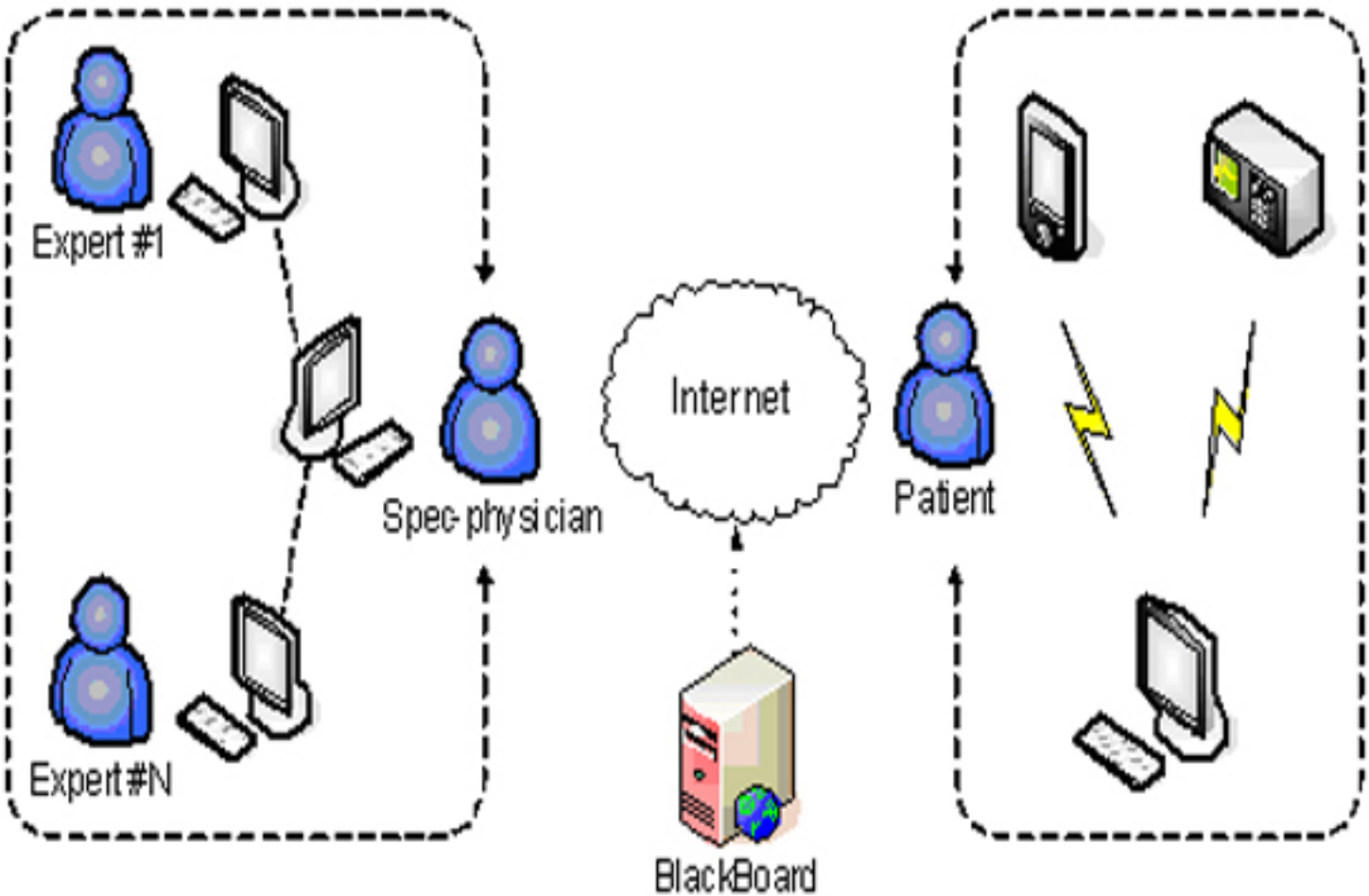
Multi-Agent Systems



Julian Curran

First Edition, 2012

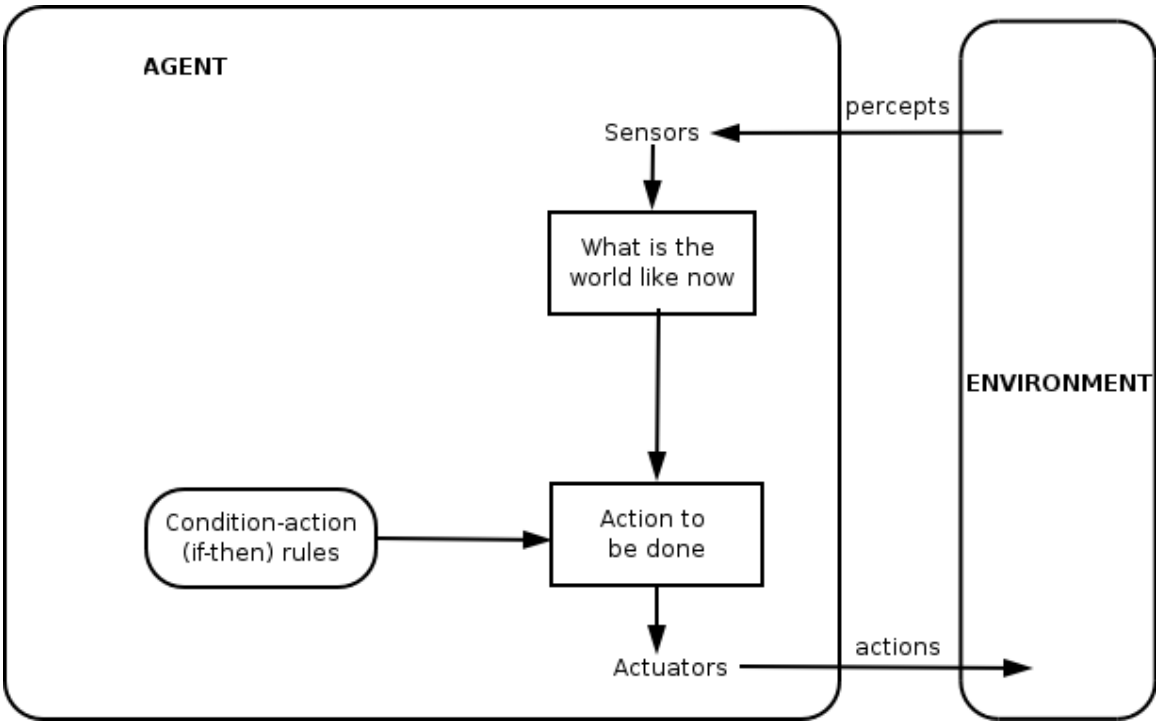
ISBN 978-81-323-3087-5

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

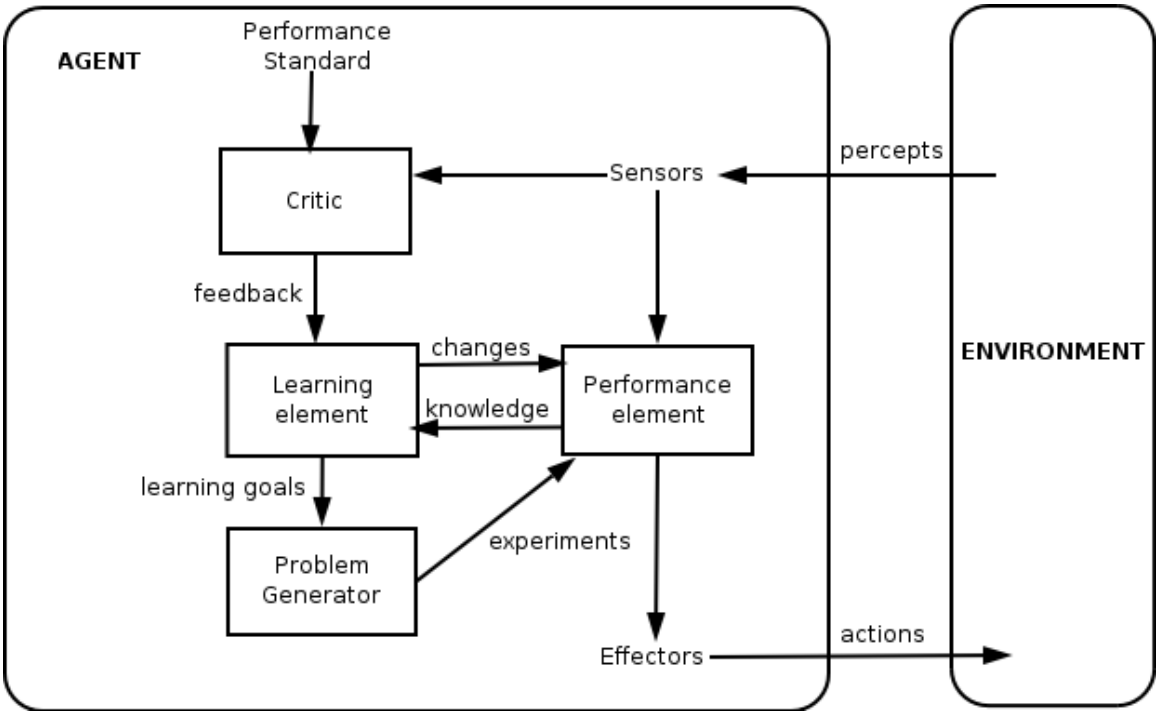
Table of Contents

Chapter 1

Multi-Agent System



Simple reflex agent



Learning agent

A **multi-agent system** (**MAS**) is a system composed of multiple interacting intelligent agents. Multi-agent systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Intelligence may include some methodic, functional, procedural or algorithmic search, find and processing approach.

Topics where multi-agent systems research may deliver an appropriate approach include online trading, disaster response, and modelling social structures.

Overview

The agents in a multi-agent system have several important characteristics:

- **Autonomy**: the agents are at least partially autonomous
- **Local views**: no agent has a full global view of the system, or the system is too complex for an agent to make practical use of such knowledge
- **Decentralization**: there is no designated controlling agent (or the system is effectively reduced to a monolithic system)

Typically multi-agent systems research refers to software agents. However, the agents in a multi-agent system could equally well be robots, humans or human teams. A multi-agent system may contain combined human-agent teams.

Self organisation and self steering

Multi-agent systems can manifest self-organization as well as self-steering and other control paradigms and related complex behaviors even when the individual strategies of all their agents are simple.

When agents can share knowledge using any agreed language, within the constraints of the system's communication protocol, the approach may lead to a common improvement. Example languages are Knowledge Query Manipulation Language (KQML) or FIPA's Agent Communication Language (ACL).

Multi-agent system basics

Multiple agent systems paradigms

Many MAS systems are implemented in computer simulations, stepping the system through discrete "time steps". The MAS components communicate typically using a weighted request matrix, e.g.

```
 Speed-VERY_IMPORTANT: min=45 mph,
 Path length-MEDIUM_IMPORTANCE: max=60 expectedMax=40,
 Max-Weight-UNIMPORTANT
 Contract Priority-REGULAR
```

and a weighted response matrix, e.g.

```
 Speed-min:50 but only if weather sunny,
 Path length:25 for sunny / 46 for rainy
 Contract Priority-REGULAR
 note - ambulance will override this priority and you'll have to wait
```

A challenge-response-contract scheme is common in MAS systems, where

```
 First a "Who can?" question is distributed.
 Only the relevant components respond: "I can, at this price".
 Finally, a contract is set up, usually in several more short
communication steps between sides,
```

also considering other components, evolving "contracts", and the restriction sets of the component algorithms.

Another paradigm commonly used with MAS systems is the pheromone, where components "leave" information for other components "next in line" or "in the vicinity". These "pheromones" may "evaporate" with time, that is their values may decrease (or increase) with time.

Properties

MAS systems, also referred to as "self-organized systems", tend to find the best solution for their problems "without intervention". There is high similarity here to physical phenomena, such as energy minimizing, where physical objects tend to reach the lowest energy possible, within the physical constrained world. For example: many of the cars entering a metropolis in the morning, will be available for leaving that same metropolis in the evening.

The main feature which is achieved when developing multi-agent systems, if they work, is flexibility, since a multi-agent system can be added to, modified and reconstructed, without the need for detailed rewriting of the application. These systems also tend to be rapidly self-recovering and failure proof, usually due to the heavy redundancy of components and the self managed features, referred to, above.

The study of multi-agent systems

The study of multi-agent systems is "concerned with the development and analysis of sophisticated AI problem-solving and control architectures for both single-agent and multiple-agent systems." Topics of research in MAS include:

- agent-oriented software engineering
- beliefs, desires, and intentions (BDI)
- cooperation and coordination
- organization
- communication
- negotiation
- distributed problem solving
- multi-agent learning
- scientific communities
- dependability and fault-tolerance

Frameworks

While ad hoc multi-agent systems are often created from scratch by researchers and developers, some frameworks have arisen that implement common standards (such as the FIPA agent system platforms and communication languages). These frameworks save developers time and also aid in the standardization of MAS development.

Applications in the real world

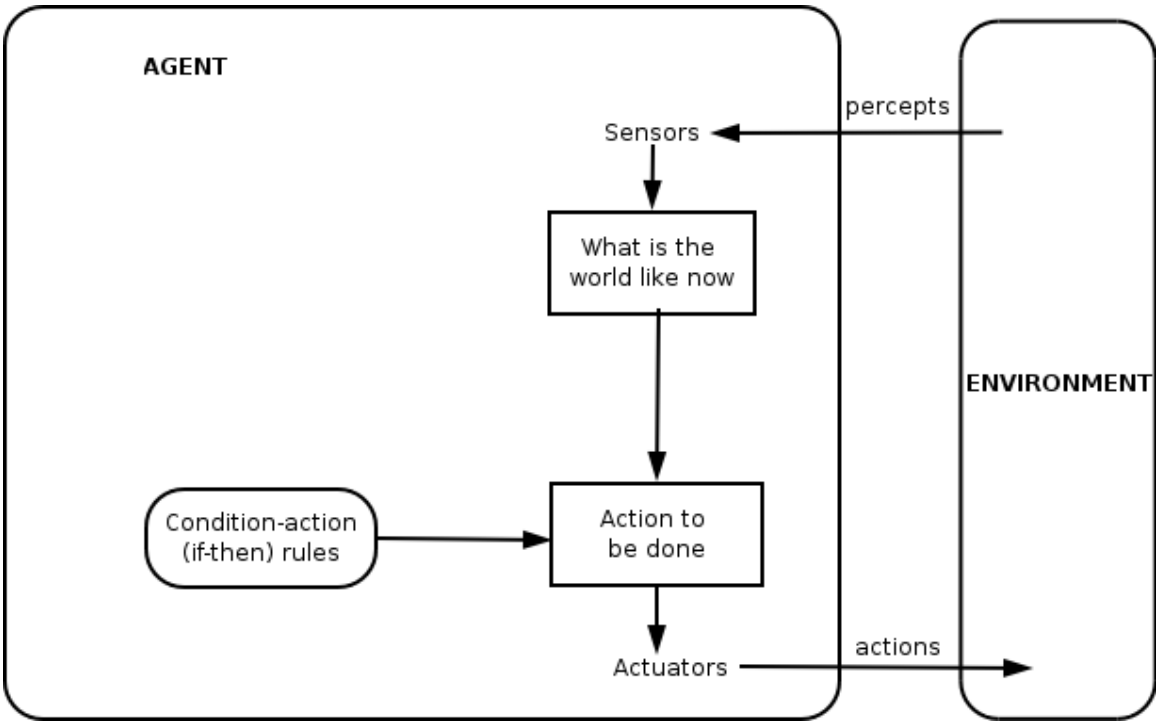
Multi-agent systems are applied in the real world to graphical applications such as computer games. Agent systems have been used in films. They are also used for coordinated defence systems. Other applications include transportation, logistics, graphics, GIS as well as in many other fields. It is widely being advocated for use in

networking and mobile technologies, to achieve automatic and dynamic load balancing, high scalability, and self-healing networks.

Chapter 2

Intelligent Agent

In artificial intelligence, an **intelligent agent** (**IA**) is an autonomous entity which observes and acts upon an environment (i.e. it is an agent) and directs its activity towards achieving goals (i.e. it is rational). Intelligent agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex: a reflex machine such as a thermostat is an intelligent agent, as is a human being, as is a community of human beings working together towards a goal.



Simple reflex agent

Intelligent agents are often described schematically as an abstract functional system similar to a computer program. For this reason, intelligent agents are sometimes called **abstract intelligent agent**s (AIA) to distinguish them from their real world implementations as computer systems, biological systems, or organizations. Some definitions of intelligent agents emphasize their autonomy, and so prefer the term **autonomous intelligent agent**s. Still others (notably Russell & Norvig (2003)) considered goal-directed behavior as the essence of intelligence and so prefer a term borrowed from economics, "rational agent".

Intelligent agents in artificial intelligence are closely related to agents in economics, and versions of the intelligent agent paradigm are studied in cognitive science, ethics, the philosophy of practical reason, as well as in many interdisciplinary socio-cognitive modeling and computer social simulations.

Intelligent agents are also closely related to software agents (an autonomous software program that carries out tasks on behalf of users). In computer science, the term *intelligent agent* may be used to refer to a software agent that has some intelligence, regardless if it is not a rational agent by Russell and Norvig's definition. For example, autonomous programs used for operator assistance or data mining (sometimes referred to as *bots*) are also called "intelligent agents".

A variety of definitions

Intelligent agents have been defined many different ways. According to Nikola Kasabov IA systems should exhibit the following characteristics:

- accommodate new problem solving rules incrementally
- adapt online and in real time
- be able to analyze itself in terms of behavior, error and success.
- learn and improve through interaction with the environment (embodiment)
- learn quickly from large amounts of data
- have memory-based exemplar storage and retrieval capacities
- have parameters to represent short and long term memory, age, forgetting, etc.

Structure of agents

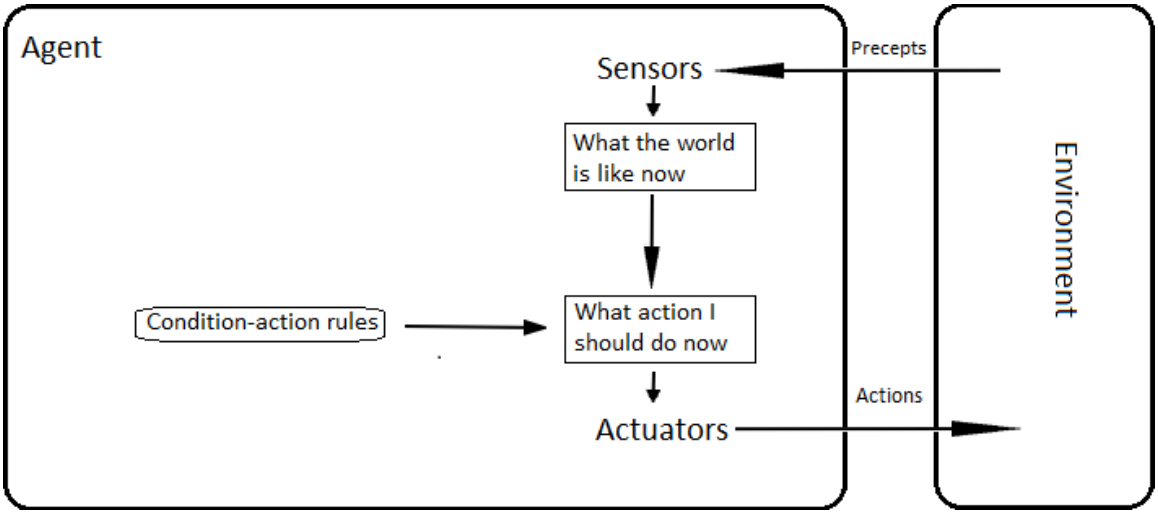
A simple agent program can be defined mathematically as an agent function which maps every possible percepts sequence to a possible action the agent can perform or to a coefficient, feedback element, function or constant that affects eventual actions:

$$f : P^* \rightarrow A$$

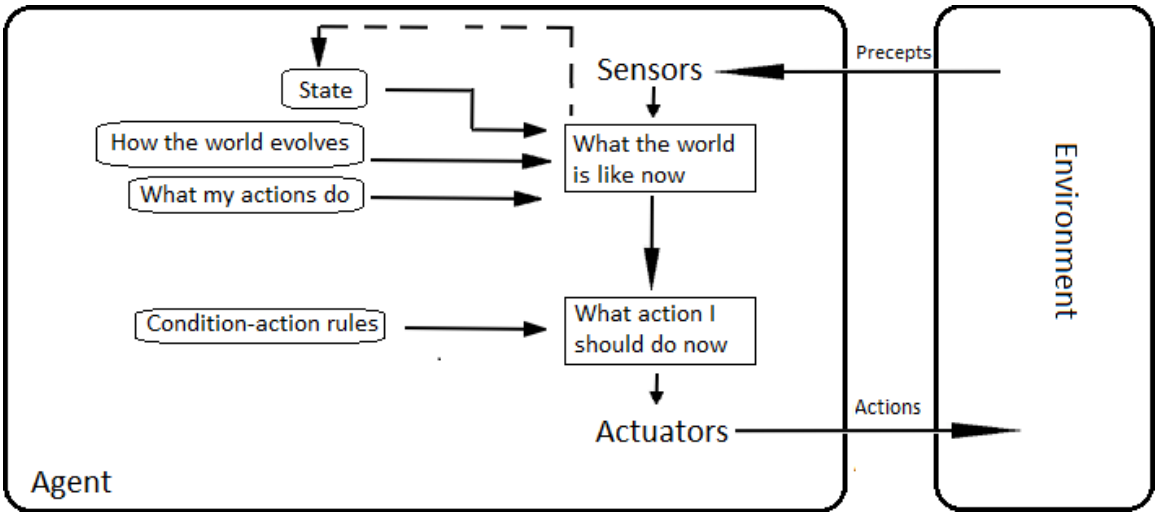
The program agent, instead, maps every possible percept to an action.

We use the term percept to refer to the agent's perceptional inputs at any given instant. In the following figures an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

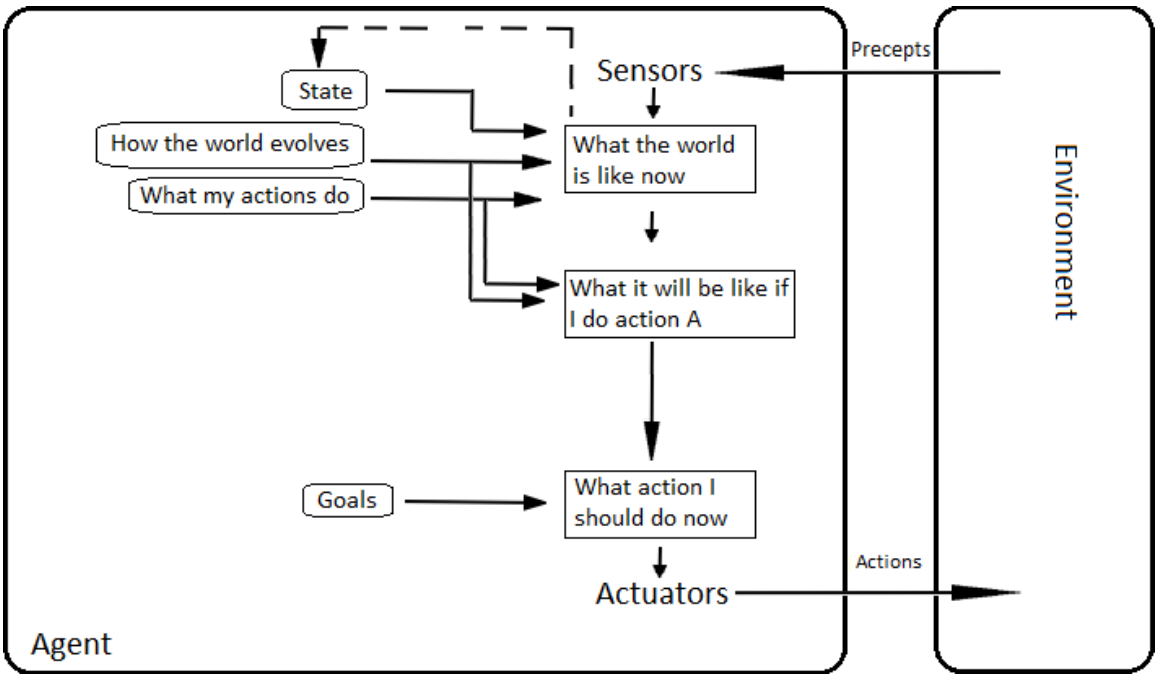
Classes of intelligent agents



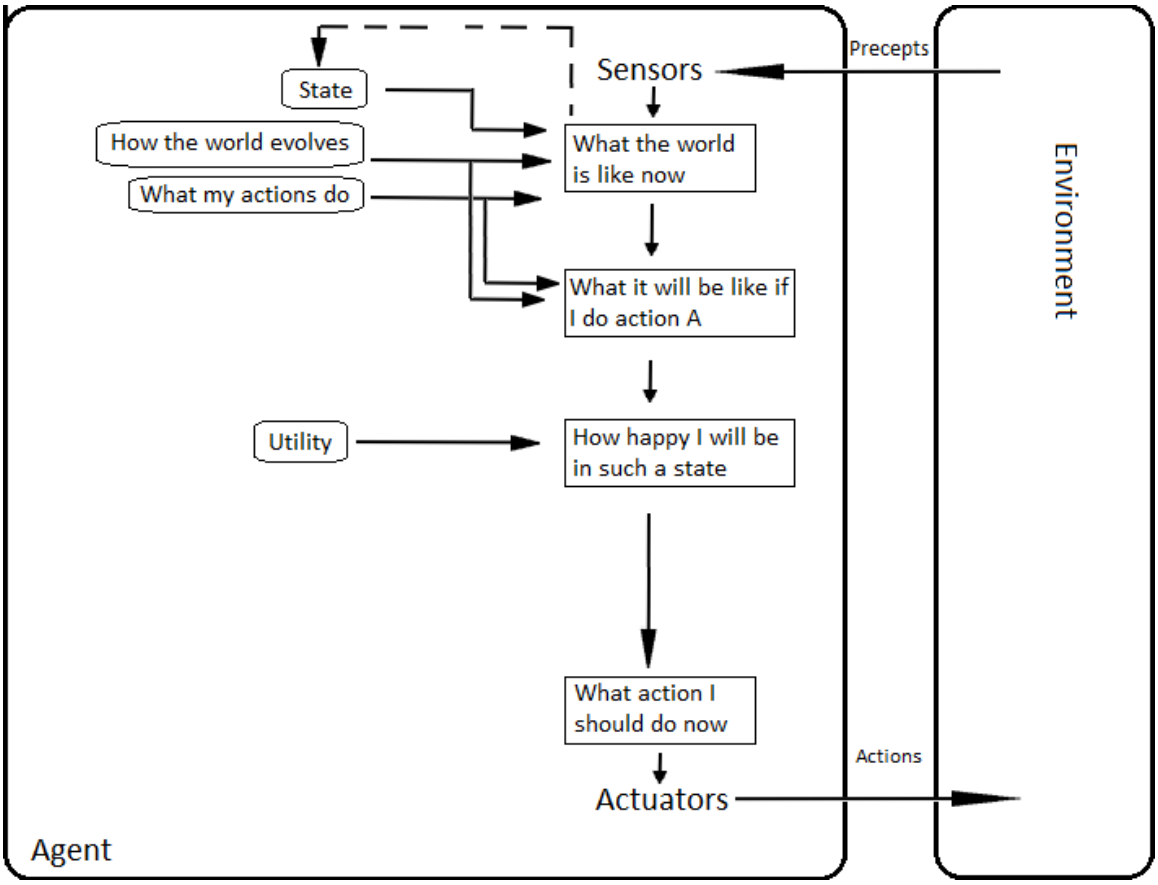
Simple reflex agent



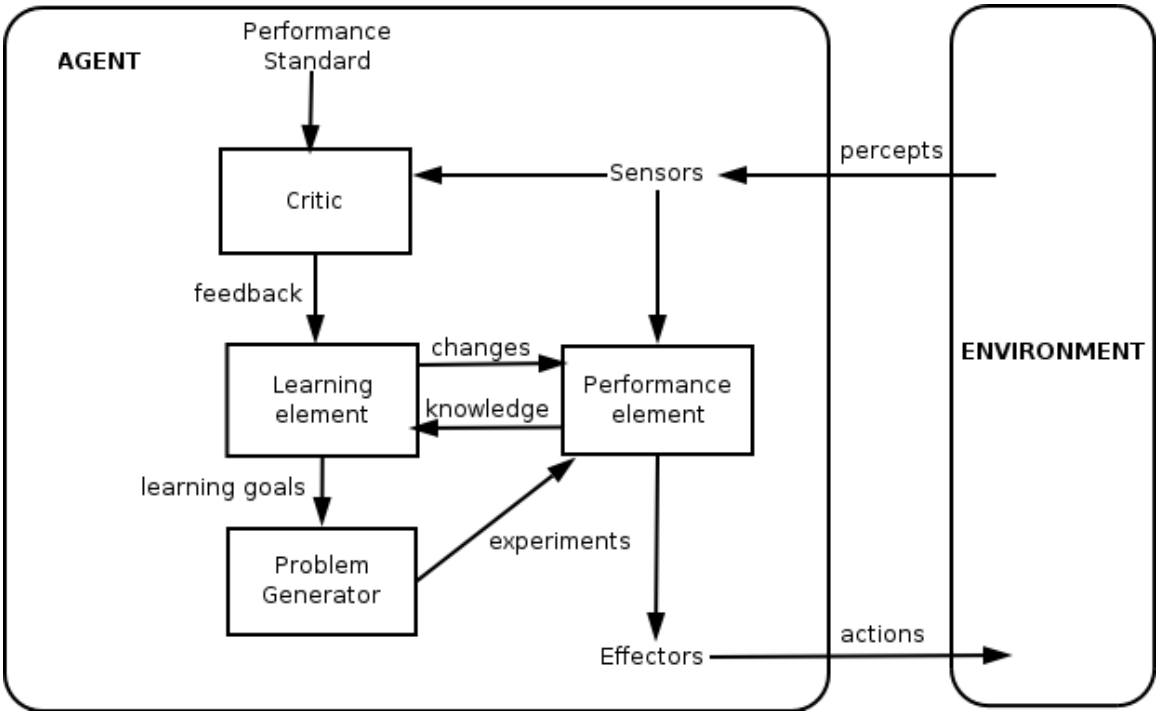
Model-based reflex agent



Model-based, goal-based agent



Model-based, utility-based agent



A general learning agent

Russell & Norvig (2003) group agents into five classes based on their degree of perceived intelligence and capability:

1. simple reflex agents
2. model-based reflex agents
3. goal-based agents
4. utility-based agents
5. learning agents

Simple reflex agents

Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the *condition-action rule*: if condition then action.

This agent function only succeeds when the environment is fully observable. Some reflex agents can also contain information on their current state which allows them to disregard conditions whose actuators are already triggered.

Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments. Note: If the agent can randomize its actions, it may be possible to escape from infinite loops.

Model-based reflex agents

A model-based agent can handle a partially observable environment. Its current state is stored inside the agent maintaining some kind of structure which describes the part of the world which cannot be seen. This knowledge about "how the world works" is called a model of the world, hence the name "model-based agent".

A model-based reflex agent should maintain some sort of internal model that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. It then chooses an action in the same way as the reflex agent.

Goal-based agents

Goal-based agents further expand on the capabilities of the model-based agents, by using "goal" information. Goal information describes situations that are desirable. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. Search and planning are the subfields of artificial intelligence devoted to finding action sequences that achieve the agent's goals.

In some instances the goal-based agent appears to be less efficient; it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

Utility-based agents

Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is. This measure can be obtained through the use of a *utility function* which maps a state to a measure of the utility of the state. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. The term utility, can be used to describe how "happy" the agent is.

A rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes- that is, the agent expects to derive, on average, given the probabilities and utilities of each outcome. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning.

Learning agents

Learning has an advantage that it allows the agents to initially operate in unknown environments and to become more competent than its initial knowledge alone might allow. The most important distinction is between the "learning element", which is responsible for making improvements, and the "performance element", which is responsible for selecting external actions.

The learning element uses feedback from the "critic" on how the agent is doing and determines how the performance element should be modified to do better in the future. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

The last component of the learning agent is the "problem generator". It is responsible for suggesting actions that will lead to new and informative experiences.

Other classes of intelligent agents

According to other sources, some of the sub-agents (not already mentioned in this treatment) that may be a part of an Intelligent Agent or a complete Intelligent Agent in themselves are:

- Decision Agents (that are geared to decision making);
- Input Agents (that process and make sense of sensor inputs - e.g. neural network based agents);
- Processing Agents (that solve a problem like speech recognition);
- Spatial Agents (that relate to the physical real-world);
- World Agents (that incorporate a combination of all the other classes of agents to allow autonomous behaviors).
- Believable agents - An agent exhibiting a personality via the use of an artificial character (the agent is embedded) for the interaction.
- Physical Agents - A physical agent is an entity which *percepts* through sensors and *acts* through actuators.
- Temporal Agents - A temporal agent may use time based stored information to offer instructions or data *acts* to a computer program or human being and takes program inputs *percepts* to adjust its next behaviors.

Hierarchies of agents

To actively perform their functions, Intelligent Agents today are normally gathered in a hierarchical structure containing many “sub-agents”. Intelligent sub-agents process and perform lower level functions. Taken together, the intelligent agent and sub-agents create a complete system that can accomplish difficult tasks or goals with behaviors and responses that display a form of intelligence.

Chapter 3

Agent-Based Model

An **agent-based model (ABM)** (also sometimes related to the term **multi-agent system** or **multi-agent simulation**) is a class of computational models for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) with a view to assessing their effects on the system as a whole. It combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming. Monte Carlo Methods are used to introduce randomness. ABMs are also called individual-based models.

The models simulate the simultaneous operations and interactions of multiple agents, in an attempt to re-create and predict the appearance of complex phenomena. The process is one of emergence from the lower (micro) level of systems to a higher (macro) level. As such, a key notion is that simple behavioral rules generate complex behavior. This principle, known as K.I.S.S. ("Keep it simple stupid") is extensively adopted in the modeling community. Another central tenet is that the whole is greater than the sum of the parts. Individual agents are typically characterized as boundedly rational, presumed to be acting in what they perceive as their own interests, such as reproduction, economic benefit, or social status, using heuristics or simple decision-making rules. ABM agents may experience "learning", adaptation, and reproduction.

Most agent-based models are composed of: (1) numerous agents specified at various scales (typically referred to as agent-granularity); (2) decision-making heuristics; (3) learning rules or adaptive processes; (4) an interaction topology; and (5) a non-agent environment.

History

The idea of agent-based modeling was developed as a relatively simple concept in the late 1940s. Since it requires computation-intensive procedures, it did not become widespread until the 1990s.

The history of the agent-based model can be traced back to the Von Neumann machine, a theoretical machine capable of reproduction. The device von Neumann proposed would follow precisely detailed instructions to fashion a copy of itself. The concept was then improved by von Neumann's friend Stanisław Ulam, also a mathematician; Ulam suggested that the machine be built on paper, as a collection of cells on a grid. The idea intrigued von Neumann, who drew it up—creating the first of the devices later termed cellular automata.

Another improvement was introduced by the mathematician John Conway. He constructed the well-known Game of Life. Unlike von Neumann's machine, Conway's Game of Life operated by tremendously simple rules in a virtual world in the form of a 2-dimensional checkerboard.

One of the earliest agent-based models in concept was Thomas Schelling's segregation model, which was discussed in his paper *Dynamic Models of Segregation* in 1971. Though Schelling originally used coins and graph paper rather than computers, his models embodied the basic concept of agent-based models as autonomous agents interacting in a shared environment with an observed aggregate, emergent outcome.

In the early 1980s, Robert Axelrod hosted a tournament of Prisoner's Dilemma strategies and had them interact in an agent-based manner to determine a winner. Axelrod would go on to develop many other agent-based models in the field of political science that examine phenomena from ethnocentrism to the dissemination of culture (Axelrod 1997).

In the late 1980s, Craig Reynolds' work on flocking models contributed to the development of some of the first biological agent-based models that contained social characteristics. He tried to model the reality of lively biological agents, known as artificial life, a term coined by Christopher Langton.

The first use of the word "agent" and a definition as it is currently used today is hard to track down. One candidate appears to be John Holland and John H. Miller's 1991 paper "Artificial Adaptive Agents in Economic Theory" which is based on an earlier conference presentation of theirs.

At the same time, during the 1980s, social scientists, mathematicians, operations researchers, and a scattering of people from other disciplines developed Computational and Mathematical Organization Theory (CMOT). This field grew as a special interest group of The Institute of Management Sciences (TIMS) and its sister society, the Operations Research Society of America (ORSA). Through the mid-1990s, the field focused on such issues as designing effective teams, understanding the communication required for organizational effectiveness, and the behavior of social networks. With the appearance of StarLogo in 1990, SWARM and NetLogo in the mid-1990s and RePast in 2000, as well as some custom-designed code, CMOT—later renamed Computational Analysis of Social and Organizational Systems (CASOS) -- incorporated more and more agent-based modeling. Samuelson (2000) is a good brief overview of the early history, and Samuelson (2005) and Samuelson and Macal (2006) trace the more recent

developments. Bonabeau (2002) is a good survey of the potential of agent-based modeling as of the time that its modelling software became widely available.

Kathleen M. Carley developed an early ABM, Construct , to explore the co-evolution of social networks and culture.

Joshua M. Epstein and Robert Axtell developed a large-scale ABM, the Sugarscape, to simulate and explore the role of social phenomenon such as seasonal migrations, pollution, sexual reproduction, combat, and transmission of disease and even culture.

Nigel Gilbert published the first textbook on Social Simulation: Simulation for the social scientist (1999) and established its most relevant journal: the Journal of Artificial Societies and Social Simulation.

In the late 1990s, the merger of TIMS and ORSA to form INFORMS, and the move by INFORMS from two meetings each year to one, helped to spur the CMOT group to form a separate society, the North American Association for Computational Social and Organizational Sciences (NAACSOS). Kathleen Carley, of Carnegie Mellon University, was a major contributor, especially to models of social networks, obtaining National Science Foundation funding for the annual conference and serving as the first President of NAACSOS. She was succeeded by David Sallach of the University of Chicago and Argonne National Laboratory, and then by Michael Prietula of Emory University. At about the same time NAACSOS began, the European Social Simulation Association (ESSA) and the Pacific Asian Association for Agent-Based Approach in Social Systems Science (PAAA), counterparts of NAACSOS, were organized. Nowadays, these three organizations collaborate internationally. The First World Congress on Social Simulation was held under their joint sponsorship in Kyoto, Japan, in August 2006. The Second World Congress was held in the northern Virginia suburbs of Washington, D.C., in July 2008, with George Mason University taking the lead role in local arrangements.

More recently, Ron Sun developed methods for basing agent-based simulation on models of human cognition, known as cognitive social simulation. Bill McKelvey, Suzanne Lohmann, Dario Nardi, Dwight Read and others at UCLA have also made significant contributions in organizational behavior and decision-making. Since 2001, UCLA has arranged a conference at Lake Arrowhead, California, that has become another major gathering point for practitioners in this field.

Theory

Most computational modeling research describes systems in equilibrium or as moving between equilibria. Agent-based modeling, however, using simple rules, can result in far more complex and interesting behavior.

The three ideas central to agent-based models are agents as objects, emergence, and complexity.

Agent-based models consist of dynamically interacting rule-based agents. The systems within which they interact can create real world-like complexity. These agents are:

- Intelligent and purposeful.
- Situated in space and time. They reside in networks and in lattice-like neighborhoods. The location of the agents and their responsive and purposeful behavior are encoded in algorithmic form in computer programs. The modeling process is best described as inductive. The modeler makes those assumptions thought most relevant to the situation at hand and then watches phenomena emerge from the agents' interactions. Sometimes that result is an equilibrium. Sometimes it is an emergent pattern. Sometimes, however, it is an unintelligible mangle.

In some ways, agent-based models complement traditional analytic methods. Where analytic methods enable humans to characterize the equilibria of a system, agent-based models allow the possibility of generating those equilibria. This generative contribution may be the most mainstream of the potential benefits of agent-based modeling. Agent-based models can explain the emergence of higher order patterns—network structures of terrorist organizations and the Internet, power law distributions in the sizes of traffic jams, wars, and stock market crashes, and social segregation that persists despite populations of tolerant people. Agent-based models also can be used to identify lever points, defined as moments in time in which interventions have extreme consequences, and to distinguish among types of path dependency.

Rather than focusing on stable states, the models consider a system's robustness—the ways that complex systems adapt to internal and external pressures so as to maintain their functionalities. The task of harnessing that complexity requires consideration of the agents themselves—their diversity, connectedness, and level of interactions.

Applications

Agent-based models have been used since the mid-1990s to solve a variety of business and technology problems. Examples of applications include supply chain optimization and logistics, modeling of consumer behavior, including word of mouth, social network effects, distributed computing, workforce management, and portfolio management. They have also been used to analyze traffic congestion. In these and other applications, the system of interest is simulated by capturing the behavior of individual agents and their interconnections. Agent-based modeling tools can be used to test how changes in individual behaviors will affect the system's emerging overall behavior.

Other models have analyzed the spread of epidemics, the threat of biowarfare, biological applications including population dynamics, the growth and decline of ancient civilizations, evolution of ethnocentric behavior, forced displacement/migration, language choice dynamics, and biomedical applications including inflammation and the human immune system. Agent-based models have also been used for developing decision support systems such as for breast cancer.

Recently, agent based modelling and simulation has been applied to various domains such as studying the impact of publication venues by researchers in the computer science domain (journals versus conferences). In addition, ABMS has been used to simulate information delivery in ambient assisted environments. In the domain of peer-to-Peer, ad-hoc and other self-organizing and complex networks, the usefulness of agent based modeling and simulation has been shown. The use of Computer Science based Formal Specification framework coupled with Wireless sensor networks and an Agent-based simulation has recently been demonstrated in.

Agent based evolutionary search or algorithm is a new research topic for solving complex optimization problems. Further details on the topic can be found in R. Sarker and T. Ray (2010) Agent based Evolutionary Approach: An Introduction, Agent Based Evolutionary Search, Springer series in Evolutionary Learning and Optimization, Springer, pp. 1–12.

Hardware

The Software described above is designed for serial von-Neumann computer architectures. This limits the speed and scalability of these systems. A recent development is the use of data-parallel algorithms on Graphics Processing Units GPUs for ABM simulation , and . The extreme memory bandwidth combined with the sheer number crunching power of multi-processor GPUs has enabled simulation of millions of agents at tens of frames per second.

Verification & Validation of Agent-Based Models

Verification and validation of simulation models is extremely important. Verification involves debugging the model to ensure it works correctly; whereas Validation ensures that you have built the right model. Verification and validation in the social sciences domain can be seen in. In Computational Economics, validation can be examined in. In, the author proposes face validation, sensitivity analysis, calibration and statistical validation. Discrete-Event Simulation Framework approach for the validation of Agent-Based systems has been proposed in. A comprehensive resource on empirical validation of agent-based models is

A formal approach for V&V of all agent-based models is based on building a VOMAS (Virtual Overlay Multi-Agent System), a software engineering based approach, where a virtual overlay Multi-agent system is developed alongside the agent-based model. The agents in the Multi-Agent System are able to gather data by generation of logs as well as provide run-time validation and verification support by watch agents and also agents to check any violation of invariants at run-time. These are set by the Simulation Specialist with help from the SME (Subject Matter Expert). An example of using VOMAS for Verification and Validation of a Forest Fire simulation model is given in

VOMAS provides a formal way of Validation and Verification. If you want to build a VOMAS, you need to start designing VOMAS agents along with the agents in the actual

simulation preferably from the start. So, in essence, by the time your simulation model is complete, you essentially have one model which contains two models:

1. An Agent Based Model of the intended system
2. An Agent Based Model of the VOMAS

Unlike all previous work on Verification and Validation, VOMAS agents ensure that the simulations are validated in-simulation i.e. even during execution. In case of any exceptional situations, which are programmed on the directive of the Simulation Specialist (SS), the VOMAS agents can report them. In addition, the VOMAS agents can be used to log key events for the sake of debugging and subsequent analysis of simulations. In other words, VOMAS allows for a flexible use of any given technique for the sake of Verification and Validation of an Agent-based Model in any domain.

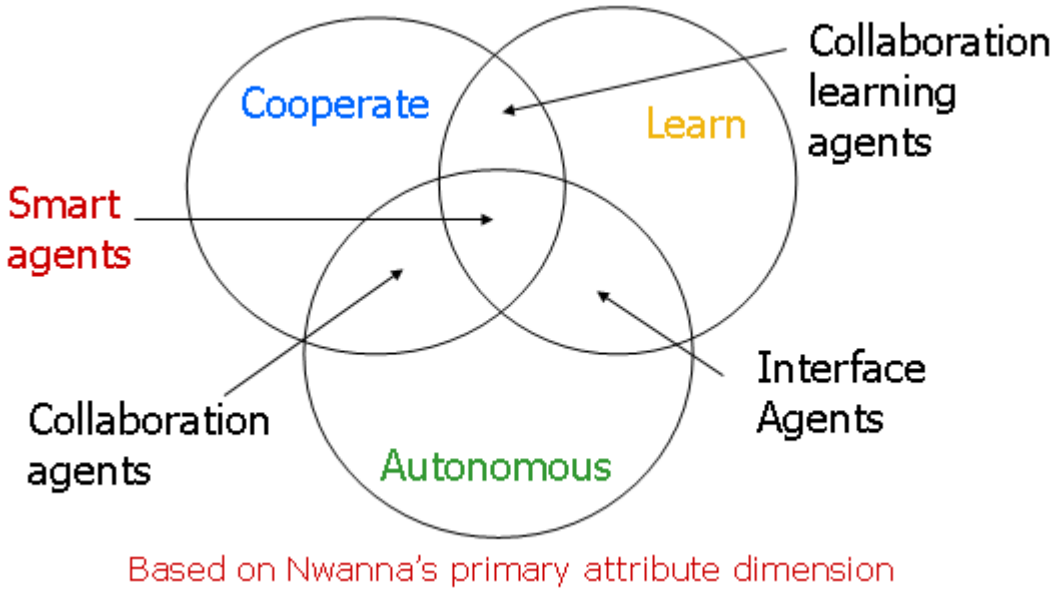
Chapter 4

Software Agent

In computer science, a **software agent** is a piece of software that acts for a user or other program in a relationship of agency, which derives from the Latin *agere* (to do): an agreement to act on one's behalf. Such "action on behalf of" implies the authority to decide which (and if) action is appropriate. The idea is that agents are not strictly invoked for a task, but activate themselves.

Related and derived concepts include *Intelligent agents* (in particular exhibiting some aspect of Artificial Intelligence, such as learning and reasoning), *autonomous* agents (capable of modifying the way in which they achieve their objectives), *distributed* agents (being executed on physically distinct computers), *multi-agent systems* (distributed agents that do not have the capabilities to achieve an objective alone and thus must communicate), and *mobile* agents (agents that can relocate their execution onto different processors).

Definition



Nwana's Category of Software Agent

The term "agent" describes a software abstraction, an idea, or a concept, similar to OOP terms such as methods, functions, and objects. The concept of an agent provides a convenient and powerful way to describe a complex software entity that is capable of acting with a certain degree of autonomy in order to accomplish tasks on behalf of its user. But unlike objects, which are defined in terms of *methods* and *attributes*, an agent is defined in terms of its behavior.

Various authors have proposed different definitions of agents, these commonly include concepts such as

- *persistence* (code is not executed on demand but runs continuously and decides for itself when it should perform some activity)
- *autonomy* (agents have capabilities of task selection, prioritization, goal-directed behaviour, decision-making without human intervention)
- *social ability* (agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task)
- *reactivity* (agents perceive the context in which they operate and react to it appropriately).

What an agent is not

It is not useful to prescribe what is, and what is not an agent. However contrasting the term with related concepts may help clarify its meaning:

Distinguishing agents from programs

Franklin & Graesser (1997) discuss four key notions that distinguish agents from arbitrary programs: reaction to the environment, autonomy, goal-orientation and persistence. Related and derived concepts include Intelligent agents (in particular exhibiting some aspect of Artificial Intelligence, such as learning and reasoning), autonomous agents (capable of modifying the way in which they achieve their objectives), distributed agents (being executed on physically distinct computers), multi-agent systems (distributed agents that do not have the capabilities to achieve an objective alone and thus must communicate), and mobile agents (agents that can relocate their execution onto different processors).

Intuitive distinguishing agents from objects

- Agents are more autonomous than objects.
- Agents have flexible behaviour, reactive, proactive, social.
- Agents have at least one thread of control but may have more.

(Wooldridge, 2002)

Distinguishing agents from expert systems

- Expert systems are not coupled to their environment;
- Expert systems are not designed for reactive, proactive behavior.
- Expert systems do not consider social ability

(Wooldridge, 2003)

Distinguishing intelligent software agents from intelligent agents in artificial intelligence

- Intelligent agents (also known as rational agents) are not just software programs, they may also be machines, human beings, communities of human beings (such as firms) or anything that is capable of goal directed behavior.

(Russell & Norvig 2003)

History

The concept of an agent can be traced back to Hewitt's Actor Model (Hewitt, 1977) - "A self-contained, interactive and concurrently-executing object, possessing internal state and communication capability."

To be more academic, software agent systems are a direct evolution from Multi-Agent Systems (MAS). MAS evolved from Distributed Artificial Intelligence (DAI),

Distributed Problem Solving (DPS) and Parallel AI (PAI), thus inheriting all characteristics (good and bad) from DAI and AI.

John Sculley's 1987 "Knowledge Navigator" video portrayed an image of a relationship between end-users and agents. Being an ideal first, this field experienced a series of unsuccessful top-down implementations, instead of a piece-by-piece, bottom-up approach. The range of agent types is now (from 1990) broad: WWW, search engines, etc.

Examples

Intelligent software agents

Haag (2006) suggests that there are only four essential types of intelligent software agents:

1. Buyer agents or shopping bots
2. User or personal agents
3. Monitoring-and-surveillance agents
4. Data Mining agents

Buyer agents (shopping bots)

Buyer agents travel around a network (i.e. the internet) retrieving information about goods and services. These agents, also known as 'shopping bots', work very efficiently for commodity products such as CDs, books, electronic components, and other one-size-fits-all products.

User agents (personal agents)

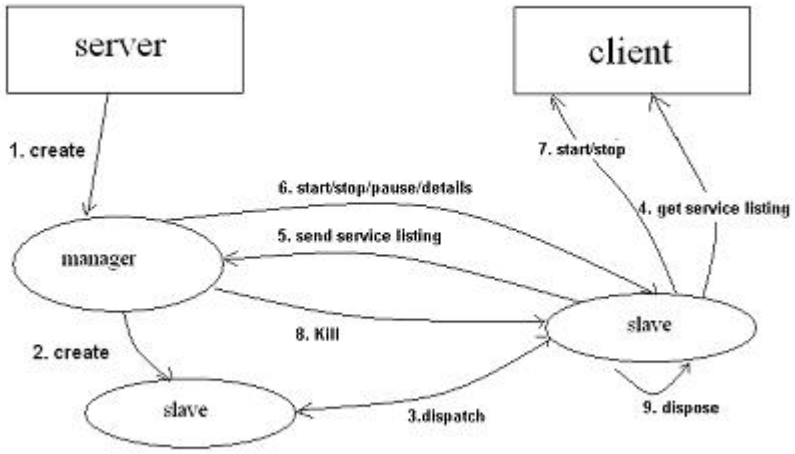
User agents, or personal agents, are intelligent agents that take action on your behalf. In this category belong those intelligent agents that already perform, or will shortly perform, the following tasks:

- Check your e-mail, sort it according to the user's order of preference, and alert you when important emails arrive.
- Play computer games as your opponent or patrol game areas for you.
- Assemble customized news reports for you. There are several versions of these, including CNN.
- Find information for you on the subject of your choice.
- Fill out forms on the Web automatically for you, storing your information for future reference
- Scan Web pages looking for and highlighting text that constitutes the "important" part of the information there
- "Discuss" topics with you ranging from your deepest fears to sports

- Facilitate with online job search duties by scanning known job boards and sending the resume to opportunities who meet the desired criteria
- Profile synchronization across heterogeneous social networks

Monitoring-and-surveillance (predictive) agents

Monitoring and Surveillance Agents are used to observe and report on equipment, usually computer systems. The agents may keep track of company inventory levels, observe competitors' prices and relay them back to the company, watch stock manipulation by insider trading and rumors, etc.



service monitoring

For example, NASA's Jet Propulsion Laboratory has an agent that monitors inventory, planning, and scheduling equipment ordering to keep costs down, as well as food storage facilities. These agents usually monitor complex computer networks that can keep track of the configuration of each computer connected to the network.

A special case of Monitoring-and-Surveillance agents are organizations of agents used to emulate the Human Decision Making process during tactical operations. The agents monitor the status of assets (ammunition, weapons available, platforms for transport, etc.) and receive Goals (Missions) from higher level agents. The Agents then pursue the Goals with the Assets at hand, minimizing expenditure of the Assets while maximizing Goal Attainment.

Data mining agents

This agent uses information technology to find trends and patterns in an abundance of information from many different sources. The user can sort through this information in order to find whatever information they are seeking.

A data mining agent operates in a data warehouse discovering information. A 'data warehouse' brings together information from lots of different sources. "Data mining" is the process of looking through the data warehouse to find information that you can use to take action, such as ways to increase sales or keep customers who are considering defecting.

'Classification' is one of the most common types of data mining, which finds patterns in information and categorizes them into different classes. Data mining agents can also detect major shifts in trends or a key indicator and can detect the presence of new information and alert you to it. For example, the agent may detect a decline in the construction industry for an economy; based on this relayed information construction companies will be able to make intelligent decisions regarding the hiring/firing of employees or the purchase/lease of equipment in order to best suit their firm.

Other examples

Some other examples of current Intelligent agents include some spam filters, game bots, and server monitoring tools. Search engine indexing bots also qualify as intelligent agents.

- User agent - for browsing the World Wide Web
- Mail transfer agent - For serving E-mail, such as *Microsoft Outlook*. Why? It communicates with the POP3 mail server, without users having to understand POP3 command protocols. It even has rule sets that filter mail for the user, thus sparing them the trouble of having to do it themselves.
- SNMP agent
- DAML (DARPA Agent Markup Language)
- Jason (multi-agent systems development platform)
- 3APL (Artificial Autonomous Agents Programming Language)
- GOAL Agent Programming Language
- Web Ontology Language (OWL)
- *daemons* in Unix-like systems.
- In Unix-style networking servers, *httpd* is an HTTP daemon which implements the HyperText Transfer Protocol at the root of the World Wide Web
- Management agents used to manage telecom devices
- Crowd simulation for safety planning or 3D computer graphics,
- Java Agent Template (JAT)

Design issues

Interesting issues to consider in the development of agent-based systems include

- how tasks are scheduled and how synchronization of tasks is achieved
- how tasks are prioritized by agents
- how agents can collaborate, or recruit resources,
- how agents can be re-instantiated in different environments, and how their internal state can be stored,
- how the environment will be probed and how a change of environment leads to behavioral changes of the agents
- how messaging and communication can be achieved,
- what hierarchies of agents are useful (e.g. task execution agents, scheduling agents, resource providers ...).

For software agents to work together efficiently they must share semantics of their data elements. This can be done by having computer systems publish their metadata.

The definition of *agent processing* can be approached from two interrelated directions:

- internal state processing and ontologies for representing knowledge
- interaction protocols - standards for specifying communication of tasks

Agent systems are used to model real world systems with concurrency or parallel processing.

- Agent Machinery - Engines of various kinds, which support the varying degrees of intelligence
- Agent Content - Data employed by the machinery in Reasoning and Learning
- Agent Access - Methods to enable the machinery to perceive content and perform actions as outcomes of Reasoning
- Agent Security - Concerns related to distributed computing, augmented by a few special concerns related to agents

The agent uses its access methods to go out into local and remote databases to forage for content. These access methods may include setting up news stream delivery to the agent, or retrieval from bulletin boards, or using a spider to walk the Web. The content that is retrieved in this way is probably already partially filtered – by the selection of the newsfeed or the databases that are searched. The agent next may use its detailed searching or language-processing machinery to extract keywords or signatures from the body of the content that has been received or retrieved. This abstracted content (or event) is then passed to the agent's Reasoning or inferencing machinery in order to decide what to do with the new content. This process combines the event content with the rule-based or knowledge content provided by the user. If this process finds a good hit or match in the new content, the agent may use another piece of its machinery to do a more detailed search on the content. Finally, the agent may decide to take an action based on the new

content; for example, to notify the user that an important event has occurred. This action is verified by a security function and then given the authority of the user. The agent makes use of a user-access method to deliver that message to the user. If the user confirms that the event is important by acting quickly on the notification, the agent may also employ its learning machinery to increase its weighting for this kind of event.

Impacts of software agents

It is unarguable that software agents are innovative technologies that may offer various benefits to their end users by automating complex or repetitive tasks. However, there are several potential organizational and cultural impacts of this technology that need to be considered. Organizational impacts include the transformation of the entire electronic commerce sector, operational encumbrance, and security overload. Software agents are able to quickly search the Internet, identify the best offers available online, and present this information to the end users in aggregate form. Therefore, users may not need to manually browse various websites of individual merchants; they are able to locate the best deal in a matter of seconds. At the same time, this increases price-based competition and transforms the entire electronic commerce sector into a uniform perfect competition market. The implementation of agents also requires additional resources from the companies, places an extra burden on their networks, and requires new security procedures. The cultural effects of the implementation of software agents include trust affliction, skills erosion, privacy attrition and social detachment. Some users may not feel entirely comfortable fully delegating important tasks to software applications. Those who start relying solely on intelligent agents may lose important skills, for example, relating to information literacy. In order to act on a user's behalf, a software agent needs to have a complete understanding of a user's profile, including his/her personal preferences. This, in turn, may lead to unpredictable privacy issues. When users start relying on their software agents more, especially, for communication activities, they may lose contact with other human users and look at the word with the eyes of their agents. It is these consequences that agent researchers and users need to consider dealing with intelligent agent technologies.

Chapter 5

GOAL Agent Programming Language

GOAL is an agent programming language for programming rational agents. GOAL agents derive their choice of action from their beliefs and goals. The language provides the basic building blocks to design and implement rational agents by means of a set of programming constructs. These programming constructs allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. The language provides an intuitive programming framework based on common sense or practical reasoning.

Overview

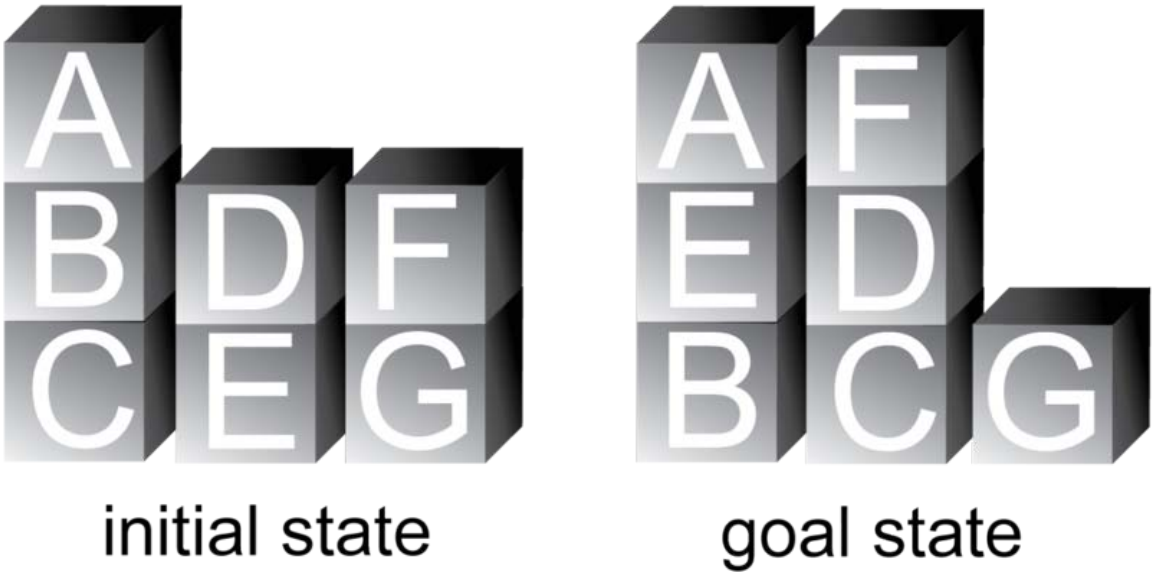
The main features of GOAL include:

- **Declarative beliefs**: Agents use a symbolic, logical language to represent the information they have, and their beliefs or knowledge about the environment they act upon in order to achieve their goals. This *knowledge representation language* is not fixed by GOAL but, in principle, may be varied according to the needs of the programmer.
- **Declarative goals**: Agents may have multiple goals that specify *what* the agent wants to achieve at some moment in the near or distant future. Declarative goals specify a state of the environment that the agent wants to establish, they do not specify actions or procedures how to achieve such states.
- **Blind commitment strategy**: Agents commit to their goals and drop goals only when they have been achieved. This commitment strategy, called a *blind* commitment strategy in the literature, is the *default* strategy used by GOAL agents. Rational agents are assumed to not have goals that they believe are already achieved, a constraint which has been built into GOAL agents by dropping a goal when it has been *completely* achieved.
- **Rule-based action selection**: Agents use so-called *action rules* to select actions, given their beliefs and goals. Such rules may *underspecify* the choice of action in

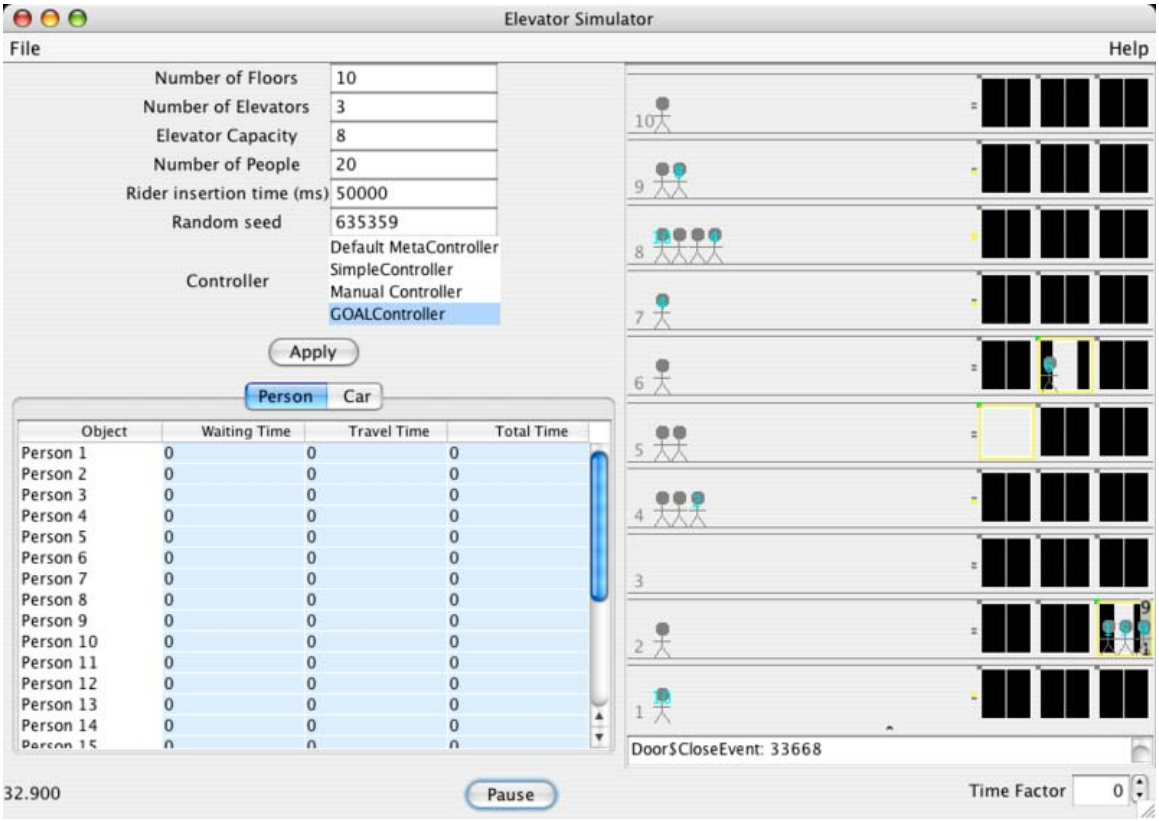
the sense that multiple actions may be performed at any time given the action rules of the agent. In that case, a GOAL agent will select an arbitrary enabled action for execution.

- **Policy-based intention modules**: Agents may focus their attention and put all their efforts on achieving a subset of their goals, using a subset of their actions, using only knowledge relevant to achieving those goals. GOAL provides modules to structure action rules and knowledge dedicated to achieving specific goals. Informally, modules can be viewed as policy-based intentions in the sense of Michael Bratman.
- **Communication at the knowledge level**: Agents may communicate with each other to exchange information, and to coordinate their actions. GOAL agents communicate using the knowledge representation language that is also used to represent their beliefs and goals.

GOAL Agent Program



An Example Blocks World Problem



Another Example: A GOAL Multi-Agent Elevator Controller

A GOAL agent program consists of six different sections, including the *knowledge*, *beliefs*, *goals*, *action rules*, *action specifications*, and *percept rules*, respectively. The knowledge, beliefs and goals are represented in a knowledge representation language such as Prolog, Answer set programming, SQL (or Datalog), or the Planning Domain Definition Language, for example. Below, we illustrate the components of a GOAL agent program using Prolog.

The overall structure of a GOAL agent program looks like:

```
main: <agentname> {
  <sections>
}
```

The GOAL agent code used to illustrate the structure of a GOAL agent is an agent that is able to solve Blocks world problems. The beliefs of the agent represent the current state of the Blocks world whereas the goals of the agent represent the goal state. The *knowledge* section listed next contains additional conceptual or domain knowledge related to the Blocks world domain.

```
knowledge{
  block(a), block(b), block(c), block(d), block(e), block(f), block(g).
  clear(table).
  clear(X) :- block(X), not(on(Y,X)).
```

```
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
```

Note that all the blocks listed in the knowledge section reappear in the *beliefs* section again as the position of each block needs to be specified to characterize the complete configuration of blocks.

```
beliefs{
  on(a,b), on(b,c), on(c,table), on(d,e), on(e,table), on(f,g),
on(g,table).
}
```

All known blocks also are present in the *goals* section which specifies a goal configuration which reuses all blocks.

```
goals{
  on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d),
on(g,table).
}
```

A GOAL agent may have multiple goals at the same time. These goals may even be conflicting as each of the goals may be realized at different times. For example, an agent might have a goal to watch a movie in the movie theater and to be at home (afterwards).

In GOAL, different notions of goal are distinguished. A **primitive goal** is a statement that follows from the goal base in conjunction with the concepts defined in the knowledge base. For example, `tower([a,e,b])` is a primitive goal and we write `goal(tower([a,e,b])` to denote this. Initially, `tower([a,e,b])` is also an **achievement goal** since the agent does not believe that a is on top of e, e is on top of b, and b is on the table. Achievement goals are primitive goals that the agent does not believe to be the case and are denoted by `a-goal(tower([a,e,b])`. It is also useful to be able to express that a **goal has been achieved**. `goal-a(tower([e,b])` is used to express, for example, that the tower `[e,b]` has been achieved with block e on top of block b. Both achievement goals as well as the notion of a goal achieved can be defined:

```
a-goal(formula) ::= goal(formula), not(bel(formula))
goal-a(formula) ::= goal(formula), bel(formula)
```

There is a significant literature on defining the concept of an achievement goal in the agent literature.

GOAL is a rule-based programming language. Rules are structured into modules. The *main module* of a GOAL agent specifies a strategy for selecting actions by means of action rules. The first rule below states that moving block X on top of block Y (or, possibly, the table) is an option if such a move is constructive, i.e. moves the block in position. The second rule states that moving a block X to the table is an option if block X is misplaced.

```
main module{
  program{
    if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then move(X,Y).
    if a-goal(tower([X|T])) then move(X,table).
  }
}
```

Actions, such as the move action used above, are specified using a STRIPS-style specification of preconditions and postconditions. A precondition specifies when the action can be performed (is enabled). A postcondition specifies what the effects of performing the action are.

```
actionspec{
  move(X,Y) {
    pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
    post{ not(on(X,Z)), on(X,Y) }
}
```

Finally, the *event module* consists of rules for processing events such as percepts received from the environment. The rule below specifies that for all percepts received that indicate that block X is on block Y, and X is believed to be on top of Z unequal to Y, the new fact on(X,Y) is to be added to the belief base and the atom on(X,Z) is to be removed.

```
event module{
  program{
    forall bel( percept(on(X,Y)), on(X,Z), not(Y=Z) ) do
insert(on(X,Y), not(on(X,Z))).
  }
}
```

Download

GOAL is available for download from the GOAL webpage hosted at the Delft University of Technology. Besides the GOAL installer the GOAL webpage provides the GOAL Programming Guide, GOAL IDE User Manual, and a Video Tutorial.

Related Agent Programming Languages

The GOAL agent programming language is related to but different from other agent programming languages such as AGENT0, AgentSpeak, 2APL, Golog, JACK Intelligent Agents, Jadex, and, for example, Jason. The distinguishing feature of GOAL are the concept of a declarative goal. Goals of a GOAL agent describe *what* an agent wants to achieve, not how to achieve it. Different from other languages, GOAL agents are committed to their goals and only remove a goal when it has been *completely* achieved. GOAL provides a programming framework with a strong focus on declarative programming and the reasoning capabilities required by rational agents.

Chapter 6

Deliberative Agent & Belief-Desire-Intention Software Model

Deliberative Agent

Deliberative agent (also known as intentional agent) is a sort of software agent used mainly in multi-agent system simulations. According to Wooldridge's definition, a deliberative agent is "one that possesses an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via symbolic reasoning".

Compared to reactive agents, which are able to reach their goal only by reacting reflexively on external stimuli, a deliberative agent's internal processes are more complex. The difference lies in fact, that deliberative agent maintains a symbolic representation of the world it inhabits. In other words, it possesses internal image of the external environment and is thus capable to plan its actions. Most commonly used architecture for implementing such behavior is Belief-Desire-Intention software model (BDI), where an agent's beliefs about the world (its image of a world), desires (goal) and intentions are internally represented and practical reasoning is applied to decide, which action to select.

There has been considerable research focused on integrating both reactive and deliberative agent strategies resulting in developing a compound called hybrid agent, which combines extensive manipulation with nontrivial symbolic structures and reflexive reactive responses to the external events.

How does deliberative agent work?

It has already been mentioned, that deliberative agents possess a) inherent image of an outer world and b) goal to achieve and is thus able to produce a list of actions (plan) to

reach the goal. In unfavorable conditions, when the plan is no more applicable, agent is usually able to recompute it.

The process of plan computing (or recomputing) is as follows:

- a sensory input is received by the *belief revision function* and agent's beliefs are altered
- *option generation function* evaluates altered beliefs and intentions and creates the options available to the agent. Agent's desires are constituted.
- *filter function* then considers current beliefs, desires and intentions and produces new intentions
- *action selection function* then receives intentions *filter function* and decides what action to perform

The deliberative agent requires symbolic representation with compositional semantics (e. g. data tree) in all major functions, for its deliberation is not limited to present facts, but construes hypotheses about possible future states and potentially also holds information about past (i.e. memory). These hypothetic states involve goals, plans, partial solutions, hypothetical states of the agent's beliefs, etc. It is evident, that deliberative process may become considerably complex and hardware killing.

History of a concept

Since the early 1970, the *AI planning community* has been involved in developing artificial *planning agent* (a predecessor of a deliberative agent), which would be able to choose a proper plan leading to a specified goal. These early attempts resulted in constructing simple planning system called STRIPS. It soon became obvious that STRIPS concept needed further improvement, for it was unable to effectively solve problems of even moderate complexity. In spite of considerable effort to raise the efficiency (for example by implementing *hierarchical* and *non-linear planning*), the system remained somewhat weak while working with any time-constrained system.

More successful attempts have been made in late 1980s to design *planning agents*. For example the *IPEM* (Integrated Planning, Execution and Monitoring system) had a sophisticated non-linear planner embedded. Further, Wood's *AUTODRIVE* simulated a behavior of deliberative agents in a traffic and Cohen's *PHOENIX* system was construed to simulate a forest fire management.

In 1976, Simon and Newell formulated the Physical Symbol System hypothesis, which claims, that both human and artificial intelligence have the same principle - symbol representation and manipulation. According to the hypothesis it follows, that there is no substantial difference between human and machine in intelligence, but just quantitative and structural - machines are much less complex. Such a provocative proposition must have become the object of serious criticism and raised a wide discussion, but the problem itself still remains unsolved in its merit until these days.

Further development of classical *symbolic AI* proved not to be dependent on final verifying the Physical Symbol System hypothesis at all. In 1988, Bratman, Israel and Pollack introduced *Intelligent Resource-bounded Machine Architecture* (IRMA), the first system implementing the Belief-Desire-Intention software model (BDI). IRMA exemplifies the standard idea of *deliberative agent* as it is known today: a software agent embedding the symbolic representation and implementing the BDI.

Efficiency of deliberative agents compared to reactive ones

Above-mentioned troubles with symbolic AI have led to serious doubts about the viability of such a concept, which resulted in developing an *reactive architecture*, which is based on wholly different principles. Developers of the new architecture have rejected using symbolic representation and manipulation as a base of any artificial intelligence. Reactive agents achieve their goals simply through reactions on changing environment, which implies reasonable computational modesty.

Even though deliberative agents consume much more system resources than their reactive colleagues, their results are significantly better just in few special situations, whereas it is usually possible to replace one deliberative agent with few reactive ones in many cases, without losing a substantial deal of the simulation result's adequacy. It seems that classical deliberative agents may be usable especially where correct action is required, for their ability to produce optimal, domain-independent solution. Deliberative agent often fails in changing environment, for it is unable to re-plan its actions quickly enough.

Belief-Desire-Intention Software Model

The **Belief-Desire-Intention (BDI) software model** (usually referred to simply, but ambiguously, as **BDI**) is a software model developed for programming intelligent agents. Superficially characterized by the implementation of an agent's *beliefs*, *desires* and *intentions*, it actually uses these concepts to solve a particular problem in agent programming. In essence, it provides a mechanism for separating the activity of selecting a plan (from a plan library) from the execution of currently active plans. Consequently, BDI agents are able to balance the time spent on deliberating about plans (choosing what to do) and executing those plans (doing it). A third activity, creating the plans in the first place (planning), is not within the scope of the model, and is left to the system designer and programmer.

Overview

In order to achieve this separation, the BDI software model implements the principal aspects of Michael Bratman's theory of human practical reasoning (also referred to as Belief-Desire-Intention, or BDI). That is to say, it implements the notions of belief, desire and (in particular) intention, in a manner inspired by Bratman. For Bratman, intention and desire are both pro-attitudes (mental attitudes concerned with action), but intention is distinguished as a conduct-controlling pro-attitude. He identifies commitment as the

distinguishing factor between desire and intention, noting that it leads to (1) temporal persistence in plans and (2) further plans being made on the basis of those to which it is already committed. The BDI software model partially addresses these issues. Temporal persistence, in the sense of explicit reference to time, is not explored. The hierarchical nature of plans is more easily implemented: a plan consists of a number of steps, some of which may invoke other plans. The hierarchical definition of plans itself implies a kind of temporal persistence, since the overarching plan remains in effect while subsidiary plans are being executed.

An important aspect of the BDI software model (in terms of its research relevance) is the existence of logical models through which it is possible to define and reason about BDI agents. Research in this area has led, for example, to the axiomatization of some BDI implementations, as well as to formal logical descriptions such as Anand Rao and Michael Georgeff's BDICTL. The latter combines a multiple-modal logic (with modalities representing beliefs, desires and intentions) with the temporal logic CTL*. More recently, Michael Wooldridge has extended BDICTL to define LORA (the Logic Of Rational Agents), by incorporating an action logic. In principle, LORA allows reasoning not only about individual agents, but also about communication and other interaction in a multi-agent system.

The BDI software model is closely associated with intelligent agents, but does not, of itself, ensure all the characteristics associated with such agents. For example, it allows agents to have private beliefs, but does not force them to be private. It also has nothing to say about agent communication. Ultimately, the BDI software model is an attempt to solve a problem that has more to do with plans and planning (the choice and execution thereof) than it has to do with the programming of intelligent agents.

BDI Agents

A BDI agent is a particular type of bounded rational software agent, imbued with particular *mental attitudes*, viz: Beliefs, Desires and Intentions (BDI).

Architecture

This section defines the idealized architectural components of a BDI system.

- **Beliefs**: Beliefs represent the informational state of the agent, in other words its beliefs about the world (including itself and other agents). Beliefs can also include inference rules, allowing forward chaining to lead to new beliefs. Using the term *belief* rather than *knowledge* recognizes that what an agent believes may not necessarily be true (and in fact may change in the future).
    - **Beliefset**: Beliefs are stored in database (sometimes called a *belief base* or a *belief set*), although that is an implementation decision.
- **Desires**: Desires represent the motivational state of the agent. They represent objectives or situations that the agent *would like* to accomplish or bring about. Examples of desires might be: *find the best price*, *go to the party* or *become rich*.

    - **Goals**: A goal is a desire that has been adopted for active pursuit by the agent. Usage of the term *goals* adds the further restriction that the set of active desires must be consistent. For example, one should not have concurrent goals to go to a party and to stay at home - even though they could both be desirable.
- **Intentions**: Intentions represent the deliberative state of the agent - what the agent *has chosen* to do. Intentions are desires to which the agent has to some extent committed. In implemented systems, this means the agent has begun executing a plan.
    - **Plans**: Plans are sequences of actions (recipes or knowledge areas) that an agent can perform to achieve one or more of its intentions. Plans may include other plans: my plan to go for a drive may include a plan to find my car keys. This reflects that in Bratman's model, plans are initially only partially conceived, with details being filled in as they progress.
- **Events**: These are triggers for reactive activity by the agent. An event may update beliefs, trigger plans or modify goals. Events may be generated externally and received by sensors or integrated systems. Additionally, events may be generated internally to trigger decoupled updates or plans of activity.

BDI Interpreter

This section defines an idealized BDI interpreter that provides the basis of the PRS linage of BDI systems:

1. initialize-state
2. repeat
    1. options: option-generator(event-queue)
    2. selected-options: deliberate(options)
    3. update-intentions(selected-options)
    4. execute()
    5. get-new-external-events()
    6. drop-unsuccessful-attitudes()
    7. drop-impossible-attitudes()
3. end repeat

This basic algorithm has been extended in many ways, for instance to support planning ahead, automated teamwork, and maintenance goals .

Limitations and Criticisms

The BDI software model is one example of a reasoning architecture for a single rational agent, and one concern in a broader multi-agent system. This section bounds the scope of concerns for the BDI software model, highlighting known limitations of the architecture.

- **Learning**: BDI agents lack any specific mechanisms within the architecture to learn from past behavior and adapt to new situations.

- **Three Attitudes**: Classical decision theorists and planning researches question the necessity of having all three attitudes, distributed AI researches question whether the three attitudes are sufficient.
- **Logics**: The multi-modal logics that underlie BDI (that do not have complete axiomatizations and are not efficiently computable) have little relevance in practice.
- **Multiple Agents**: In addition to not explicitly supporting learning, the framework may not be appropriate to learning behavior. Further, the BDI model does not explicitly describe mechanisms for interaction with other agents and integration into a multi-agent system..
- **Explicit Goals**: Most BDI implementations do not have an explicit representation of goals.
- **Lookahead**: The architecture does not have (by design) any lookahead deliberation or forward planning. This may not be desirable because adopted plans may use up limited resources, actions may not be reversible, task execution may take longer than forward planning, and actions may have undesirable side effects if unsuccessful.

BDI Agent Implementations

'Pure' BDI

- Procedural Reasoning System (PRS)
- IRMA (not implemented but can be considered as PRS with non-reconsideration)
- UM-PRS
- Distributed Multi-Agent Reasoning System (dMARS)
- AgentSpeak(L)
- AgentSpeak(RT)
- Agent Real-Time System (ARTS)
- JAM
- JACK Intelligent Agents
- JADEX (Open Source Project)
- Jason (Open Source Project)
- SPARK
- 3APL
- 2APL
- GOAL Agent Programming Language
- CogniTAO (Think-As-One)
- LS/TS - Living Systems Technology Suite

Extensions and Hybrid Systems

- JACK Teams
- CogniTAO (Think-As-One)
- LS/TS - Living Systems Technology Suite
- Brahms

Chapter 7

Contract Net Protocol & Semi Human Instinctive Artificial Intelligence

Contract Net Protocol

Contract Net Protocol (CNP) is a well known task sharing protocol that is used for task allocation in multi-agent systems and consists of a collection of nodes or software agents that form the contract net. Each node on the network can, at different times or for different tasks be a manager or a contractor.

When a node gets a composite task (or for any reason cannot solve the present task) it breaks the problem down into sub-tasks (If possible) and announces the sub-task to the contract net acting as a manager. Bids are then received from potential contractors which the winning contractor(s) are awarded the job(s).

The Contract Net

Task distribution is viewed as a kind of contract negotiation and happens in 5 stages.

1. Recognition
2. Announcement
3. Bidding
4. Awarding
5. Expediting

Recognition

An agent recognises it has a problem that it wants help with. The agent has a goal, and either:

- Realises it cannot achieve the goal in isolation - does not have the capability to fulfil the goal
- Realises it would prefer not to achieve the goal in isolation - typically because of solution quality, deadline, etc.

Announcement

The agent with the task sends out an announcement of the task which includes a specification of the task to be achieved. The specification must encode:

- Description of the task itself
- Any constraints
- Meta-task information

Bidding

Agents that receive the announcement decide themselves whether they should bid for the task. Factors that are taken into consideration are:

- The agent must decide whether it is capable of the expecting task
- The agent must determine the quality constrains and the price information (if relevant)

Awarding

Agents that send the task announcement must choose between the received bids and decide who to award the contract to. The result of this process is communicated to agents that submitted a bid.

Expediting

The successful contractor then expedites the task. This may involve the generation of further contract nets in the form of sub-contracting to complete the task.

Example uses of Contract Net Protocol

An electronic market place for buying and selling goods. For example a system where a user could specify the goods that they want as well as a maximum price that they are willing to pay. The agent programs then would find other user(s) willing to sell the goods within the desired price range. The user with the lowest price would then be selected to fulfill the contract. Other constraints could be applied such as delivery time and the location of the goods.

Semi Human Instinctive Artificial Intelligence

Semi Human Instinctive Artificial Intelligence (**SHIAI**) is a new Artificial Intelligence methodology, first designed to be used in RoboCup competitions. Nowadays it has been used to resolve many different problems.

Overview

The goal of SHIAI is to provide robots (or any other intelligent embedded system) with manlike instincts. SHIAI proposes a nondeterministic decision making theory based on Semi Human Instincts implemented by learned potential fields, using neural networks and fuzzy logic, offline and online learning algorithms, which enable the agent to perform in anonymous, dynamic and non-deterministic environments. SHIAI is like a newly born baby who uses his/her instincts and will gradually become more and more intelligent as the brain learns more about its environment. The use of a new world modeling method called ARPL in SHIAI enables the agent to perform better within anonymous environments where positioning is an important and complex issue.

History

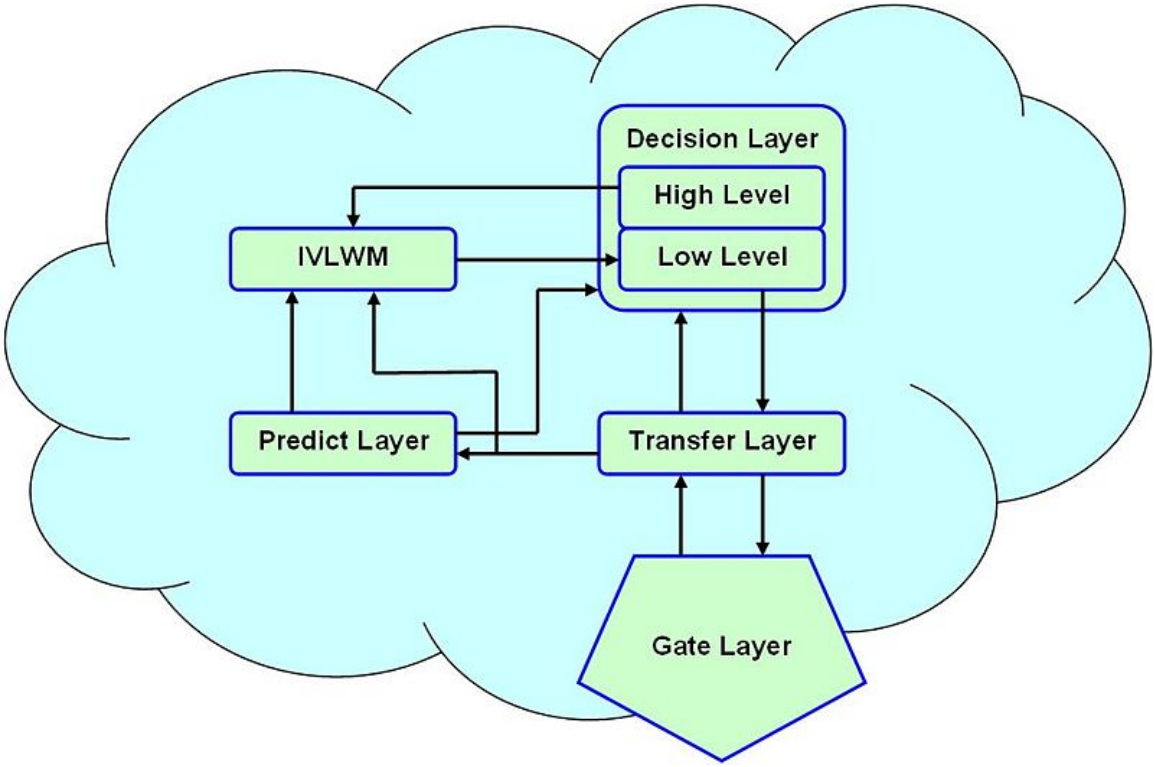
The research and work on this subject started from year 2000 and after 4 years of work and research and consulting with neurologists and psychologists resulted in presenting the MMLAI method. It was practically implemented and tested on the RoboCup Middle Size League (class F-2000) during RoboCup 2004 competitions, which revealed astonishing results some of which not even expected. This achievement encouraged us to work on it harder to cover its weaknesses and make it more optimized and adaptive to perform more efficient in noisy and anonymous environments such as soccer pitch. This led to the invention of SHIAI that was, like MMLAI, practically implemented and tested on the MiddleSize League Robots.

Principles

The first fundamental principle of this theory is based on instinct definition such that every problem has to be partitioned into its main and complex sections and then find a basic but reliable solution for each section using the laws of physics, chemistry, or mathematics. Providing the agent with these collections of instincts, we would have an agent that makes decisions without a particular knowledge and only by its defined instincts even if these decisions are false.
The Second principle is machine learning. In which there are two methods in SHIAI. As a baby learns (meaning both learning with supervisor and without supervisor) and gains experience, he/she would be able to make more optimized decisions and the chosen traveling paths in case of object avoidance will be more accurate.
The third principle of this theory is replacing quantity with quality even within calculations. That is, the volume of calculations is considerable reduced and is more similar to human brain. This will be done using ARPL that has eliminated the need for

exact global positioning. Therefore, relative polar localization substitutes the global positioning where no complex algorithm is required which decreases calculation errors and speeds up the decision making system.
The last principle is decision making under any circumstances. In fact, with this theory we make sure that there is nothing as unexpected condition because basically no conditions are defined in this theory to have unexpected condition.
In SHIAI, depending on the area of performance basic instincts will be defined for the intelligent agent, and then the agent itself nourishes its instincts using learning techniques and special analytical process of the surrounding environment to make more optimized and realistic decisions.

SHIAI Layers



SHIAI Layers

SHI-AI is consisted of five collaborating layers:

Gate Layer (GL)

Gate Layer performs as a gateway between SHI-AI and the surrounding world where all communications between SHIAI and the hardware world are done through this layer. This layer can be compatible with any hardware by making minor changes to the GL structure. Gate Layer is in contact with the Transfer Layer where gathered inputs by the Gate Layer are sent to Transfer Layer and the desired outputs are sent to the Gate Layer by the Transfer Layer.

Transfer Layer (TL)

Transfer Layer is responsible of parsing, correcting, and optimizing all the input and output data. This layer receives inputs from the Gate Layer and, if necessary, will make appropriate changes to the data formats and normalizes them to be ready to be sent to the upper and higher layers. Transfer Layer, also, recognizes errors in input data and will correct them before sending them to the upper or lower layers. This layer synchronously sends the same data that is being sent to the Decision Layer, to IVLWM, Learning and Predict Layers where data will be processed by each of the mentioned layers for different purposes. Finally when the optimum decision has been made by the Decision Layer the output will be sent to the Transfer Layer for optimization and then changed to be ready to be sent to the Gate Layer for final execution.

Decision Layer (DL)

The Decision Layer is consisted of two Low-Level and High-Level sub-layers.

- The Low-Level Decision is based on static laws which are called instinctive decision making in the real world. This decision making method enables the agent to make logical (but not optimized) decisions without a prior learning process, and furthermore provides the agent with unconscious decision making. Unconscious decision making is inevitable in virtual world and specially anonymous and on-deterministic environments. This sub-layer creates the main output of decision layer which is passed to the Transfer Layer to be executed.
- The High Level Decision recognizes and analysis its surrounding environment using appropriate data from the world. The decisions made in this layer are directly influencing the IVLWM. In fact, the decision making process of the agent is to first make a high

level decision having enough information from the world, predicted states, and additional information or commands from other active elements of the environment. This decision is then passed to IVLWM to model a world appropriate for the defined formulas of instincts.

Instinctive Virtual Layered World Modeler (IVLWM)

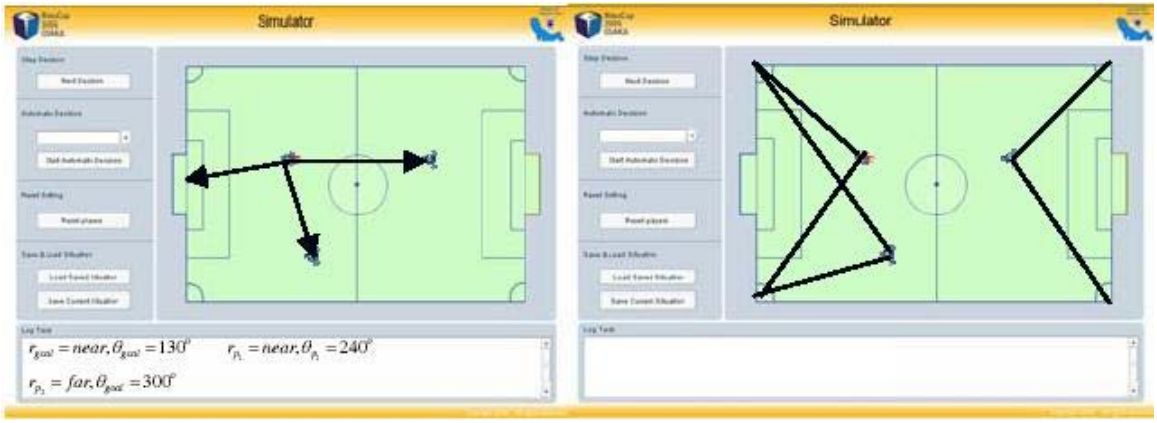
This layer is the most important layer of SHI-AI. As the name implies, IVLWM is responsible for converting the agent's surrounding world to a virtual world where affected by defined laws of instincts formation. The way instincts laws influence the real and virtual worlds depend on decision making conditions and learnings. This layer directly interacts with the learning layer. Thus, IVLWM generates more applicable and optimized virtual world having been fed by the learning process.

Predict Layer (PL)

The Predict Layer is the forecasting side of information processing. The aim here is to derive information about how the surrounding world will be like at some time $t_0 + \varepsilon_t$ in the future, for some ε_t &gt 0, by using data measured up to and including time ε_t. The predicted world is quiet useful for making high level decisions, specially in case of determining action strategies.

The collaboration and communication between layers is done via defined protocols. These protocols have been defined to be compatible with any area of performance by only applying minor changes to the low level making.

Agent Relative Polar Localization (ARPL)



Right picture is similar to the regular positioning techniques and the left picture is a sample of world generation using ARPL technique in SHI-AI Middle-Size Simulator

ARPL is a method for modeling agent's surrounding world based on polar coordinates of r and θ where r represents distance and θ represents angle. In this method, the agent retrieves the location of surrounding objects using the above mentioned coordinate relative to itself. That is, each object will have a distance and angle relative to the agent that results a polar position vector. The collection of these polar position vectors will make the agent's world. To have this method better understood we should now refer to RoboCup implementation of ARPL. In RoboCup implementation of APRL, we may have two ways of expressing the values of distance. The first one would be exact logarithmic value which is actually the logarithmic position of the object in a parabolic mirror, where robots vision is through an omni-directional parabolic mirror (this position is not the exact metric position of the object since it is not re-calculated through the parabolic formula of the mirror). The latter one which is used by Decision Layer is the linguistic fuzzy representation of the distance. This is done by dividing the circular visible area of the robot into several logarithmic sections defined as linguistic quantities like "close", "near", "far", etc. The magnitude of these ranges are increased exponentially from the closest point (tangent point) of the agent to the defined far most point.

Chapter 8

Agent-Based Model in Biology

Agent-based models have many applications in biology, primarily due to the characteristics of the modeling method. Agent-based modeling is a rule-based, computational modeling methodology that focuses on rules and interactions among the individual components or the agents of the system. The goal of this modeling method is to generate populations of the system components of interest and simulate their interactions in a virtual world. Agent-based models start with rules for behavior and seek to reconstruct, through computational instantiation of those behavioral rules, the observed patterns of behavior. Several of the characteristics of agent-based models important to biological studies include:

1. Modular structure: The behavior of an agent-based model is defined by the rules of its agents. Existing agent rules can be modified or new agents can be added without having to modify the entire model.
2. Emergent properties: Through the use of the individual agents that interact locally with rules of behavior, agent-based models result in a synergy that leads to a higher level whole with much more intricate behavior than those of each individual agent.
3. Abstraction: Either by excluding non-essential details or when details are not available, agent-based models can be constructed in the absence of complete knowledge of the system under study. This allows the model to be as simple and verifiable as possible.
4. Stochasticity: Biological systems exhibit behavior that appears to be random. The probability of a particular behavior can be determined for a system as a whole and then be translated into rules for the individual agents.

Before the agent-based model can be developed, one must choose the appropriate software or modeling toolkit to be used. Madey and Nikolai provide an extensive list of toolkits in their paper “Tools of the Trade: A Survey of Various Agent Based Modeling Platforms”. The paper seeks to provide users with a method of choosing a suitable toolkit

by examining five characteristics across the spectrum of toolkits: the programming language required to create the model, the required operating system, availability of user support, the software license type, and the intended toolkit domain. Some of the more commonly used toolkits include Swarm, NetLogo, RePast, and Mason. Listed below are summaries of several articles describing agent-based models that have been employed in biological studies. The summaries will provide a description of the problem space, an overview of the agent-based model and the agents involved, and a brief discussion of the model results.

Forest insect infestations

In the paper titled "Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations", an agent-based model was developed to simulate attack behavior of the Mountain Pine Beetle, *Dendroctonus ponderosae*, (MPB) in order to evaluate how different harvesting policies influence spatial characteristics of the forest and spatial propagation of the MPB infestation over time. About two-thirds of the land in British Columbia, Canada is covered by forests that are constantly being modified by natural disturbances such as fire, disease, and insect infestation. Forest resources make up approximately 15% of the province's economy, so infestations caused by insects such as the MPB can have significant impacts on the economy. The MPB outbreaks are considered a major natural disturbance that can result in widespread mortality of the Lodgepole pine tree, one of the most abundant commercial tree species in British Columbia. Insect outbreaks have resulted in the death of trees over areas of several thousand square kilometers.

The agent-based model developed for this study was designed to simulate the MPB attack behavior in order to evaluate how management practices influence the spatial distribution and patterns of insect population and their preferences for attacked and killed trees. Three management strategies were considered by the model: 1) no management, 2) sanitation harvest and 3) salvage harvest. In the model, the Beetle Agent represented the MPB behavior; the Pine Agent represented the forest environment and tree health evolution; the Forest Management Agent represented the different management strategies. The Beetle Agent follows a series of rules to decide where to fly within the forest and to select a healthy tree to attack, feed, and breed. The MPB typically kills host trees in its natural environment in order to successfully reproduce. The beetle larvae feed on the inner bark of mature host trees, eventually killing them. In order for the beetles to reproduce, the host tree must be sufficiently large and have thick inner bark. The MPB outbreaks end when the food supply decreases to the point that there is not enough to sustain the population or when climatic conditions become unfavorable for the beetle. The Pine Agent simulates the resistance of the host tree, specifically the Lodgepole pine tree, and monitors the state and attributes of each stand of trees. At some point in the MPB attack, the number of beetles per tree reaches the host tree capacity. When this point is reached, the beetles release a chemical to direct beetles to attack other trees. The Pine Agent models this behavior by calculating the beetle population density per stand and passes the information to the Beetle Agents. The Forest Management Agent was used, at the stand level, to simulate two common silviculture practices (sanitation and

salvage) as well as the strategy where no management practice was employed. With the sanitation harvest strategy, if a stand has an infestation rate greater than a set threshold, the stand is removed as well as any healthy neighbor stand when the average size of the trees exceeded a set threshold. For the salvage harvest strategy, a stand is removed even it is not under a MPB attack if a predetermined number of neighboring stands are under a MPB attack.

The study considered a forested area in the North-Central Interior of British Columbia of approximately 560 hectare. The area consisted primarily of Lodgepole pine with smaller proportions of Douglas fir and White spruce. The model was executed for five time steps, each step representing a single year. Thirty simulation runs were conducted for each forest management strategy considered. The results of the simulation showed that when no management strategy was employed, the highest overall MPB infestation occurred. The results also showed that the salvage forest management technique resulted in a 25% reduction in the number of forest strands killed by the MPB, as opposed to a 19% reduction by the salvage forest management strategy. In summary, the results show that the model can be used as a tool to build forest management policies.

Invasive species

Invasive species refers to "non-native" plants and animals that adversely affect the environments they invade. The introduction of invasive species may have environmental, economic, and ecological implications. In the paper titled "An Agent-Based Model of Border Enforcement for Invasive Species Management", an agent-based model is presented that was developed to evaluate the impacts of port-specific and importer-specific enforcement regimes for a given agricultural commodity that presents invasive species risk. Ultimately, the goal of the study was to improve the allocation of enforcement resources and to provide a tool to policy makers to answer further questions concerning border enforcement and invasive species risk.

The agent-based model developed for the study considered three types of agents: invasive species, importers, and border enforcement agents. In the model, the invasive species can only react to their surroundings, while the importers and border enforcement agents are able to make their own decisions based on their own goals and objectives. The invasive species has the ability to determine if it has been released in an area containing the target crop, and to spread to adjacent plots of the target crop. The model incorporates spatial probability maps that are used to determine if an invasive species becomes established. The study focused on shipments of broccoli from Mexico into California through the ports of entry Calexico, California and Otay Mesa, California. The selected invasive species of concern was the crucifer flea beetle (*Phyllotreta cruciferae*). California is by far the largest producer of broccoli in the United States and so the concern and potential impact of an invasive species introduction through the chosen ports of entry is significant. The model also incorporated a spatially explicit damage function that was used to model invasive species damage in a realistic manner. Agent-based modeling provides the ability to analyze the behavior of heterogeneous actors, so three different types of importers were considered that differed in terms of commodity infection rates (high, medium, and

low), pretreatment choice, and cost of transportation to the ports. The model gave predictions on inspection rates for each port of entry and importer and determined the success rate of border agent inspection, not only for each port and importer but also for each potential level of pretreatment (no pretreatment, level one, level two, and level three).

The model was implemented and ran in NetLogo, version 3.1.5. Spatial information on the location of the ports of entry, major highways, and transportation routes was included in the analysis as well as a map of California broccoli crops layered with invasive species establishment probability maps. BehaviorSpace, a software tool integrated with NetLogo, was used to test the effects of different parameters (e.g. shipment value, pretreatment cost) in the model. On average, 100 iterations were calculated at each level of the parameter being used, where an iteration represented a one-year run.

The results of the model showed that as inspection efforts increase, importers increase due care, or the pretreatment of shipments, and the total monetary loss of California crops decreases. The model showed that importers respond differently to an increase in inspection effort. Some importers responded to increased inspection rate by increasing pretreatment effort, while others chose to avoid shipping to a specific port, or shopped for another port. An important result of the model results is that it can show or provide recommendations to policy makers about the point at which importers may start to shop for ports, such as the inspection rate at which port shopping is introduced and the importers associated with a certain level of pest risk or transportation cost are likely to make these changes. Another interesting outcome of the model is that when inspectors were not able to learn to respond to an importer with previously infested shipments, damage to California broccoli crops was estimated to be $150 million. However, when inspectors were able to increase inspection rates of importers with previous violations, damage to the California broccoli crops was reduced by approximately 12%. The model provides a mechanism to predict the introduction of invasive species from agricultural imports and their likely damage. Equally as important, the model provides policy makers and border control agencies with a tool that can be used to determine the best allocation of inspectional resources.

Aphid population dynamics

In the article titled “Aphid Population Dynamics in Agricultural Landscapes: An Agent-based Simulation Model”, an agent-based model is presented to study the population dynamics of the bird cherry-oat aphid, *Rhopalosiphum padi* (L.). The study was conducted in a five square kilometer region of North Yorkshire, a county located in the Yorkshire and the Humber region of England. The agent-based modeling method was chosen because of its focus on the behavior of the individual agents rather than the population as a whole. The authors propose that traditional models that focus on populations as a whole do not take into account the complexity of the concurrent interactions in ecosystems, such as reproduction and competition for resources which may have significant impacts on population trends. The agent-based modeling approach also allows modelers to create more generic and modular models that are more flexible

and easier to maintain than modeling approaches that focus on the population as a whole. Other proposed advantages of agent-based models include realistic representation of a phenomenon of interest due to the interactions of a group of autonomous agents, and the capability to integrate quantitative variables, differential equations, and rule based behavior into the same model.

The model was implemented in the modeling toolkit Repast using the JAVA programming language. The model was ran in daily time steps and focused on the autumn and winter seasons. Input data for the model included habitat data, daily minimum, maximum, and mean temperatures, and wind speed and direction. For the Aphid agents, age, position, and morphology (alate or apterous) were considered. Age ranged from 0.00 to 2.00, with 1.00 being the point at which the agent becomes an adult. Reproduction by the Aphid agents is dependent on age, morphology, and daily minimum, maximum, and mean temperatures. Once nymphs hatch, they remain in the same location as their parents. The morphology of the nymphs is related to population density and the nutrient quality of the aphid's food source. The model also considered mortality among the Aphid agents, which is dependent on age, temperatures, and quality of habitat. The speed at which an Aphid agent ages is determined by the daily minimum, maximum, and mean temperatures. The model considered movement of the Aphid agents to occur in two separate phases, a migratory phase and a foraging phase, both of which affect the overall population distribution.

The study started the simulation run with an initial population of 10,000 alate aphids distributed across a grid of 25 meter cells. The simulation results showed that there were two major population peaks, the first in early autumn due to an influx of alate immigrants and the second due to lower temperatures later in the year and a lack of immigrants. Ultimately, it is the goal of the researchers to adapt this model to simulate broader ecosystems and animal types.

Aquatic population dynamics

In the article titled "Exploring Multi-Agent Systems In Aquatic Population Dynamics Modeling", a model is proposed to study the population dynamics of two species of macrophytes. Aquatic plants play a vital role in the ecosystems in which they live as they may provide shelter and food for other aquatic organisms. However, they may also have harmful impacts such as the excessive growth of non-native plants or eutrophication of the lakes in which they live leading to anoxic conditions. Given these possibilities, it is important to understand how the environment and other organisms affect the growth of these aquatic plants to allow mitigation or prevention of these harmful impacts.

Potamogeton pectinatus is one of the aquatic plant agents in the model. It is an annual growth plant that absorbs nutrients from the soil and reproduces through root tubers and rhizomes. Reproduction of the plant is not impacted by water flow, but can be influenced by animals, other plants, and humans. The plant can grow up to two meters tall, which is a limiting condition because it can only grow in certain water depths, and most of its biomass is found at the top of the plant in order to capture the most sunlight possible. The

second plant agent in the model is *Chara aspera*, also a rooted aquatic plant. One major difference in the two plants is that the latter reproduces through the use of very small seeds called oospores and bulbills which are spread via the flow of water. *Chara aspera* only grows up to 20 cm and requires very good light conditions as well as good water quality, all of which are limiting factors on the growth of the plant. *Chara aspera* has a higher growth rate than *Potamogeton pectinatus* but has a much shorter life span. The model also considered environmental and animal agents. Environmental agents considered included water flow, light penetration, and water depth. Flow conditions, although not of high importance to *Potamogeton pectinatus*, directly impact the seed dispersal of *Chara aspera*. Flow conditions affect the direction as well as the distance the seeds will be distributed. Light penetration strongly influences *Chara aspera* as it requires high water quality. Extinction coefficient (EC) is a measure of light penetration in water. As EC increases, the growth rate of *Chara aspera* decreases. Finally, depth is important to both species of plants. As water depth increases, the light penetration decreases making it difficult for either species to survive beyond certain depths.

The area of interest in the model was a lake in the Netherlands named Lake Veluwe. It is a relatively shallow lake with an average depth of 1.55 meters and covers about 30 square kilometers. The lake is under eutrophication stress which means that nutrients are not a limiting factor for either of the plant agents in the model. The initial position of the plant agents in the model was randomly determined. The model was implemented using Repast software package and was executed to simulate the growth and decay of the two different plant agents, taking into account the environmental agents previously discussed as well as interactions with other plant agents. The results of the model execution show that the population distribution of *Chara aspera* has a spatial pattern very similar to the GIS maps of observed distributions. The authors of the study conclude that the agent rules developed in the study are reasonable to simulate the spatial pattern of macrophyte growth in this particular lake.

Chapter 9

Osmius

Osmius

Osmius Desktop

Developer(s)	Peopleware SL
Initial release	July 14, 2006
Stable release	10.7 / July 23, 2010; 8 months ago
Development status	Active
Written in	C++, Java, Groovy
Operating system	Linux, Windows, Solaris
Available in	English, Spanish, French
Type	System monitoring, Business monitoring, KPI tracking

Osmius is a monitoring tool prepared to deal with thousands of devices, servers, applications and business indicators with incredible performance. Osmius was designed from the beginning to offer both the technical and the business view of your systems so the user can see that Server "Moon" is down but also what Services and Applications are affected. Osmius keeps track of the state and availability of every component and check

the Service Level Agreements regularly providing Business Intelligence and DataMining tools and Control Panels.

Overview

These are, in short, Osmius main features:

- **Prepared to monitor Everything**: In addition to CPU Load and Network traffic, why don't you monitor the *number of sells per hour*? Or the time to upload a document to your CMS?
- **SLA management**. Services, dependencies, propagation rules, capacity planning and ITIL best practices.
- **Performance**: More than 60.000 events per minute. Thousands of devices (things) and millions of events.
- **Multiplatform**: From Unix to Windows, Databases from MySQL to Oracle, Virtual Machine Engines, etc. Easy to develop new agents.
- **Customize it!** Use and integrate your own technical team scripts, add new SNPM or WMI events. Get new plugins from the Osmius portal or, better, make your owns.
- **Integrated GIS**. Locate your instances and leverage on the GIS engine capabilities.
- **Notification system**. Want to receive alerts into your mobile? Send alerts via e-mail, jabber or SMS to the 24×7 support team?
- **AutoDiscovery**: Discover the elements in your network by type: Databases, Webs, Servers, SNMP devices.
- **Customized Desktop**: Based on Widgets.
- **Mobile Console**: Check from iPhone or Android devices the real time state or the SLA Dash board.

- **Professional Support**: Peopleware is the company behind Osmius providing continuous improvements, bug fixing and development and support services.

History

Osmius started as an idea from some of the final development team members to address the problem of 2000's monitoring tools:

- Hard to implement and difficult to understand.
- Privative tools were very expensive and closed to learn and to expand.
- Free Tools were too technical, complicated and based on heavy scripting.
- Documentation is expensive or incomplete (or non-existent).
- Few of them were Business Oriented and ITIL integrated tools.

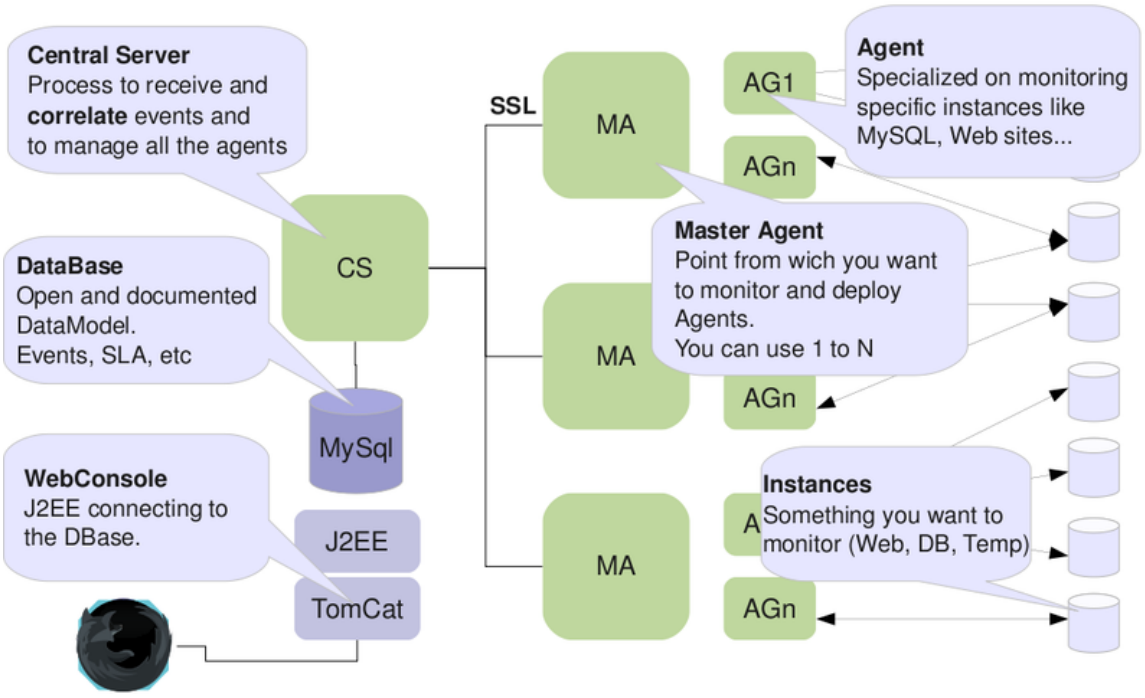
After spending years in the monitoring career, it was clear that there is room and that there are needs about monitoring systems and services, and that an initially small team could design a new tool with these traits:

- Easy to understand. Few and powerful concepts.
- Easy to administer. No scattered scripting and no more lack of central management.
- Robust and scalable. Real time event processing and tiny footprints on the monitored "things".
- Business Oriented. SLA management, offering technical impact analysis as well as business views.
- Prepared to monitor KPIs whether they are CPU Load or number of sells in Michigan.
- Open: Open Source, open documentation.
- Professional: Support, development and consulting services.

In 2010 Osmius is a consolidated product with thousands of download from SourceForge.

Architecture

Osmius is a web based interface tool from which the user can manage the overall infrastructure that can be as simple as a single Central Osmius server, or use Master Agents from where monitor and administer monitoring and monitored platform.



You can choose between intrusive (agent based) or external monitoring depending on your needs and restrictions. If any the connections with the agents are properly encrypted and there are process in the Central Server that deal with tasks such as sending new configuration parameters, prepare a server to monitor new systems or applications or to receive measurements in a robust and reliable way.

Osmius uses MySQL and the underlying storage engine (InnoDB) and is prepared to receive millions of events every day. Osmius implements a Round Robin Database policy so the user can apply event grouping as the data gets older and loose relevance.

New monitoring capabilities can be added creating user defined events or adding plug-ins to the platform, and there are modules to create new events based on current measures and to derive new data using statistical functions.

The Osmius Engine (agents, master agen and, central processes) is based on ACE ADAPTIVE Communication Environment and thanks to that we have multiplatform capabilities (code once, compile many) and real time as well as very small footprints.

Monitoring Capabilities

Osmius agents can monitor devices, systems and applications in Windows, Linux, Solaris and HP-UX and they normally rely on native APIs and connection pools to reduce footprints. Every **Osmius agent** can be configured to use "silent mode" to prevent network resources starvation and they are intelligent enough to recover lost from lost connections and even from process failures.

Operative systems

Linux: local and remote monitoring. Monitors uptime, system resource consumption, processes running state, resource consumption per process, filesystems, system temperature, network traffic.

Solaris: local monitoring. Monitors uptime, system resource consumption (cpu, memory, swap ...), processes and daemons running state, filesystems, users connected ...

Windows: local and remote monitoring. Monitors uptime, system resources consumption (cpu, memory, disk, virtual memory ...), processes and services running state, Windows EventLog, network traffic, resources consumption per process, your own WMI queries

Databases

MS SQL Server: local and remote monitoring. Monitors memory used by database server, splitted pages, used space by log and data, locks, number of connections, free pages, batch request ...

MySQL: local and remote monitoring. Monitors connections, threads, network traffic, queries per second, locks, slow queries, uptime, open tables ...

Oracle: local and remote monitoring. Monitors availability, uptime, locks, connections, invalid objects, performance problems, extensions problems, used space, slow queries, tablespaces usage, data and instructions cache success percentage, cursor consumption ...

PostgreSQL: local and remote monitoring. Monitors availability, number of connections, percentage of maximum connections used and database size in Megabytes.

SNMP Devices

Osmius agent to obtain v1 and v2c SNMP **variables** and **traps** from **any SNMP device**.

In addition Osmius have a preconfigured agent to monitors **generic CISCO routers**.Osmius has **Autodiscovery** of new traps.

Applications

Apache: local and remote monitoring. Monitors availability, uptime, request and a lot of data from your Apache server.

Asterisk: local and remote monitoring. Using AMI monitors availability, uptime, extensions state and mailbox messages.

Exchange: local and remote monitoring. Monitors Exchange 2003 and 2007 performance counters such as connected users, queues size, delivered, sent and failed messages, RPC requests and latency.

IIS: local and remote monitoring. Monitors uptime, connections, network traffic, request types and IIS errors.

Tomcat: local and remote monitoring. Using JMX monitors availability, uptime, applications running state, deployment status, memory usage, threads, pending objects and processor usage.

Technologies

SSH: local and remote monitoring. Monitors any SSH able device and use your own commands.

WMI: local and remote monitoring. Monitors any Microsoft and Windows WMI application.

IP: ping time reply, service availability such us http, ftp, ssh, telnet, imap, pop3, smtp ... at any IP address.

IPMI: local and remote monitoring. Monitors availability, uptime and any data from IPMI able device sensors.

LOG: local monitoring. Monitors log file size and content using regular expressions.

WEB: local and remote monitoring. Monitors ,with extraordinary performance, web availability, load time, transfer time, download size and content (using regular expressions) of any http or https source.

Check for updates about Osmius monitoring capabilities here.

Performance

In order to guarantee Osmius scalability and its smooth running in data intensive installations Osmius is tested in high-performance-requirement environments. Over an Intel Core Duo 2.5 GHz:

- Process 1.500 events each minute.
- Store millions of measures.
- Monitor thousands of instances.
- Manage the SLAs of thousands of services.
- Desploy 1000 agents in 15 minutes aprox.

Check about Osmius requirements and benchmarking results here

Releases

Osmius releases are numbered the same way as Ubuntu does. That is two figures for the year and the next two for the month.

Osmius 10.07 corresponds to July, 2010 release.

The Osmius development Team create a new version every month, but they are only made public twice a year, normally in January and July every year.

Chapter 10

Pandora FMS

Pandora FMS

Pandora FMS v 3.0 Dashboard

Original author(s)	Sancho Lerena
Developer(s)	Ártica ST
Initial release	October 14, 2004
Stable release	3.2.1 / February 23, 2011; 35 days ago
Development status	Active
Written in	Perl, PHP
Operating system	Linux
Available in	English, Spanish, Italian, Polish
Type	Network monitoring
License	GNU General Public License

or proprietary EULA

Pandora FMS (for *Pandora Flexible Monitoring System*) is software solution for monitoring computer networks. Pandora FMS allows monitoring in a visual way the status and performance of several parameters from different operating systems, servers, applications and hardware systems such as firewalls, proxies, databases, web servers or routers.

Pandora FMS can be deployed in almost any operating system. It features remote monitoring (WMI, SNMP, TCP. UDP, ICMP, HTTP...) and it can also use agents. An agent is available for each platform. It can also monitor hardware systems with a TCP/IP stack, such as load balancers, routers, network switches, printers or firewalls.

Pandora FMS has several servers that process and get information from different sources, using WMI for gathering remote Windows information, a predictive server, a plug-in server which makes complex user-defined network tests, an advanced export server to replicate data between different sites of Pandora FMS, a network discovery server, and an SNMP Trap console.

Released under the terms of the GNU General Public License, Pandora FMS is free software.

Components

Pandora Servers and SNMP Console

In Pandora FMS architecture, servers are the core of the system because they are the recipients of bundles of information. They also generate monitoring alerts. It is possible to have different modular configurations for the servers: several servers for very big systems, or just a single server. Servers are also responsible for inserting the gathered data into Pandora's database. It is possible to have several Pandora Servers connected to a single Database.

Servers are developed in Perl and work on any platform that has the required modules. Pandora was originally developed for Linux.

Web console

Pandora's user interface allows people to operate and manage the monitoring system. It is developed in PHP and depends on a database and a web server. It can work in in a wide range of platforms: Linux, Solaris, Microsoft Windows, AIX and others. Several web consoles can be deployed in the same system if required.

Agents

Agents are daemons or services that can monitor any numeric parameter, boolean status, string or numerical incremental data and/or condition. They can be developed in any language (as Shellscript, WSH, Perl or C). They run on any type of platform (Microsoft, AIX, Solaris, Linux, IPSO, Mac OS or FreeBSD), also SAP, because the agents can communicate with the Pandora FMS Servers to send data in XML using SSH, FTP, NFS, Tentacle (protocol) or any data transfer mean.

Database

The database module is the core module of Pandora. All the information of the system resides here. For example, all data gathered by agents, configuration defined by administrator, events, incidents, audit info, etc. are stored in the database.

At present, only MySQL database is supported, although support for others is planned to be added in the future.

Releases

Stable releases

Pandora FMS has been stable since version 1.0.

Version	Date	Information
3.2	December 27, 2010	Performance improved, smartphone web console, topology maps improved, policy manager improved, new file distribution for agents and new languages supported in reports.
3.1	June 01, 2010	
3.0	December 29, 2009	Flash interactive graphs, alert improvement and redesign, important improvements on windows agent, new SNMP trap console
2.1	March 3, 2009	Multi server support, Tentacle server and client included in packages, usability and performance improvements
2.0	September 3, 2008	New WMI, Plugin and Prediction Server. Better network auto discovery and topology detection, correlated alerts, log parser capable agents (win, Unix), planned downtimes, better reporting and more.
1.3.1	April 30, 2008	New transfer protocol: Tentacle (protocol), new server options
1.3	October 12, 2007	New logo, new recon server

1.2	December 5, 2006	New logo, renamed as Pandora FMS, changed homepage
1.1	May 13, 2005	New agents, console and documentation
1.0	October 14, 2004	First stable release (known as pandoramon)

Software Appliances

With the release of Pandora FMS 3.0 the pandora team had made available several software appliances of Pandora FMS.

VMWare or VirtualBox Image

Pandora Team created an OpenSUSE 11.1 VMware test image with final version of Pandora FMS 3.0. This image could be used with VMWare Player, VMWare Server or VirtualBox.

It has all the Pandora FMS components installed and MySQL, Apache2 and PHP with all modules dependencies resolved, including Pear Graph for reporting.

Chapter 11

Procedural Reasoning System

In Artificial Intelligence, the **Procedural Reasoning System (PRS)** is a framework for constructing real-time reasoning systems that can perform complex tasks in dynamic environments. It is based on the notion of a Rational agent or Intelligent agent using the Belief-Desire-Intention software model.

Overview

A user application is predominately defined, and provided to a PRS system is a set of *knowledge areas*. Each knowledge area is a piece of procedural knowledge that specifies how to do something, e.g., how to navigate down a corridor, or how to plan a path (in contrast with robotic architectures where the programmer just provides a model of what the states of the world are and how the agent's primitive actions affect them). Such a program, together with the PRS interpreter, is used to control the agent.

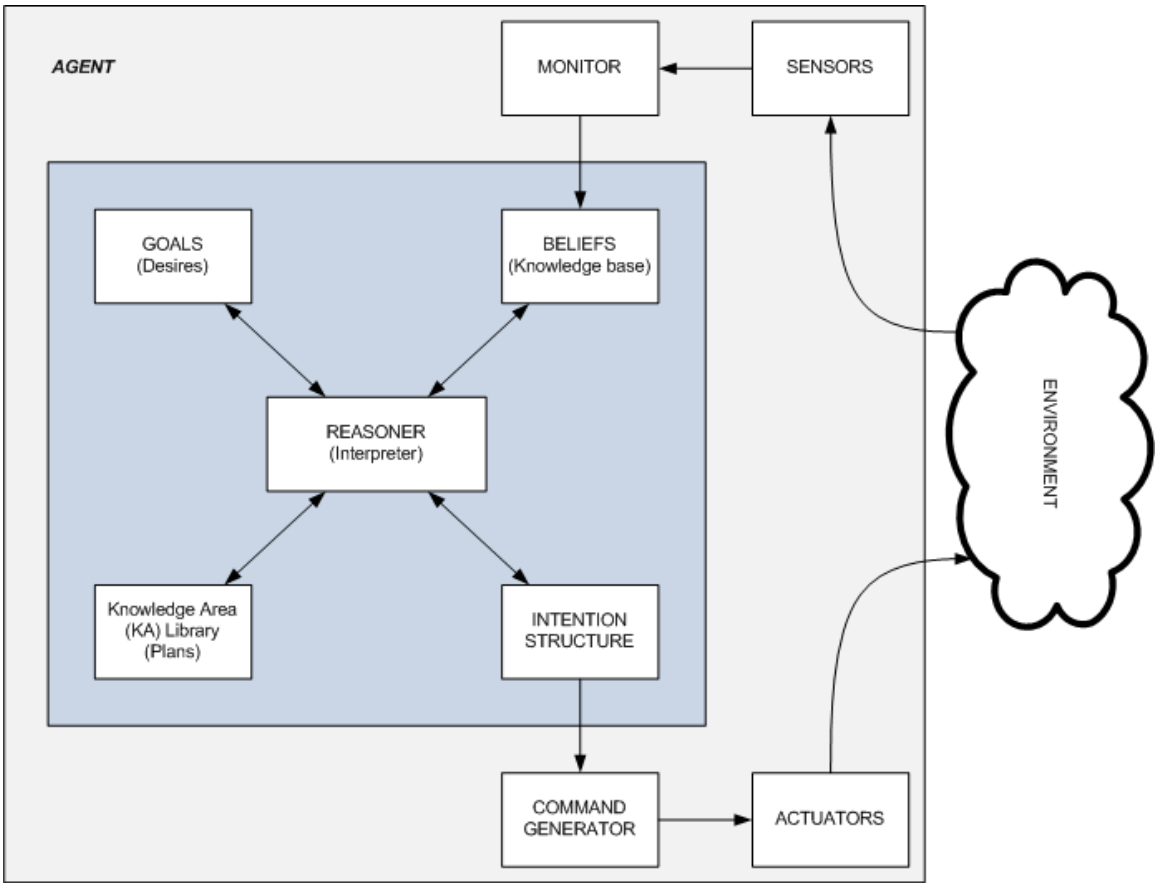
The interpreter is responsible for maintaining beliefs about the world state, choosing which goals to attempt to achieve next, and choosing which knowledge area to apply in the current situation. How exactly these operations are performed might depend on domain-specific meta-level knowledge areas. Unlike traditional AI planning systems that generate a complete plan at the beginning, and replan if unexpected things happen, PRS interleaves planning and doing actions in the world. At any point, the system might only have a partially specified plan for the future.

PRS is based on the BDI or belief-desire-intention framework for intelligent agents. Beliefs consist of what the agent believes to be true about the current state of the world, desires consist of the agent's goals, and intentions consist of the agent's current plans for achieving those goals. Furthermore, each of these three components is typically *explicitly* represented somewhere within the memory of the PRS agent at runtime, which is in contrast to purely reactive systems, such as the subsumption architecture.

History

The PRS was developed by the Artificial Intelligence Center at SRI International during the 1980s. Many were involved in the development of the PRS, not limited to Michael Georgeff, Amy L. Lansky, and François Félix Ingrand. The framework was responsible for exploiting and popularizing the Belief-Desire-Intention model in software for control of an Intelligent agent. The seminal application of the framework was a fault detection system for the Reaction Control System of the NASA Space Shuttle Discovery. Development on the PRS continued at the Australian Artificial Intelligence Institute through to the late 1990s which lead to the development of a C++ implementation and extension called dMARS.

Architecture



Depiction of the PRS Architecture.

The system architecture of PRS is composed of the following components:

- **Database** for beliefs about the world, represented using first order predicate calculus.
- **Goals** to be realized by the system as conditions over an interval of time on internal and external state descriptions (beliefs).

- **Knowledge Areas** (KA) or plans that define sequences of low-level actions toward achieving a goal in specific situations.
- **Intentions** that include those KA that have been selection for current and eventual execution.
- **Interpreter** or inference mechanism that manages the system.

Features

The PRS was developed for embedded application in dynamic and real-time environments. As such it specifically addressed the limitations of other contemporary control and reasoning architectures like Expert Systems and the Blackboard system. The following define the general requirements for the development of the PRS:

- asynchronous event handling
- guaranteed reaction and response types
- procedural representation of knowledge
- handling of multiple problems
- reactive and goal-directed behavior
- focus of attention
- reflective reasoning capabilities
- continuous embedded operation
- handling of incomplete of inaccurate data
- handling of transients
- modeling delayed feedback
- operator control

Applications

The seminal application of the PRS was a monitoring and fault detection system for the Reaction Control System (RCS) on the NASA space shuttle. The RCS provides propulsive forces from a collection of jet thrusters and controls attitude of the space shuttle. A PRS-based fault diagnostic system was developed and tested using a simulator. It included over 100 KAs and over 25 meta level KAs. RCS specific KAs were written by space shuttle mission controllers. It was implemented on the Symbolics 3600 Series LISP machine and used multiple communicating instances of PRS. The system maintained over 1000 facts about the RCS, over 650 facts for the forward RCS alone and half of which are updated continuously during the mission. A version of the PRS was used to monitor the Reaction Control System on the NASA Space Shuttle Discovery.

PRS was tested on Shakey the robot including navigational and simulated jet malfunction scenarios based on the space shuttle. Later applications included a network management monitor called the Interactive Real-time Telecommunications Network Management System (IRTNMS) for Telecom Australia.

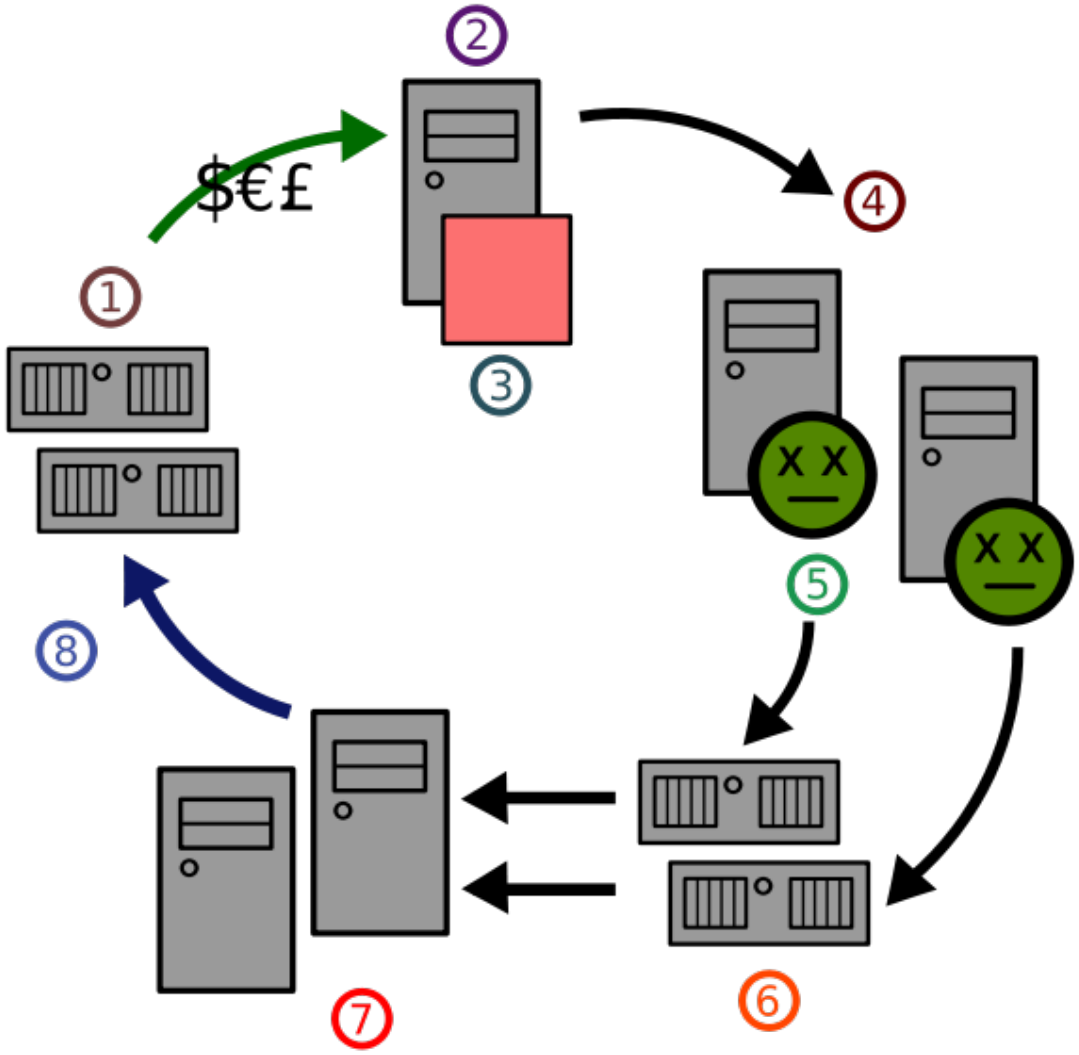
Extensions

The following list the major implementations and extensions of the PRS architecture.

- UM-PRS
- OpenPRS (formerly C-PRS and Propice)
- AgentSpeak
- Distributed Multi-Agent Reasoning System (dMARS)
- JAM
- JACK Intelligent Agents
- SRI Procedural Agent Realization Kit (SPARK)
- PRS-CL

Chapter 12

Storm Botnet



The typical lifecycle of spam that originates from a botnet:
(1) Spammer's web site (2) Spammer (3) Spamware (4) Infected computers (5) Virus or trojan (6) Mail servers (7) Users (8) Web traffic

The **Storm botnet** or **Storm worm botnet** (not to be confused with StormBot, which is a TCL script that is not malicious) is a remotely controlled network of "zombie" computers

(or "botnet") that has been linked by the Storm Worm, a Trojan horse spread through e-mail spam. Some have estimated that by September 2007 the Storm botnet was running on anywhere from 1 million to 50 million computer systems. Other sources have placed the size of the botnet to be around 250,000 to 1 million compromised systems. More conservatively, one network security analyst claims to have developed software that has crawled the botnet and estimates that it controls 160,000 infected computers. The Storm botnet was first identified around January 2007, with the Storm worm at one point accounting for 8% of all malware on Microsoft Windows computers.

The Storm botnet has been used in a variety of criminal activities. Its controllers and the authors of the Storm Worm have not yet been identified. The Storm botnet has displayed defensive behaviors that indicated that its controllers were actively protecting the botnet against attempts at tracking and disabling it. The botnet has specifically attacked the online operations of some security vendors and researchers who attempted to investigate the botnet. Security expert Joe Stewart revealed that in late 2007, the operators of the botnet began to further decentralize their operations, in possible plans to sell portions of the Storm botnet to other operators. Some reports as of late 2007 indicated the Storm botnet to be in decline, but many security experts reported that they expect the botnet to remain a major security risk online, and the United States Federal Bureau of Investigation considers the botnet a major risk to increased bank fraud, identity theft, and other cybercrimes.

The botnet is reportedly powerful enough as of September 2007 to force entire countries off the Internet, and is estimated to be capable of executing more instructions per second than some of the world's top supercomputers. However, it is not a completely accurate comparison, according to security analyst James Turner, who said that comparing a botnet to a supercomputer is like comparing an army of snipers to a nuclear weapon. Bradley Anstis, of the United Kingdom security firm Marshal, said, "The more worrying thing is bandwidth. Just calculate four million times a standard ADSL connection. That's a lot of bandwidth. It's quite worrying. Having resources like that at their disposal—distributed around the world with a high presence and in a lot of countries—means they can deliver very effective distributed attacks against hosts."

Origins

First detected on the Internet in January 2007, the Storm botnet and worm are so-called because of the storm-related subject lines its infectious e-mail employed initially, such as "230 dead as storm batters Europe." Later provocative subjects included, "Chinese missile shot down USA aircraft," and "U.S. Secretary of State Condoleezza Rice has kicked German Chancellor Angela Merkel." It is suspected by some information security professionals that well-known fugitive spammers, including Leo Kuvayev, may be involved in the operation and control of the Storm botnet. According to technology journalist Daniel Tynan, writing under his "Robert X. Cringely" pseudonym, a great portion of the fault for the existence of the Storm botnet lay with Microsoft and Adobe Systems. Other sources state that Storm Worm's primary method of victim acquisition is through enticing users via frequently changing social engineering (confidence trickery)

schemes. According to Patrick Runald, the Storm botnet has a strong American focus, and likely has agents working to support it within the United States. Some experts, however, believe the Storm botnet controllers are Russian, some pointing specifically at the Russian Business Network, citing that the Storm software mentions a hatred of the Moscow-based security firm Kaspersky Lab, and includes the Russian word *"buldozhka,"* which means "bulldog."

Composition

The botnet, or zombie network, comprises computers running Microsoft Windows as their operating system. Once infected, a computer becomes known as a bot. This bot then performs automated tasks—anything from gathering data on the user, to attacking web sites, to forwarding infected e-mail—without its owner's knowledge or permission. Estimates indicate that 5,000 to 6,000 computers are dedicated to propagating the spread of the worm through the use of e-mails with infected attachments; 1.2 billion virus messages have been sent by the botnet through September 2007, including a record 57 million on August 22, 2007 alone. Lawrence Baldwin, a computer forensics specialist, was quoted as saying, "Cumulatively, Storm is sending billions of messages a day. It could be double digits in the billions, easily." One of the methods used to entice victims to infection-hosting web sites are offers of free music, for artists such as Beyoncé Knowles, Kelly Clarkson, Rihanna, The Eagles, Foo Fighters, R. Kelly, and Velvet Revolver. Signature-based detection, the main defense of most computer systems against virus and malware infections, is hampered by the large number of Storm variants.

Back-end servers that control the spread of the botnet and Storm worm automatically re-encode their distributed infection software twice an hour, for new transmissions, making it difficult for anti-virus vendors to stop the virus and infection spread. Additionally, the location of the remote servers which control the botnet are hidden behind a constantly changing DNS technique called ‘fast flux’, making it difficult to find and stop virus hosting sites and mail servers. In short, the name and location of such machines are frequently changed and rotated, often on a minute by minute basis. The Storm botnet's operators control the system via peer-to-peer techniques, making external monitoring and disabling of the system more difficult. There is no central "command-and-control point" in the Storm botnet that can be shut down. The botnet also makes use of encrypted traffic. Efforts to infect computers usually revolve around convincing people to download e-mail attachments which contain the virus through subtle manipulation. In one instance, the botnet's controllers took advantage of the National Football League's opening weekend, sending out mail offering "football tracking programs" which did nothing more than infect a user's computer. According to Matt Sergeant, chief anti-spam technologist at MessageLabs, "In terms of power, [the botnet] utterly blows the supercomputers away. If you add up all 500 of the top supercomputers, it blows them all away with just 2 million of its machines. It's very frightening that criminals have access to that much computing power, but there's not much we can do about it." It is estimated that only 10%-20% of the total capacity and power of the Storm botnet is currently being used.

Computer security expert Joe Stewart detailed the process by which compromised machines join the botnet: attempts to join the botnet are made by launching a series of EXE files on the said machine, in stages. Usually, they are named in a sequence from *game0.exe* through *game5.exe*, or similar. It will then continue launching executables in turn. They typically perform the following:

1. `game0.exe` - Backdoor/downloader
2. `game1.exe` - SMTP relay
3. `game2.exe` - E-mail address stealer
4. `game3.exe` - E-mail virus spreader
5. `game4.exe` - Distributed denial of service (DDoS) attack tool
6. `game5.exe` - Updated copy of Storm Worm dropper

At each stage the compromised system will connect into the botnet; fast flux DNS makes tracking this process exceptionally difficult. This code is run from *%windir%\system32\wincom32.sys* on a Windows system, via a kernel rootkit, and all connections back to the botnet are sent through a modified version of the eDonkey/Overnet communications protocol.

Methodology

The Storm botnet and its variants employ a variety of attack vectors, and a variety of defensive steps exist as well. The Storm botnet was observed to be defending itself, and attacking computer systems that scanned for Storm virus-infected computer systems online. The botnet will defend itself with DDoS counter-attacks, to maintain its own internal integrity. At certain points in time, the Storm worm used to spread the botnet has attempted to release hundreds or thousands of versions of itself onto the Internet, in a concentrated attempt to overwhelm the defenses of anti-virus and malware security firms. According to Joshua Corman, an IBM security researcher, "This is the first time that I can remember ever seeing researchers who were actually afraid of investigating an exploit." Researchers are still unsure if the botnet's defenses and counter attacks are a form of automation, or manually executed by the system's operators. "If you try to attach a debugger, or query sites it's reporting into, it knows and punishes you instantaneously. [Over at] SecureWorks, a chunk of it DDoS-ed [distributed-denial-of-service attacked] a researcher off the network. Every time I hear of an investigator trying to investigate, they're automatically punished. It knows it's being investigated, and it punishes them. It fights back," Corman said.

Spameater.com as well as other sites such as 419eater.com and Artists Against 419, both of which deal with 419 spam e-mail fraud, have experienced DDoS attacks, temporarily rendering them completely inoperable. The DDoS attacks consist of making massed parallel network calls to those and other target IP addresses, overloading the servers' capacities and preventing them from responding to requests. Other anti-spam and anti-fraud groups, such as the Spamhaus Project, were also attacked. The webmaster of Artists Against 419 said that the website's server succumbed after the attack increased to over 100Mbit. As the theoretical maximum is never practically attainable, the number of

machines used may have been as much as twice that many, and similar attacks were perpetrated against over a dozen anti-fraud site hosts. Jeff Chan, a spam researcher, stated, "In terms of mitigating Storm, it's challenging at best and impossible at worst since the bad guys control many hundreds of megabits of traffic. There's some evidence that they may control hundreds of Gigabits of traffic, which is enough to force some countries off the Internet."

The Storm botnet's systems also take steps to defend itself locally, on victims' computer systems. The botnet, on some compromised systems, creates a computer process on the Windows machine that notifies the Storm systems whenever a new program or other processes begin. Previously, the Storm worms locally would tell the other programs — such as anti-virus, or anti-malware software, to simply not run. However, according to IBM security research, versions of Storm also now simply "fool" the local computer system to run the hostile program successfully, but in fact, they are not doing anything. "Programs, including not just AV exes, dlls and sys files, but also software such as the P2P applications BearShare and eDonkey, will appear to run successfully, even though they didn't actually do anything, which is far less suspicious than a process that gets terminated suddenly from the outside," said Richard Cohen of Sophos. Compromised users, and related security systems, will assume that security software is running successfully when it in fact is not.

On September 17, 2007, a Republican Party website in the United States was compromised, and used to propagate the Storm worm and botnet. In October 2007, the botnet took advantage of flaws in YouTube's captcha application on its mail systems, to send targeted spam e-mails to Xbox owners with a scam involving winning a special version of the video game *Halo 3*. Other attack methods include using appealing animated images of laughing cats to get people to click on a trojan software download, and tricking users of Yahoo!'s GeoCities service to download software that was claimed to be needed to use GeoCities itself. The GeoCities attack in particular was called a "full-fledged attack vector" by Paul Ferguson of Trend Micro, and implicated members of the Russian Business Network, a well-known spam and malware service. On Christmas Eve in 2007, the Storm botnet began sending out holiday-themed messages revolving around male interest in women, with such titles as "Find Some Christmas Tail", "The Twelve Girls of Christmas," and "Mrs. Claus Is Out Tonight!" and photos of attractive women. It was described as an attempt to draw more unprotected systems into the botnet and boost its size over the holidays, when security updates from protection vendors may take longer to be distributed. A day after the e-mails with Christmas strippers were distributed, the Storm botnet operators immediately began sending new infected e-mails that claimed to wish their recipients a "Happy New Year 2008!"

In January 2008, the botnet was detected for the first time to be involved in phishing attacks against major financial institutions, targeting both Barclays and Halifax.

Encryption and sales

Around October 15, 2007, it was uncovered that portions of the Storm botnet and its variants could be for sale. This is being done by using unique security keys in the encryption of the botnet's Internet traffic and information. The unique keys will allow each segment, or sub-section of the Storm botnet, to communicate with a section that has a matching security key. However, this may also allow people to detect, track, and block Storm botnet traffic in the future, if the security keys have unique lengths and signatures. Computer security vendor Sophos has agreed with the assessment that the partitioning of the Storm botnet indicated likely resale of its services. Graham Cluley of Sophos said, "Storm's use of encrypted traffic is an interesting feature which has raised eyebrows in our lab. Its most likely use is for the cybercriminals to lease out portions of the network for misuse. It wouldn't be a surprise if the network was used for spamming, distributed denial-of-service attacks, and other malicious activities." Security experts reported that if Storm is broken up for the malware market, in the form of a "ready-to-use botnet-making spam kit", the world could see a sharp rise in the number of Storm related infections and compromised computer systems. The encryption only seems to affect systems compromised by Storm from the second week of October 2007 onwards, meaning that any of the computer systems compromised before that time frame will remain difficult to track and block.

Within days of the discovery of this segmenting of the Storm botnet, spam e-mail from the new subsection was uncovered by major security vendors. In the evening of October 17, security vendors began seeing new spam with embedded MP3 sound files, which attempted to trick victims into investing in a penny stock, as part of an illegal pump-and-dump stock scam. It was believed that this was the first-ever spam e-mail scam that made use of audio to fool victims. Unlike nearly all other Storm-related e-mails, however, these new audio stock scam messages did not include any sort of virus or Storm malware payload; they simply were part of the stock scam.

In January 2008, the botnet was detected for the first time to be involved in phishing attacks against the customers of major financial institutions, targeting banking establishments in Europe including Barclays, Halifax and the Royal Bank of Scotland. The unique security keys used indicated to F-Secure that segments of the botnet were being leased.

Claimed decline of the botnet

On September 25, 2007, it was estimated that a Microsoft update to the Windows Malicious Software Removal Tool (MSRT) may have helped reduce the size of the botnet by up to 20%. The new patch, as claimed by Microsoft, removed Storm from approximately 274,372 infected systems out of 2.6 million scanned Windows systems. However, according to senior security staff at Microsoft, "the 180,000+ additional machines that have been cleaned by MSRT since the first day are likely to be home user machines that were not notably incorporated into the daily operation of the 'Storm' botnet," indicating that the MSRT cleaning may have been symbolic at best.

As of late October 2007, some reports indicated that the Storm botnet was losing the size of its Internet footprint, and was significantly reduced in size. Brandon Enright, a University of California at San Diego security analyst, estimated that the botnet had by late October fallen to a size of approximately 160,000 compromised systems, from Enright's previous estimated high in July 2007 of 1,500,000 systems. Enright noted, however, that the botnet's composition was constantly changing, and that it was still actively defending itself against attacks and observation. "If you're a researcher and you hit the pages hosting the malware too much… there is an automated process that automatically launches a denial of service [attack] against you," he said, and added that his research caused a Storm botnet attack that knocked part of the UC San Diego network offline.

The computer security company McAfee is reported as saying that the Storm Worm would be the basis of future attacks. Craig Schmugar, a noted security expert who discovered the Mydoom worm, called the Storm botnet a trend-setter, which has led to more usage of similar tactics by criminals. One such derivative botnet has been dubbed the "Celebrity Spam Gang", due to their use of similar technical tools as the Storm botnet controllers. Unlike the sophisticated social engineering that the Storm operators use to entice victims, however, the Celebrity spammers make use of offers of nude images of celebrities such as Angelina Jolie and Britney Spears. Cisco Systems security experts stated in a report that they believe the Storm botnet would remain a critical threat in 2008, and said they estimated that its size remained in the "millions".

As of early 2008, the Storm botnet also found business competition in its black hat economy, in the form of Nugache, another similar botnet which was first identified in 2006. Reports have indicated a price war may be underway between the operators of both botnets, for the sale of their spam E-mail delivery. Following the Christmas and New Year's holidays bridging 2007-2008, the researchers of the German Honeynet Project reported that the Storm botnet may have increased in size by up to 20% over the holidays. The *MessageLabs Intelligence* report dated March 2008 estimates that over 20% of all spam on the Internet originates from Storm.

Present state of the botnet

The Storm botnet was sending out spam for more than two years until its decline in late 2008. One factor in this — on account of making it less interesting for the creators to maintain the botnet — may have been the Stormfucker tool, which made it possible to take control over parts of the botnet.

Stormbot 2

On April 28, 2010, McAfee made an announcement that the so-called "rumors" of a Stormbot 2 were verified. Mark Schloesser, Tillmann Werner, and Felix Leder, the German researchers who did a lot of work in analyzing the original Storm, found that around two-thirds of the “new” functions are a copy and paste from the last Storm code

base. The only thing missing is the P2P infrastructure, perhaps because of the tool which used P2P to bring down the original Storm. Honeynet blog dubbed this Stormbot 2.

Chapter 13

Daemon (Computer Software)

In Unix and other computer multitasking operating systems, a **daemon** is a computer program that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes. Typically daemons have names that end with the letter "d": for example, `syslogd`, the daemon that handles the system log, or `sshd`, which handles incoming SSH connections.

In a Unix environment, the parent process of a daemon is often (but not always) the init process (PID=1). Processes usually become daemons by forking a child process and then having their parent process immediately exit, thus causing init to adopt the child process. This is a somewhat simplified view of the process as other operations are generally performed, such as dissociating the daemon process from any controlling tty. Convenience routines such as daemon(3) exist in some UNIX systems for that purpose.

Systems often start (or "launch") daemons at boot time: they often serve the function of responding to network requests, hardware activity, or other programs by performing some task. Daemons can also configure hardware (like udevd on some GNU/Linux systems), run scheduled tasks (like cron), and perform a variety of other tasks.

Terminology

The term was coined by the programmers of MIT's Project MAC. They took the name from Maxwell's demon, an imaginary being from a famous thought experiment that constantly works in the background, sorting molecules. Unix systems inherited this terminology. A derivation from the phrase "Disk and Execution Monitor" is a backronym. Daemons are also characters in Greek mythology, some of whom handled tasks that the gods could not be bothered with. BSD and some of its derivatives have adopted a daemon as its mascot, although this mascot is actually a cute variation of the demons which appear in Christian artwork.

Types of daemons

In a strictly technical sense, a Unix-like system process is a daemon when its parent process terminates and is therefore 'adopted' by the init process (process number 1) as its parent process and has no controlling terminal. However, more commonly, a daemon may be any background process, whether a child of `init` or not.

The common method for a process to become a daemon involves:

- Dissociating from the controlling tty
- Becoming a session leader
- Becoming a process group leader
- Staying in the background by forking and exiting (once or twice). This is required sometimes for the process to become a session leader. It also allows the parent process to continue its normal execution. This idiom is sometimes summarized with the phrase "fork off and die"
- Setting the root directory ("/") as the current working directory so that the process will not keep any directory in use that may be on a mounted file system (allowing it to be unmounted).
- Changing the umask to 0 to allow open(), creat(), et al. calls to provide their own permission masks and not to depend on the umask of the caller
- Closing all inherited open files at the time of execution that are left open by the parent process, including file descriptors 0, 1 and 2 (stdin, stdout, stderr). Required files will be opened later.
- Using a logfile, the console, or /dev/null as stdin, stdout, and stderr

List of service daemons for Unix and Unix-like systems

- amd - Auto Mount Daemon
- anacron - Executed delayed cron tasks at boot time
- apmd - Advanced Power Management Daemon
- arpwatch - watches for Ethernet IP address pairings that are resolved using the ARP protocol
- atd - Runs jobs queued using the at tool
- bincimapd - An imap server daemon
- biod - Cooperates with a remote nfsd to handle client Network file system requests
- bootparmd - A Internet Bootstrap Protocol server daemon
- chttpd - An http server daemon
- configd - A daemon that maintains dynamic configuration information about the computer and its environment (mainly the network)
- crond - The task scheduler daemon
- cupsd - CUPS printer daemon
- devfsd -
- dhcpd - Dynamic Host Configuration Protocol and Internet Bootstrap Protocol Server

- drakfont - A font server daemon used in Mandrake Linux
- dpid - A plugin handler for the dillo internet web browser
- egpup -
- fetchmail - daemon to retrieve mail from servers at regular intervals
- fingerd - Provides a finger protocol server
- ftpd - FTP Server Daemon
- gated - routing daemon that handles multiple routing protocols and replaces routed and egpup
- gpm - General Purpose Mouse Daemon
- httpd - Web Server Daemon
- identd - Provides the identity of a user of a particular TCP connection
- idmapd - Translates user and group identities to names, and vice versa (for NFS4)
- inetd - Internet Superserver Daemon
- initd - Initial process Daemon
- imapd - An imap server daemon
- innd - A Usenet News Server Daemon
- ipchains - A deprecated packet forwarding / firewall daemon
- isdn - ISDN network interfacing server daemon
- kapmd - Advanced Power Management Daemon
- kblockd
- kerneld - Automatically loads and unloads kernel modules
- keventd
- keytable - Loads the appropriate keyboard map from the /etc/sysconfig/keyboard configuration file
- kheader -
- klogd -
- ksoftirqd -
- kswapd - Kernel Swap daemon
- kswapd0 -
- kudzu - Detects and configures new or changed hardware during boot
- kupdated -
- launchd - PID 1 on Mac OS X systems
- linuxconf - Startup hook for the linuxconf configuration system
- lockd
- lpd - Line Printer Daemon
- mathoptd - A small single process httpd daemon with CGI support
- mcserv - Server for the midnight command networking filesystem
- memcached - In-memory distributed object caching daemon
- mpd - Music Player Daemon
- micro httpd -
- mountd - mount daemon
- mysql - Database server daemon
- named - A DNS server daemon
- netfs - Network Filesystem Mounter
- network - Activates all network interfaces at boot time
- nfsd - Network File Sharing Daemon

- nfslock - Used to start and stop nfs file locking services
- nmbd - Network Message Block Daemon
- ntpd - Network Time Protocol service daemon
- numlock - This daemon locks the numlock key during a runlevel change
- pcmcia - Provides generic pcmcia services
- portmap -
- postfix - A mail transport agent used as a replacement for sendmail
- postgresql - Database server daemon
- random - Random number generating daemon
- retchmail - A mail retrieval agent
- rlprd - Remote line printer proxy daemon
- routed - Manages routing tables
- rpcbind - Remote Procedure Call Bind Daemon
- rpciod - Remote Procedure Call Daemon
- rquotad
- rstatd - kernel statistics server daemon
- rusersd - Allows users to find one another over the network
- rwalld - Allows users to write messages on remote terminals using rwall
- rwhod - maintains the database used by the rwho and ruptime tools
- sched - Copies process regions to swap space to reclaim physical pages of memory
- sendmail - A mail transfer agent Daemon
- smbd - Samba (an SMB Server) Daemon
- smtpd - Simple Mail Transfer Protocol Daemon
- snmpd - Simple Network Management Protocol Daemon
- sound - A sound server daemon
- squid - A web page caching proxy server daemon
- sshd - Secure Shell Server Daemon
- statd -
- swapper - Copies process regions to swap space to reclaim physical pages of memory
- syncd - Keeps the file systems synchronized with system memory
- syslogd - System logging daemon
- tcpd - Service wrapper restricts access to inetd based services through hosts.allow and hosts.deny
- telnetd - Telnet Server daemon
- usb - Manages devices attached to the universal serial bus
- uptimed - Uptime logging daemon
- utelnetd - A telnet server daemon
- vhand - The "page stealing daemon" releases pages of memory for use by other processes
- vsftpd - Very Secure FTP Daemon
- walld -
- webmin - Web based administration server daemon
- xfsd - X font server daemon
- xinetd - Enhanced Internet Superserver Daemon

- xntd - Network Time Server Daemon
- ypbind - A bind server for Network Information Services

Windows equivalent

In the Microsoft DOS environment, such programs were written as Terminate and Stay Resident (TSR) software. On Microsoft Windows NT systems, programs called *services* perform the functions of daemons. They run as processes, usually do not interact with the monitor, keyboard, and mouse, and may be launched by the operating system at boot time. With Windows 2000 and later versions, one can configure and manually start and stop Windows services using the Control Panel → Administrative Tools or typing "Services.msc" in the Run command on Start menu.

However, any Windows application can perform the role of a daemon, not just a service, and some daemons for Windows have the option of running as a normal process,, where it still has no visual output.

Mac OS equivalent

On the original Mac OS, optional features and services were provided by files loaded at startup time that patched the operating system; these were known as system extensions and control panels. Later versions of classic Mac OS augmented these with fully-fledged faceless background applications: regular applications that ran in the background. To the user, these were still described as regular system extensions.

Mac OS X, being a Unix system, has daemons. There is a category of software called *services* as well, but these are different in concept from Windows' services.

Etymology

In the general sense, daemon is an older form of the word demon, from the Greek δαίμων. In the Unix System Administration Handbook, Evi Nemeth states the following about daemons:

Many people equate the word "daemon" with the word "demon", implying some kind of satanic connection between UNIX and the underworld. This is an egregious misunderstanding. "Daemon" is actually a much older form of "demon"; daemons have no particular bias towards good or evil, but rather serve to help define a person's character or personality. The ancient Greeks' concept of a "personal daemon" was similar to the modern concept of a "guardian angel" — *eudaemonia* is the state of being helped or protected by a kindly spirit. As a rule, UNIX systems seem to be infested with both daemons and demons. (p.403)

A further characterization of the mythological symbolism is that a daemon is something which is not visible yet is always present and working its will. Plato's Socrates describes his own personal daemon to be something like the modern concept of a moral conscience:

> *The favour of the gods has given me a marvelous gift, which has never left me since my childhood. It is a voice which, when it makes itself heard, deters me from what I am about to do and never urges me on.*
>
> —Character of Socrates in *Theages*, Plato

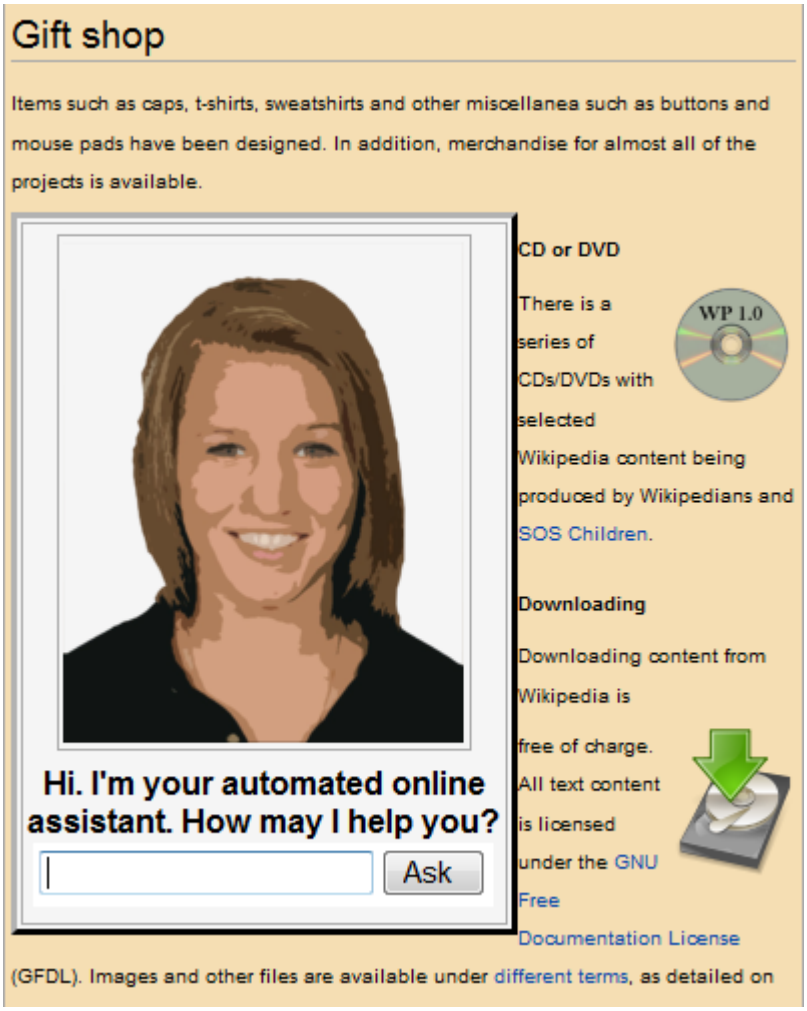
Chapter 14

Internet Bot

Internet bots, also known as **web robots**, **WWW robots** or simply **bots**, are software applications that run automated tasks over the Internet. Typically, bots perform tasks that are both simple and structurally repetitive, at a much higher rate than would be possible for a human alone. The largest use of bots is in web spidering, in which an automated script fetches, analyzes and files information from web servers at many times the speed of a human. Each server can have a file called `robots.txt`, containing rules for the spidering of that server that the bot is supposed to obey.

In addition to their uses outlined above, bots may also be implemented where a response speed faster than that of humans is required (*e.g.*, gaming bots and auction-site robots) or less commonly in situations where the emulation of human activity is required, for example chat bots. Recently bots have been used for search advertising, such as Google Adsense.

Commercial purposes



Example of an automated online assistant, where chatterbots are major components.

Chatterbots are used in automated online assistants by organizations as a way of interacting with consumers and users of services. This can avail for enterprises to reduce their operating and training cost. A major underlying technology to such systems is natural language processing.

There has been a great deal of controversy about the use of bots in an automated trading function. Auction website eBay has been to court in an attempt to suppress a third-party company from using bots to traverse their site looking for bargains; this approach backfired on eBay and attracted the attention of further bots.

Charitable purposes

Bots have also been known to fast-track the purposes of charities, one of which is FreeRice.

On FreeRice

Since FreeRice became well-known through Digg.com and other news sources, many programming-adept users created scripts to automatically play the game for them. The scripts operate far faster than humans alone and run for 24 hours a day. At first, the scripts got only ≈1/4 of the words correct by random chance. Eventually, these bots were adapted with automated online dictionary search, dictionary files, and word database dumps so the programs can choose the correct answers the first time more often. The word database dumps were created so when the incorrect answer was chosen, the bots would record the correct answer the next page would show. Thus, the bot would choose the correct answer whenever it happened upon the same words later. Due to the growing number of scripts used on FreeRice, the number of rice donated has remarkably risen. Currently there are no rules governing "ricebots", as they are called. Until those rules are formed, anyone is free to program and use the scripts. With a delay of about 3 seconds between iterations, it is estimated that a script can feed about 8 people per day, if running 24/7. The idea was taken even further to create a multi-threaded bot which can run fifty or more browser instances at a time, enough to liberate as much as 600,000 grains of rice per hour or to feed 720 people per day. One script with 1,000 threads was able to donate over 3,000,000 grains in just a few hours.

Donated rice comes from the advertisements from sponsors, therefore abuse of scripts will likely lead to catastrophe, as advertisers prefer that actual people view their advertisements. Knowing the existence of the bots, FreeRice updated their FAQ explaining the potential damage of botting. Some bots have made changes to make sure they won't spoil the FreeRice spirit.

Malicious purposes

Another, more malicious use of bots is the coordination and operation of an automated attack on networked computers, such as a denial-of-service attack by a botnet. Internet bots can also be used to commit click fraud and more recently have seen usage around MMORPG games as computer game bots. A spambot is an internet bot that attempts to spam large amounts of content on the Internet, usually adding advertising links.

- There are malicious bots (and botnets) of the following types:

1. Spambots that harvest email addresses from internet forums, contact forms or guestbook pages
2. Downloader programs that suck bandwidth by downloading entire web sites
3. Web site scrapers that grab the content of web sites and re-use it without permission on automatically generated doorway pages
4. Viruses and worms
5. DDoS attacks
6. Botnets / zombie computers; etc.
7. File-name modifiers on peer-to-peer file-sharing networks. These change the names of files (often containing malware) to match user search queries.

8. Automating the entry of internet sweepstakes or instant win games to get an advantage
9. Automating tasks on promotional web sites to win prizes
10. votebots which automatically cast votes for or againsts certain forms of user-contributed content such as videos on youtube or reader comments on blog pages.

- Bots are also used to buy up good seats for concerts, particularly by ticket brokers who resell the tickets. Bots are employed against entertainment event-ticketing sites, like TicketMaster.com. The bots are used by ticket brokers to unfairly obtain the best seats for themselves while depriving the general public from also having a chance to obtain the good seats. The bot runs through the purchase process and obtains better seats by pulling as many seats back as it can.
- Bots are often used in massively multiplayer online role-playing games (MMORPG) to farm for resources that would otherwise take significant time or effort to obtain; this is a concern for most online in-game economies.

The most widely used anti-bot technique is the use of CAPTCHA, which is a type of Turing test used to distinguish between a human user and a less-sophisticated AI-powered bot, by the use of graphically encoded human-readable text.

Chapter 15

Cognitive Architecture

A **cognitive architecture** is a blueprint for intelligent agents. It proposes (artificial) computational processes that act like certain cognitive systems, most often, like a person, or acts intelligent under some definition. Cognitive architectures form a subset of general agent architectures. The term 'architecture' implies an approach that attempts to model not only behavior, but also structural properties of the modelled system. These need not be physical properties: they can be properties of virtual machines implemented in physical machines (e.g. brains or computers).

Characterization

Common among researchers on cognitive architectures is the belief that understanding (human, animal or machine) cognitive processes means being able to implement them in a working system, though opinions differ as to what form such a system can have: some researchers assume that it will necessarily be a symbolic computational system whereas others argue for alternative models such as connectionist systems or dynamical systems. Cognitive architectures can be characterized by certain properties or goals, as follows, though there is not general agreement on all aspects:

1. Implementation of not just various different aspects of cognitive behavior but of cognition as a whole (Holism, e.g. Unified theory of cognition). This is in contrast to cognitive models, which focus on a particular competence, such as a kind of problem solving or a kind of learning.
2. The architecture often tries to reproduce the behavior of the modelled system (human), in a way that timely behavior (reaction times) of the architecture and modelled cognitive systems can be compared in detail. Other cognitive limitations are often modeled as well, e.g. limited working memory, attention or issues due to cognitive load.

3. Robust behavior in the face of error, the unexpected, and the unknown.
4. Learning (not for all cognitive architectures)
5. Parameter-free: The system does not depend on parameter tuning (in contrast to Artificial neural networks) (not for all cognitive architectures)
6. Some early theories such as SOAR and ACT-R originally focused only on the 'internal' information processing of an intelligent agent, including tasks like reasoning, planning, solving problems, learning concepts. More recently many architectures (including SOAR, ACT-R, PreAct, ICARUS, CLARION), FORR have expanded to include perception, action and also affective states and processes including motivation, attitudes, and emotions.
7. On some theories the architecture may be composed of different kinds of sub-architectures (often described as 'layers' or 'levels') where the layers may be distinguished by types of function, types of mechanism and representation used, types of information manipulated, or possibly evolutionary origin. These are hybrid architectures (e.g., CLARION).
8. Some theories allow different architectural components to be active concurrently, whereas others assume a switching mechanism that selects one component or module at a time, depending on the current task. Concurrency is normally required for an architecture for an animal or robot that has multiple sensors and effectors in a complex and dynamic environment, but not in all robotic paradigms.
9. Most theories assume that an architecture is fixed and only the information stored in various subsystems can change over time (e.g. Langley et al., below), whereas others allow architectures to grow, e.g. by acquiring new subsystems or new links between subsystems (e.g. Minsky and Sloman, below).

It is important to note that cognitive architectures don't have to follow a top-down approach to cognition (cf. Top-down and bottom-up design).

Distinctions

Cognitive architectures can be symbolic, connectionist, or hybrid. Some cognitive architectures or models are based on a set of generic rules, as, e.g., the Information Processing Language (e.g., Soar based on the unified theory of cognition, or similarly ACT). Many of these architectures are based on the-mind-is-like-a-computer analogy. In contrast subsymbolic processing specifies no such rules a priori and relies on emergent properties of processing units (e.g. nodes). Hybrid architectures combine both types of processing (such as CLARION). A further distinction is whether the architecture is centralized with a neural correlate of a processor at its core, or decentralized (distributed). The decentralized flavor, has become popular under the name of parallel distributed processing in mid-1980s and connectionism, a prime example being neural networks. A further design issue is additionally a decision between holistic and atomistic, or (more concrete) modular structure. By analogy, this extends to issues of knowledge representation.

In traditional AI, intelligence is often programmed from above: the programmer is the creator, and makes something and imbues it with its intelligence, though many traditional

AI systems were also designed to learn (e.g. improving their game-playing or problem-solving competence). Biologically inspired computing, on the other hand, takes sometimes a more bottom-up, decentralised approach; bio-inspired techniques often involve the method of specifying a set of simple generic rules or a set of simple nodes, from the interaction of which emerges the overall behavior. It is hoped to build up complexity until the end result is something markedly complex. However, it is also arguable that systems designed top-down on the basis of observations of what humans and other animals can do rather than on observations of brain mechanisms, are also biologically inspired, though in a different way.

Some well-known cognitive architectures

- 4CAPS, developed at Carnegie Mellon University under Marcel A. Just
- ACT-R, developed at Carnegie Mellon University under John R. Anderson.
- PreAct, developed under Dr. Norm Geddes at ASI.
- Apex developed under Michael Freed at NASA Ames Research Center.
- CHREST, developed under Fernand Gobet at Brunel University and Peter C. Lane at the University of Hertfordshire.
- CLARION the cognitive architecture, developed under Ron Sun at Rensselaer Polytechnic Institute and University of Missouri.
- CoJACK An ACT-R inspired extension to the JACK multi-agent system that adds a cognitive architecture to the agents for eliciting more realistic (human-like) behaviors in virtual environments.
- Copycat, by Douglas Hofstadter and Melanie Mitchell at the Indiana University.
- DUAL, developed at the New Bulgarian University under Boicho Kokinov.
- EPIC, developed under David E. Kieras and David E. Meyer at the University of Michigan.
- The H-Cogaff architecture, which is a special case of the CogAff schema.
- FORR developed by Susan L. Epstein at The City University of New York.
- IDA and LIDA, developed under Stan Franklin at the University of Memphis.
- PRODIGY, by Veloso et al.
- PRS 'Procedural Reasoning System', developed by Michael Georgeff and Amy Lansky at SRI International.
- Psi-Theory developed under Dietrich Dörner at the Otto-Friedrich University in Bamberg, Germany.
- R-CAST, developed at the Pennsylvania State University.
- Soar, developed under Allen Newell and John Laird at Carnegie Mellon University and the University of Michigan.
- Society of mind and its successor the Emotion machine proposed by Marvin Minsky.
- Subsumption architectures, developed e.g. by Rodney Brooks (though it could be argued whether they are *cognitive*).

Chapter 16

Evolutionary Computation

In computer science, **evolutionary computation** is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems.

Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution.

As evolution can produce highly optimised processes and networks, it has many applications in computer science. Here, simulations of evolution using evolutionary algorithms and artificial life started with the work of Nils Aall Barricelli in the 1960s, and was extended by Alex Fraser, who published a series of papers on simulation of artificial selection. Artificial evolution became a widely recognised optimisation method as a result of the work of Ingo Rechenberg in the 1960s and early 1970s, who used evolution strategies to solve complex engineering problems. Genetic algorithms in particular became popular through the writing of John Holland. As academic interest grew, dramatic increases in the power of computers allowed practical applications, including the automatic evolution of computer programs. Evolutionary algorithms are now used to solve multi-dimensional problems more efficiently than software produced by human designers, and also to optimise the design of systems.

History

The use of Darwinian principles for automated problem solving originated in the fifties. It was not until the sixties that three distinct interpretations of this idea started to be developed in three different places.

Evolutionary programming was introduced by Lawrence J. Fogel in the USA, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies. These areas developed separately for about 15 years. From the early nineties on they are unified as different representatives ("dialects") of one technology, called evolutionary computing. Also in the early nineties, a fourth stream following the general ideas had emerged – genetic programming.

These terminologies denote the field of evolutionary computing and consider evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas.

Techniques

Evolutionary computing techniques mostly involve metaheuristic optimization algorithms. Broadly speaking, the field includes:

Evolutionary algorithms

- Genetic algorithms
- Evolutionary programming
- Evolution strategy
- Genetic programming

Swarm intelligence

- Ant colony optimization
- Particle swarm optimization

and in a lesser extent also:

- Artificial life
- Artificial immune systems
- Cultural algorithms
- Differential evolution
- Harmony search
- Learning classifier systems
- Learnable Evolution Model
- Self-organization such as self-organizing maps, competitive learning

Evolutionary algorithms

Evolutionary algorithms form a subset of evolutionary computation in that they generally only involve techniques implementing mechanisms inspired by biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within which the solutions

"live". Evolution of the population then takes place after the repeated application of the above operators.

In this process, there are two main forces that form the basis of evolutionary systems: **Recombination** and **mutation** create the necessary diversity and thereby facilitate novelty, while **selection** acts as a force increasing quality.

Many aspects of such an evolutionary process are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen. On the other hand, selection operators can be either deterministic, or stochastic. In the latter case, individuals with a higher fitness have a higher chance to be selected than individuals with a lower fitness, but typically even the weak individuals have a chance to become a parent or to survive.

Evolutionary computation practitioners

- Kalyanmoy Deb
- David E. Goldberg
- John Henry Holland
- John Koza
- Peter Nordin
- Ingo Rechenberg
- Hans-Paul Schwefel

Chapter 17

Self-Reconfiguring Modular Robot

Modular self-reconfiguring robotic systems or **self-reconfigurable modular robots** are autonomous kinematic machines with variable morphology. Beyond conventional actuation, sensing and control typically found in fixed-morphology robots, self-reconfiguring robots are also able to deliberately change their own shape by rearranging the connectivity of their parts, in order to adapt to new circumstances, perform new tasks, or recover from damage.

For example, a robot made of such components could assume a worm-like shape to move through a narrow pipe, reassemble into something with spider-like legs to cross uneven terrain, then form a third arbitrary object (like a ball or wheel that can spin itself) to move quickly over a fairly flat terrain; it can also be used for making "fixed" objects, such as walls, shelters, or buildings.

In some cases this involves each module having 2 or more connectors for connecting several together. They can contain electronics, sensors, computer processors, memory, and power supplies; they can also contain actuators that are used for manipulating their location in the environment and in relation with each other. A feature found in some cases is the ability of the modules to automatically connect and disconnect themselves to and from each other, and to form into many objects or perform many tasks moving or manipulating the environment.

By saying "self-reconfiguring" or "self-reconfigurable" it means that the mechanism or device is capable of utilizing its own system of control such as with actuators or stochastic means to change its overall structural shape. Having the quality of being "modular" in "self-reconfiguring modular robotics" is to say that the same module or set of modules can be added or removed to the system, as opposed to being generically "modularized" in the broader sense. The underlying intent is to have an indefinite number of identical modules, or a finite and relatively small set of identical modules, in a mesh or matrix structure of self-reconfigurable modules.

Self-reconfiguration is also different from the concept of self-replication, and self-replication is not necessarily a quality that a self-reconfigurable module or collection of such modules can or must possess. A matrix of N-number of modules does not need to be able to increase the quantity of modules to greater than N to be considered self-reconfigurable. It is sufficient for self-reconfigurable modules to be a device that is produced at a conventional factory, where dedicated machines stamp or mold components, and factory workers on an assembly line assemble the components to build each module.

There are two basic types of methods of segment articulation that self-reconfigurable mechanisms can utilize to reshape their structures, chain reconfiguration and lattice reconfiguration.

Structure and control

Modular robots are usually composed of multiple building blocks of a relatively small repertoire, with uniform docking interfaces that allow transfer of mechanical forces and moments, electrical power and communication throughout the robot.

The modular building blocks usually consist of some primary structural actuated unit, and potentially additional specialized units such as grippers, feet, wheels, cameras, payload and energy storage and generation.

A taxonomy of architectures

Modular self-reconfiguring robotic systems can be generally classified into several architectural groups by the geometric arrangement of their unit (lattice vs. chain). Several systems exhibit hybrid properties.

- **Lattice architectures** have units that are arranged and connected in some regular, space-filling three-dimensional pattern, such as a cubical or hexagonal grid. Control and motion are executed in parallel. Lattice architectures usually offer simpler computational representation that can be more easily scaled to complex systems.

- **Chain/tree architectures** have units that are connected together in a string or tree topology. This chain or tree can fold up to become space filling, but underlying architecture is serial. Chain architectures can reach any point in space, and are therefore more versatile but more computationally difficult to represent and analyze. Tree architectures may resemble a bush robot

Modular robotic systems can also be classified according to the way by which units are reconfigured (moved) into place.

- **Deterministic reconfiguration** relies on units moving or being directly manipulated into their target location during reconfiguration. The exact location

of each unit is known at all times. Reconfiguration times can be guaranteed, but sophisticated feedback control is necessary to assure precise manipulation. Macro-scale systems are usually deterministic.

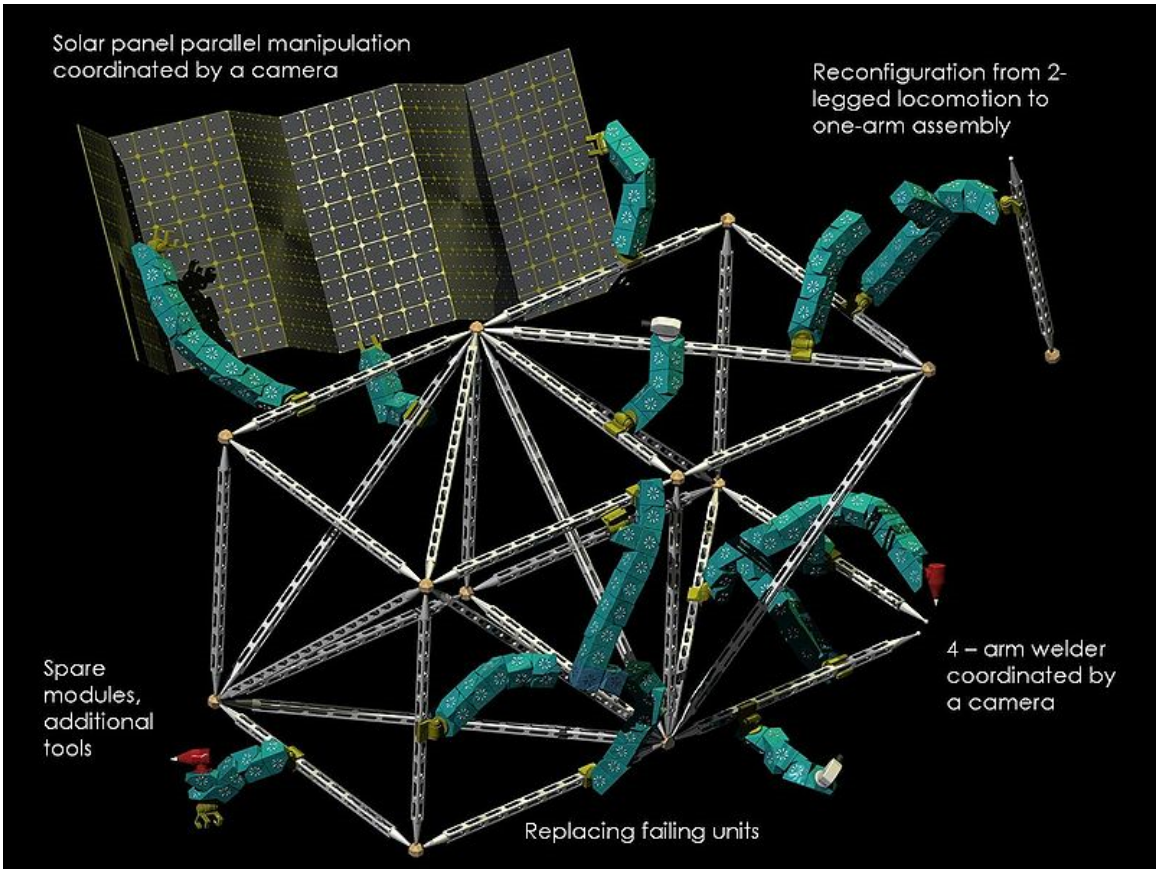
- **Stochastic reconfiguration** relies on units moving around using statistical processes (like Brownian motion). The exact location of each unit only known when it is connected to the main structure, but it may take unknown paths to move between locations. Reconfiguration times can be guaranteed only statistically. Stochastic architectures are more favorable at micro scales.

Other modular robotic systems exist which are not self-reconfigurable, and thus do not formally belong to this family of robots though they may have similar appearance. For example, self-assembling systems may be composed of multiple modules but cannot dynamically control their target shape. Similarly, tensegrity robotics may be composed of multiple interchangeable modules but cannot self-reconfigure.

Motivation and inspiration

There are two key motivations for designing modular self-reconfiguring robotic systems.

- **Functional advantage**: Self reconfiguring robotic systems are potentially more **robust** and more **adaptive** than conventional systems. The reconfiguration ability allows a robot or a group of robots to disassemble and reassemble machines to form new morphologies that are better suitable for new tasks, such as changing from a legged robot to a snake robot and then to a rolling robot. Since robot parts are interchangeable (within a robot and between different robots), machines can also replace faulty parts autonomously, leading to self-repair.



Autonomous modular robotics in space

- **Economic advantage**: Self reconfiguring robotic systems can potentially lower overall robot cost by making a range of complex machines out of a single (or relatively few) types of mass-produced modules.

Both these advantages have not yet been fully realized. A modular robot is likely to be inferior in performance to any single custom robot tailored for a specific task. However, the advantage of modular robotics is only apparent when considering multiple tasks that would normally require a set of different robots.

The added degrees of freedom make modular robots more versatile in their potential capabilities, but also incur a performance tradeoff and increased mechanical and computational complexities.

The quest for self-reconfiguring robotic structures is to some extent inspired by envisioned applications such as long-term space missions, that require long-term self-sustaining robotic ecology that can handle unforeseen situations and may require self repair. A second source of inspiration are biological systems that are self-constructed out of a relatively small repertoire of lower-level building blocks (cells or amino acids, depending on scale of interest). This architecture underlies biological systems' ability to

physically adapt, grow, heal, and even self replicate – capabilities that would be desirable in many engineered systems.

Application areas

Given these advantages, where would a modular self-reconfigurable system be used? While the system has the promise of being capable of doing a wide variety of things, finding the "killer application" has been somewhat elusive. Here are several examples:

Space exploration

One application that highlights the advantages of self-reconfigurable systems is long-term space missions. These require long-term self-sustaining robotic ecology that can handle unforeseen situations and may require self repair. Self-reconfigurable systems have the ability to handle tasks that are not known a priori especially compared to fixed configuration systems. In addition, space missions are highly volume and mass constrained. Sending a robot system that can reconfigure to achieve many tasks is better than sending many robots that each can do one task.

Telepario

Another example of an application has been coined "telepario" by CMU professors Todd Mowry and Seth Goldstein. What the researchers propose to make are moving, physical, three-dimensional replicas of people or objects, so lifelike that human senses would accept them as real. This would eliminate the need for cumbersome virtual reality gear and overcome the viewing angle limitations of modern 3D approaches. The replicas would mimic the shape and appearance of a person or object being imaged in real time, and as the originals moved, so would their replicas. One aspect of this application is that the main development thrust is geometric representation rather than applying forces to the environment as in a typical robotic manipulation task. This project is widely known as claytronics or programmable matter.

Bucket of stuff

A third long term vision for these systems has been called "bucket of stuff". In this vision, consumers of the future have a container of self-reconfigurable modules say in their garage, basement, or attic. When the need arises, the consumer calls forth the robots to achieve a task such as "clean the gutters" or "change the oil in the car" and the robot assumes the shape needed and does the task. One source of inspiration for the development of these systems comes from the application. A second source is biological systems that are self-constructed out of a relatively small repertoire of lower-level building blocks (cells or amino acids, depending on scale of interest). This architecture underlies biological systems' ability to physically adapt, grow, heal, and even self replicate – capabilities that would be desirable in many engineered systems.

History and state of the art

The roots of the concept of modular self-reconfigurable robots can be traced back to the "quick change" end effector and automatic tool changers in computer numerical controlled machining centers in the 1970s. Here, special modules each with a common connection mechanism could be automatically swapped out on the end of a robotic arm. However, taking the basic concept of the common connection mechanism and applying it to the whole robot was introduced by Toshio Fukuda with the CEBOT (short for cellular robot) in the late 1980s.

The early 1990s saw further development from Greg Chirikjian, Mark Yim, Joseph Michael, and Satoshi Murata. Chirikjian, Michael, and Murata developed lattice reconfiguration systems and Yim developed a chain based system. While these researchers started with from a mechanical engineering emphasis, designing and building modules then developing code to program them, the work of Daniela Rus and Wei-min Shen developed hardware but had a greater impact on the programming aspects. They started a trend towards provable or verifiable distributed algorithms for the control of large numbers of modules.

One of the more interesting hardware platforms recently has been the MTRAN II and III systems developed by Satoshi Murata et al. This system is a hybrid chain and lattice system. It has the advantage of being able to achieve tasks more easily like chain systems, yet reconfigure like a lattice system.

More recently new efforts in stochastic self-assembly have been pursued by Hod Lipson and Eric Klavins. A large effort at CMU headed by Seth Goldstein and Todd Mowry has started looking at issues in developing millions of modules.

Many tasks have been shown to be achievable, especially with chain reconfiguration modules. This demonstrates the versatility of these systems however, the other two advantages, robustness and low cost have not been demonstrated. In general the prototype systems developed in the labs have been fragile and expensive as would be expected during any initial development.

There is a growing number of research groups actively involved in modular robotics research. To date, about 30 systems have been designed and constructed, some of which are shown below.

Physical systems created			
System	**Class, DOF**	**Author**	**Year**
CEBOT	Mobile	Fukuda et al. (Tsukuba)	1988
Polypod	chain, 2, 3D	Yim (Stanford)	1993
Metamorphic	lattice, 6, 2D	Chirikjian (Caltech)	1993
Fracta	lattice, 3 2D	Murata (MEL)	1994

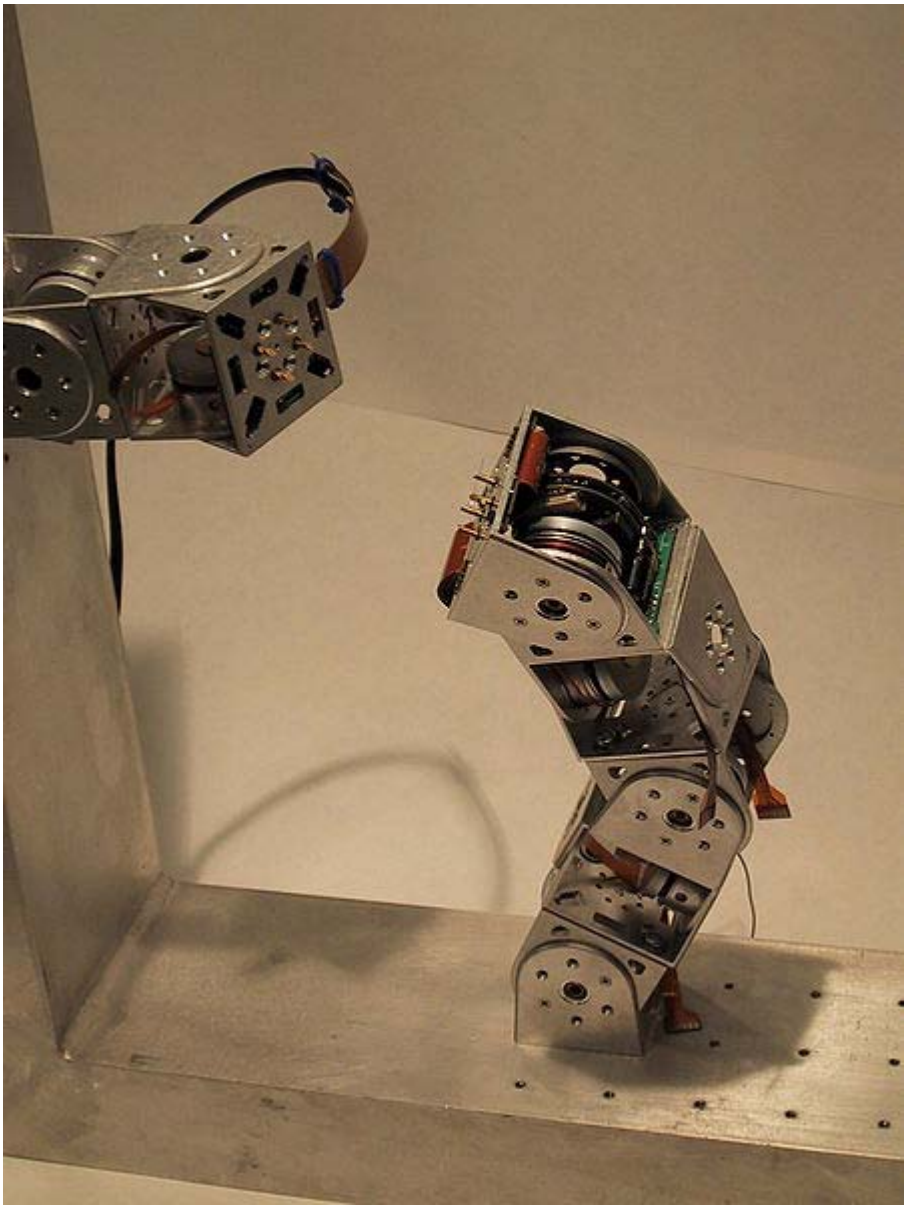
Fractal Robots	lattice, 3D	Michael(UK)	1995
Tetrobot	chain, 1 3D	Hamline et al. (RPI)	1996
ANAT Robot	Chain/tree, 8D	Charles Khairallah (CA)	1997
3D Fracta	lattice, 6 3D	Murata et al. (MEL)	1998
Molecule	lattice, 4 3D	Kotay & Rus (Dartmouth)	1998
CONRO	chain, 2 3D	Will & Shen (USC/ISI)	1998
PolyBot	chain, 1 3D	Yim et al. (PARC)	1998
TeleCube	lattice, 6 3D	Suh et al., (PARC)	1998
Vertical	lattice, 2D	Hosakawa et al., (Riken)	1998
Crystalline	lattice, 4 2D	Vona & Rus, (Dartmouth)	1999
I-Cube	lattice, 3D	Unsal, (CMU)	1999
M-TRAN I	hybrid, 2 3D	Murata et al.(AIST)	1999
Pneumatic	lattice, 2D	Inou et al., (TiTech)	2002
Uni Rover	mobile, 2 2D	Hirose et al., (TiTech)	2002
M-TRAN II	hybrid, 2 3D	Murata et al., (AIST)	2002
Atron	lattice, 1 3D	Stoy et al., (U.S Denmark)	2003
S-bot	mobile, 3 2D	Mondada et al., (EPFL)	2003
Stochastic	lattice, 0 3D	White, Kopanski, Lipson (Cornell)	2004
Superbot	hybrid, 3 3D	Shen et al., (USC/ISI)	2004
Y1 Modules	Chain, 1 3D	Gonzalez-Gomez et al., (UAM)	2004
M-TRAN III	hybrid, 2 3D	Kurokawa et al., (AIST)	2005
AMOEBA-I	Mobile, 7 3D	Liu JG et al., (SIA)	2005
Catom	lattice, 0 2D	Goldstein et al., (CMU)	2005
Stochastic-3D	lattice, 0 3D	White, Zykov, Lipson (Cornell)	2005
Molecubes	chain, 1 3D	Zykov, Mytilinaios, Lipson (Cornell)	2005
Prog. parts	lattice, 0 2D	Klavins, (U. Washington)	2005
Miche	lattice, 0 3D	Rus et al., (MIT)	2006
GZ-I Modules	Chain, 1 3D	Zhang & Gonzalez-Gomez (U. Hamburg, UAM)	2006
Evolve	Chain, 2 3D	Chang Fanxi, Francis (NUS)	2008
Odin	Hybrid, 3 3D	Lyder et al., Modular Robotics Research Lab, (USD)	2008
Roombots	Hybrid, 3 3D	Sproewitz, Moeckel, Ijspeert, Biorobotics Laboratory, (EPFL)	2009

Some current systems

ANAT Robot (1997)

ANAT Walker

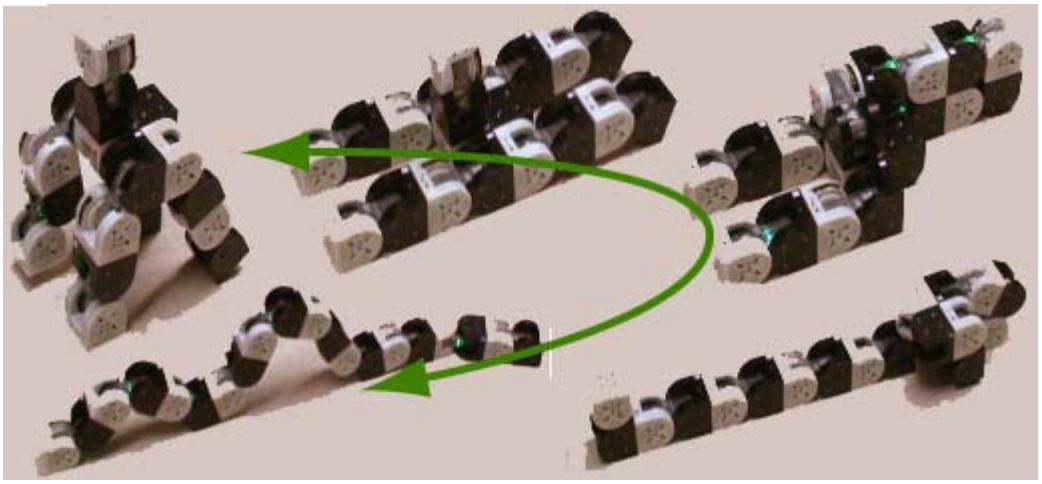
A chain/tree hyper-redundant modular robotic system invented by Charles Khairallah from Robotics Design Inc. in Montreal, Quebec, Canada. This robot is designed with ANAT Technology and is currently used for industrial manipulating under the name ANAT AMI-100, and Robotics Design's patented U and H shaped modules, of which this robot is composed, form the other robots in the ANAT robotics family. This robot can re-configure and/or self-reconfigure to form different shapes, due to its LEGO-like sets of modules with 1 degree of freedom each. Configurations range from mobile robots (ANATROLLER), manipulators (ANAT AMI-100) to walking robots (ANAT Walker).



PolyBot G3 (2002)

A chain self-reconfiguration system. Each module is about 50 mm on a side, and has 1 rotational DOF. It is part of the PolyBot modular robot family that has demonstrated many modes of locomotion including walking: biped, 14 legged, slinky-like, snake-like: concertina in a gopher hole, inchworm gaits, rectilinear undulation and sidewinding gaits, rolling like a tread at up to 1.4m/s, riding a tricycle, climbing: stairs, poles pipes, ramps etc. More information can be found at the polybot webpage at PARC.

M-TRAN III (2005)



M-TRAN III

A hybrid type self-reconfigurable system. Each module is two cube size (65 mm side), and has 2 rotational DOF and 6 flat surfaces for connection. It is the 3rd M-TRAN prototypes. Compared with the former (M-TRAN II), speed and reliability of connection is largely improved. As a chain type system, locomotion by CPG (Central Pattern Generator) controller in vaious shapes has been demonstrated by M-TRAN II. As a lattice type system, it can change its configuration, e.g., between a 4 legged walker to a caterpillar like robot.

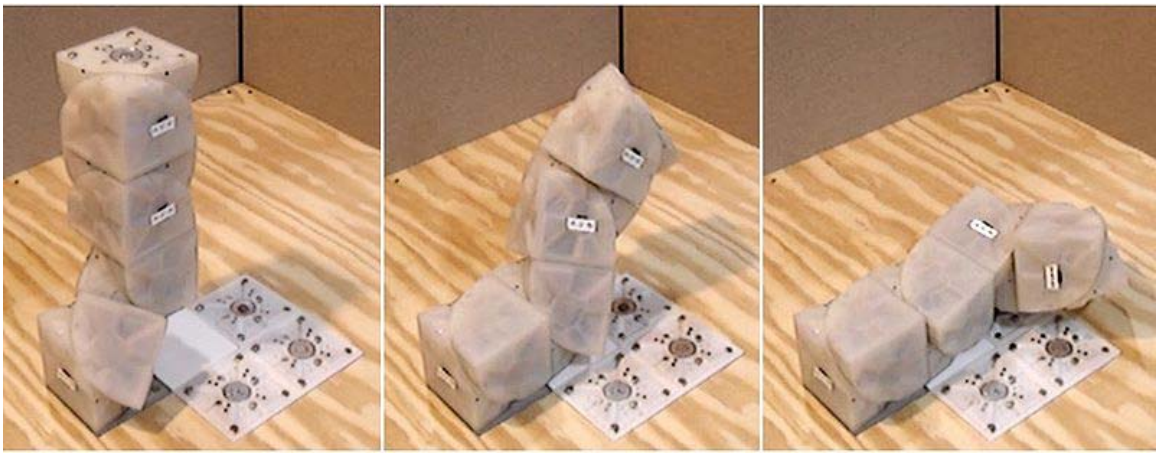
AMOEBA-I (2005)

AMOEBA-I, a three-module reconfigurable mobile robot was developed in Shenyang Institute of Automation (SIA), Chinese Academy of Sciences (CAS) by Liu J G et al..AMOEBA-I has nine kinds of non-isomorphic configurations and high mobility under unstructured environments.Four generations of its platform have been developed and a series of researches have been carried out on their reconfiguration mechanism, non-isomorphic configurations, tipover stability, and reconfiguration planning. Experiments have demonstrated that such kind structure permits good mobility and high flexibility to uneven terrain. Being hyper-redundant, modularized and reconfigurable, AMOEBA-I has many possible applications such as Urban Search and Rescue (USAR) and space exploration.

Stochastic-3D (2005)

High spatial resolution for arbitrary three-dimensional shape formation with modular robots can be accomplished using lattice system with large quantities of very small, prospectively microscopic modules. At small scales, and with large quantities of modules, deterministic control over reconfiguration of individual modules will become unfeasible, while stochastic mechanisms will naturally prevail. Microscopic size of modules will make the use of electromagnetic actuation and interconnection prohibitive, as well, as the use of on-board power storage.

Three large scale prototypes were built in attempt to demonstrate dynamically programmable three-dimensional stochastic reconfiguration in a neutral-buoyancy environment. The first prototype used electromagnets for module reconfiguration and interconnection. The modules were 100 mm cubes and weighed 0.81 kg. The second prototype used stochastic fluidic reconfiguration and interconnection mechanism. Its 130 mm cubic modules weighed 1.78 kg each and made reconfiguration experiments excessively slow. The current third implementation inherits the fluidic reconfiguration principle. The lattice grid size is 80 mm, and the reconfiguration experiments are under way.



Molecubes in motion

Molecubes (2005)

This chain self-reconfiguring system was built by the Cornell Computational Synthesis Lab to physically demonstrate artificial kinematic self-reproduction. Each module is a 0.65 kg cube with 100 mm long edges and one rotational degree of freedom. The axis of rotation is aligned with the cube's longest diagonal. Physical self-reproduction of a three- and a four-module robots was demonstrated. It was also shown that, disregarding the gravity constraints, an infinite number of self-reproducing chain meta-structures can be built from Molecubes. More information can be found at the CCSL Self-Replication webpage.

The Programmable Parts (2005)

The programmable parts are stirred randomly on an air-hockey table by randomly actuated air jets. When they collide and stick, they can communicate and decide whether to stay stuck, or if and when to detach. Local interaction rules can be devised and optimized to guide the robots to make any desired global shape. More information can be found at the programmable parts web page.

SuperBot (2006)

The SuperBot modules fall into the chain/tree architecture. The modules have three degrees of freedom each. The design is based on two previous systems: Conro (by the same research group) and MTRAN (by Murata et al.). Each module can connect to another module through one of its six dock connectors. They can communicate and share power through their dock connectors. Several locomotion gaits have been developed for different arrangements of modules. For high-level communication the modules use hormone-based control, a distributed, scalable protocol that does not require the modules to have unique ID's.

Miche (2006)

The Miche system is a modular lattice system capable of arbitrary shape formation. Each module is an autonomous robot module capable of connecting to and communicating with its immediate neighbors. When assembled into a structure, the modules form a system that can be virtually sculpted using a computer interface and a distributed process. The group of modules collectively decide who is on the final shape and who is not using algorithms that minimize the information transmission and storage. Finally, the modules not in the structure let go and fall off under the control of an external force, in this case gravity. More details at Miche (Rus et al.).

Roombots (2009)

Roombots have a hybrid chain/tree architecture. Each module has three degree of freedom, two of them using the diametrical axis within a regular cube, and a third (center) axis of rotation connecting the two spherical parts. All three axes are continuously rotatory. The outer Roombots DOF is using the same axis-orientation as Molecubes, the third, central Roombots axis enables the module to rotate its two outer DOF against each other. This novel feature enables a single Roombots module to locomote on flat terrain, but also to climb a wall, or to cross a concave, perpendicular edge. Convex edges require the assembly of at least two modules into a Roombots "Metamodule". Each module has ten available connector slots, currently two of them are equipped with an active connection mechanism based on mechanical latches. Roombots are designed for two tasks: to eventually shape objects of daily life, e.g. furniture, and to locomote, e.g. as a quadruped or a tripod robot made from multiple modules. More information can be found at Biorobotics Laboratory Roombots webpage.

Quantitative accomplishment

- The robot with most active modules has 56 units <polybot centipede, PARC>
- The smallest actuated modular unit has a size of <add>mm <add refs>
- The largest actuated modular unit (by volume) has the size of 8 m^3 <(GHFC)giant helium filled catoms, CMU>
- The strongest actuation modules are able to lift 5 identical horizontally cantilevered units.<PolyBot g1v5, PARC>
- The fastest modular robot can move at 23 unit-sizes/second.<CKbot, dynamic rolling, ISER'06>
- The largest simulated system contained many 100,000's of units.

Challenges, solutions, and opportunities

Since the early demonstrations of early modular self-reconfiguring systems, the size, robustness and performance has been continuously improving. In parallel, planning and control algorithms have been progressing to handle thousands on units. There are, however, several key steps that are necessary for these systems to realize their promise of *adaptability, robustness and low cost*. These steps can be broken down into challenges in the hardware design, in planning and control algorithms and in application. These challenges are often intertwined.

Hardware design challenges

The extent to which the promise of self-reconfiguring robotic systems can be realized depends critically on the numbers of modules in the system. To date, only systems with up to about 50 units have been demonstrated, with this number stagnating over almost a decade. There are a number of fundamental limiting factors that govern this number:

- Limits on strength, precision, and field robustness (both mechanical and electrical) of bonding/docking interfaces between modules
- Limits on motor power, motion precision and energetic efficiency of units, (i.e. specific power, specific torque)
- Hardware/software design. Hardware that is designed to make the software problem easier. Self-reconfiguring systems have more tightly coupled hardware and software than any other existing system.

Planning and control challenges

Though algorithms have been developed for handling thousands of units in ideal conditions, challenges to scalability remain both in low-level control and high-level planning to overcome realistic constraints:

- Algorithms for parallel-motion for large scale manipulation and locomotion
- Algorithms for robustly handling a variety of failure modes, from misalignments, dead-units (not responding, not releasing) to units that behave erratically.

- Algorithms that determine the optimal configuration for a given task
- Algorithms for optimal (time, energy) reconfiguration plan
- Efficient and scalable (asynchronous) communication among multiple units

Application challenges

Though the advantages of Modular self-reconfiguring robotic systems is largely recognized, it has been difficult to identify specific application domains where benefits can be demonstrated in the short term. Some suggested applications are

- Space exploration and Space colonization applications, e.g. Lunar colonization
- Construction of large architectural systems
- Deep sea exploration/mining
- Search and rescue in unstructured environments
- Rapid construction of arbitrary tools under space/weight constraints
- Disaster relief shelters for displaced peoples
- Shelters for impoverished areas which require little on-the-ground expertise to assemble

Grand Challenges

Several robotic fields have identified ‘’Grand Challenges’’ that act as a catalyst for development and serve as a short-term goal in absence of immediate ‘’killer apps’’. The Grand Challenge is not in itself a research agenda or milestone, but a means to stimulate and evaluate coordinated progress across multiple technical frontiers. Several Grand Challenges have been proposed for the modular self-reconfiguring robotics field:

- **Demonstration of a system with >1000 units**. Physical demonstration of such a system will inevitably require rethinking key hardware and algorithmic issues, as well as handling noise and error.

- **Robosphere**. A self-sustaining robotic ecology, isolated for a long period of time (1 year) that needs to sustain operation and accomplish unforeseen tasks without *any* human presence.

- **Self replication** A system with many units capable of self replication by collecting scattered building blocks will require solving many of the hardware and algorithmic challenges.

- **Ultimate Construction** A system capable of making objects out of the components of, say, a wall.

- **Biofilter analogy** If the system is ever made small enough to be injected into a mammal, one task may be to monitor molecules in the blood stream and allow some to pass and others not to, somewhat like the Blood-brain barrier. As a challenge, an analogy may be made where system must be able to:

- be inserted into a hole one module's diameter.
- travel some specified distance in a channel that is say roughly 40 x 40 module diameters in area.
- form a barrier fully conforming to the channel (whose shape is non-regular, and unknown beforehand).
- allow some objects to pass and others not to (not based on size).
- Since sensing is not the emphasis of this work, the actual detection of the passable objects should be made trivial.

Inductive Transducers

A unique potential solution that can be exploited is the use of inductors as transducers. This could be useful for dealing with docking and bonding problems. At the same time it could also be beneficial for its capabilities of docking detection (alignment and finding distance), power transmission, and (data signal) communication. A proof-of-concept video can be seen here. The rather limited exploration down this avenue is probably a consequence of the historical lack of need in any applications for such an approach.

Modular Robotics Google Group

Modular Robotics Google Group is an open public forum dedicated to announcements of events in the field of Modular Robotics. This medium is used to disseminate calls to workshops, special issues and other academic activities of interest to modular robotics researchers. The founders of this Google group intend it to facilitate the exchange of information and ideas within the community of modular robotics researchers around the world and thus promote acceleration of advancements in modular robotics. Anybody who is interested in objectives and progress of Modular Robotics can join this Google group and learn about the new developments in this field.