous, making decisions without a user's input, or controllable, acting as a middleman between two agents. AOP is an abstract programming language. Agent-Oriented Software Engineering is a new paradigm in software engineering research. But to become a new software paradigm, strong and easy-to-use processes and tools must be established. Shoham coined the term AOP.

ESSENTIALS OF AN AGENT-ORIENTED PROGRAMMING

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

1. A formal language with precise syntax for describing a person's mental state. This would most likely include structure for expressing beliefs, conveying messages, and so forth.

2. A programming language for creating agents. This language's semantics should be closely tied to formal language.

3. A strategy for transforming non-agent programs into agents. An agent will be able to speak with a non-agent using this type of technology.

AGENT

Before we go into the programming side of AOP, let's define the agent notion.

Agent is a software process that follows OAA (Open Agent Architecture) conventions. An agent is a model abstraction that helps model a system. An agent-based paradigm can model a complicated system as a series of entities. It does so by registering its services, speaking the Inter-agent Communication language (IACL), and sharing features common to all OAA Agents, such as installing triggers, managing data, etc. Shoham defines an agent as "an entity whose state includes mental components such as beliefs, capacities, choices, and commitments". This is at best cryptic and at worst worthless. Agent-Oriented Programming has no agreed-upon concept of entities.

"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, an agent is a computer system having the following properties:

§ **Autonomy:** The ability to operate without the 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. The environment is usually thought to be physical, unpredictable, and containing other agents

§ **Pro-activeness:** The ability to exhibit goal-directed behavior by taking the initiative instead of just acting in response. An agent is proactive 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 waiting for orders stating to it what it must do.

Some other attributes of the agency are:

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

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

ª **Benevolence:** The assumption of not having conflicting goals.

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

So an agent is a piece of autonomous software that is intelligent. Can you define the words intelligent and agent? An agent is intelligent when it chooses actions based on knowledge. Agents are permitted to act for or in place of others.

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 be used by a software engineer.

So the agent needs to have

¥ **Belief:** This represents the agent's 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 the 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 agent's desires and is generally implemented as plans. Plans 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 must not only understand what needs to be accomplished but also be capable of taking appropriate action to guarantee that the intended state of the world is realized. Therefore, an agent system requires not only abductive reasoning but also a suitable idea of the agent.

AGENT CLASSIFICATION

So far, we've defined an agent. Now we can talk about agents. Classification is based on the existence or absence of the above attributes. The concept of agency is ambiguous. Agent theories rely on these two concepts. Agents have free choice (autonomy), can connect with others (social capacity), respond to stimuli (reactivity), and take initiative (initiative) (pro-activity). Agents need simply a white box to be weak. Agents are defined purely by observable attributes. Agents can move around (mobility), are truthful (veracity), do what they are directed to do (benevolence), and will perform optimally to attain goals (rationality). i.e. mental ideas are used to model a strong sense of agency. These mental ideas should be implicitly represented in agent implementation. Agents are forced inside a white box. McCarthy sees mental attributes as a tool for comprehending and communicating amongst humans, a way of expressing existing knowledge about a program or its current state.

Shall we go into detail about the idea of mental concepts

"All the ... reasons for ascribing beliefs 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 the organization to systems."

McCarthy provides the example of a program in source code form to demonstrate why this point of view is logical. By simulating the given code, it is feasible to entirely define the behavior of the program, i.e. no mental categories are required 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 program's state at a particular point in time is usually not directly observable.

2. Therefore, the observable information is better expressed in mental categories.

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

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

5. 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 developer's 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 to Agent-0-based languages.

Describing Mental State

In this section, I'll describe some of the psychological terms we've been using and outline the semantics of the emergent languages. Several of the terms we need to describe our judgments are influenced in part by the agent's assumptions about the condition of the environment and its capabilities. Agent-based languages employ a novel decision-making approach that eliminates some functionality while simplifying the language creation process. Rather than having an agent pick an action based on its beliefs and the state of the universe, Shoham introduces the concept of commitments and defines decision-making as an obligation to oneself to do an action. The language designer's job has been simplified in a number of ways:

There is no need to introduce a new method to accomplish action selection. With commitments, the agent can assume that it miraculously acquires certain duties in the form of ideas about how it will act. For instance, the Moon-Rover may have the conviction that it will turn left at time t. The language specification itself must include a system for determining when to embrace these commitments; this is a form of action selection.

Developing a huge system gets difficult. Decisions can be made using a dynamic action-selection process by some (potentially) pre-specified rules. When a commitment system is used, the rules must be set throughout the program's development and must cover an enormous number of possible events.

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 defined as the conviction that one agent will generate the truth of a statement (for the benefit of another agent) at a particular moment in time. Alternatively, $OBL^{t}_{me', you}\ turn(me, left)^{t+1}$

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 can see that that statement holds 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, capacities are expected to be reasonably constant; this is due to the relatively dependent nature of action selection and the problematic characterization of decisions. If an agent's capabilities fluctuated widely, its obligations would be useless. It would be unable of meeting its obligations.

LANGUAGES

We need to take a quick look at agent-oriented programming languages. Let us examine the agent-0 programming language.

4.1 Agent-0

This is a language that supports agent-oriented programming. When Agent-0 executes a program, it, like all of its counterparts seen here, uses a simple control loop.

1. Gather incoming messages and update the mental state accordingly

2. Execute commitments (using capabilities)

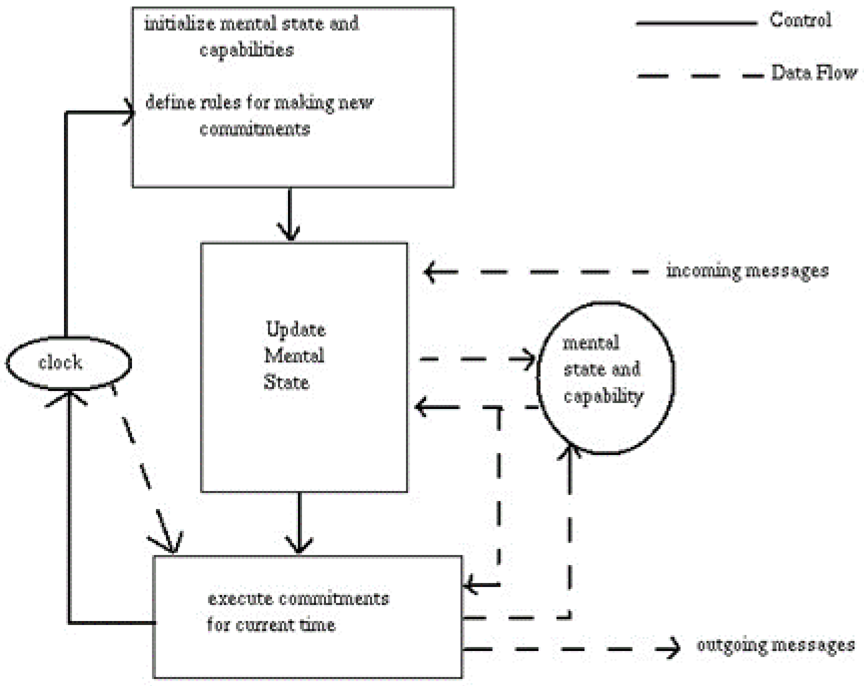
initialize mental state and capabilities
define rules for making new commitments
Control
Data Flow
incoming messages
Update Mental State
clock
mental state and capability
execute commitments for current time
outgoing messages

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

The control cycle of an Agent-0 program is depicted in Figure 1. While it may be helpful for the author to think of an Agent-0 program as a logic program, this is not the case. There is no such thing as proposition matching; this is merely command execution.

In Agent-0, a program is composed of rules for establishing commitment, rules for executing capabilities, and a collection of initial beliefs. Agent-0 permits commitments only for the execution of primitive activities; any activity requiring planning must be committed to as anticipated pieces of its plan. For example, unless it was a primitive operation, the Rover could not commit to 'go-to-MAP-POINT(x)'. Rather than that, it would have to commit to the succession of 'left-turn', 'right-turn', and 'advance' operations necessary to reach point x. This is one of the reasons for the rigid nature of activities; if an activity fails, the agent is powerless to devise a new plan or abandon the aim. To begin, 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 do either private or communicative acts. When private actions are encountered, they are processed by a separate entity (not the AOP interpreter). In Agent-0, there is no mechanism in place for private actions to accomplish anything other than warn the user that they are about to be executed. For instance, the Rover's turn-right private action would do nothing more than print turn right under the present Agent-0 standard. It is not straightforward to envision a system for informing other programs (say, Java) to execute these primitives. Shoham, on the other hand, does not address this shortcoming and may have alternative ambitions.

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 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 receives 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.

A UNREQUEST is the inverse of a REQUEST. A REFRAIN message asks that 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

In a database, capabilities are kept as (private-action mntlcond) pairs. The mental conditions enable the system to release commitments to execute capabilities that would no longer function properly as a result of a change in the world's state. What should be done in this scenario is unclear; Agent-0 simply discards any obligations that cannot be fulfilled without telling the agent that requested the action take place.

So what motivates an agent to act in the first place? No agent would ever commit to anything unless another agent specifically requested it. In reality, an agent can be programmed to make initial commitments and then expand on them; for example, the Rover could be programmed to drive around randomly until it discovered something interesting (when mental conditions would spawn other commitments). This is similar to the event-driven programming paradigm, in which behavior is described in terms of external occurrences.

Comparison of OOP and AOP

Declarative programming is an early computational theory that was intended to serve as the foundation for a "natural" style of programming, however actual research in cognitive psychology has proven that this claim does not always hold true.

1. Agent-oriented programming (AOP) can be thought of as a subset of object-oriented programming (OOP), whereas OOP can be thought of as the successor of structured programming.

2. The object is the central entity in OOP. An object is a logical representation of a collection of data structures and their associated procedures (functions). An object can be anything from a physical being in the physical world to a conceptual entity that lives just in the designer's mind. Each object in the system is assigned a unique class that defines the entity's fundamental attributes. Classes can be connected in a variety of ways. The most well-known relationship between two classes is probably inheritance, which represents a conceptual expansion of a shared base specification. Throughout their existence, items connect via messages. These messages may be used to request services from the receiving object, such as the provision of internal data or the modification of the current state.

3. In the actual world, objects are successfully employed as abstractions for passive entities (e.g., a house), while agents are viewed as possible successors to objects due to their ability to improve the abstractions of active things.

4. Agents, like objects, provide structures for representing mental components, such as beliefs and commitments.

5. Additionally, agents enable high-level interaction between agents (via agent-communication languages) based on the "speech act" theory, as opposed to the ad-hoc messages usually employed between objects; examples of such languages include ACL and KQML.

6. The attribute of agent autonomy is a significant distinction between the two paradigms. By definition, an object is passive and is activated upon receipt of a message (typically reception of an invocation of one of its methods). It is devoid of any capacity for action selection. As for the agent, it has a purpose inside the system, which justifies it's being brought in to perform actions on its behalf, even if it is already in direct contact with other agents. It is perpetually active. This is summed up in the now-famous sentence: "Objects do things for free; agents do things because they want to." Thus, the typical understanding of an object is that of a passive thing that operates only in response to external stimuli. However, with the emergence of the concept of the active object, this notion of the object's passivity is put into doubt. As a result, the issue "is an active item an agent?" emerges. The notions indeed become more similar when proactivity and interpretation of autonomy, social ability, and reactivity are disregarded.

7. Another significant distinction between AOP and OOP is that objects can be controlled from the outside (whitebox control), in contrast to agents with autonomous behavior that cannot be controlled directly from the outside (blackbox control). Agents, in other words, have the right to say "no."

In summary, in the agent-oriented universe, there is no single agreed definition of entities that are dealt with. An object is a logical combination of data structures and their corresponding methods (functions). Agents support high-level interaction between agents based on speech act theory as opposed to ad-hoc messages.

8. The essential point is that OOP has no bearing on who has access to an object's methods, except to establish a global private/public distinction. In AO programming, the designer should be required to conceive in terms of an agent-to-agent messaging interface, with any method access subject to security limitations. An AO language would attempt to compel developers to provide higher-level interfaces, so shifting the locus of control away from the method being accessed and toward a self-contained entity that verifies who is requesting acting on it. This entire technique is intended to be more suitable to situations in which a variety of different people from diverse organizations have produced the components, rather than to environments in which a variety of different items have been built by a group of people attempting to collaborate.

By substituting role for class, state variable for belief/knowledge, and method for message, these notions relate to the agent-oriented world. Thus, a role specification outlines the agent's skills, the data required to generate the intended results, and the requests that initiate the execution of a certain service. Apart from this fundamental connection, there are numerous more conceptual parallels between object- and agent-orientation that can be mapped onto one another, as shown 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

9. In an object-oriented runtime system, the *object architecture* serves as a static representation of the objects. This architecture is typically extremely minimal, containing only the object's present state and its relationship to the object's class (which determines the operations that can be performed on the object). Typically, an object is represented as an unrestricted collection of data items with associated functions, and the granularity of objects is unrestricted. However, due to efficiency concerns, not every entity is treated as an object, and therefore this conceptual benefit is slightly lessened in practice. The *object management system* is in charge of representing relationships between defined classes and manipulating objects such as generating and removing them. Additionally, the object management system is responsible for dynamic factors such as polymorphous object method selection, exception handling, and garbage collection.

Typical applications of agent programming

mobile computing

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

ª **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 for a transient environment.

ª **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. The agent may also transfer, user or system privileges.

ª **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 the client.

ª **Communication traffic routing:** Determining the optimum network path for network communication has always been a 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.

ª **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.

ª **Butterfly:** Samples thousands of real-time conversational groups & recommend those that are of interest.

ª **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 an agent are KidSim agent, IBM agent, Hayes- Roth agent, etc.

◄ Preliminary Activity for Week 10

Jump to...

Analysis, Application, and Exploration for Week 10 ►

Navigation

Home
- Dashboard
  - Site pages
  - My courses
    - Capstone Project 2
    - Network Defense and Remote Access Configuration
    - OJT/Practicum 2
    - Seminars and Tours
      - Participants
      - General
      - 10 [Enter Module Title Here]
        - Preliminary Activity for Week 10
        - **Lesson Proper for Week 10**
        - Analysis, Application, and Exploration for Week 10
        - Generalization for Week 10
        - Evaluation for Week 10
        - Assignment for Week 10
  - Courses

Fair Warning

Activities

- Assignments
- Forums
- Quizzes
- Resources