Using Food Web as an Evolution Computing Model for Internet-Based Multimedia Agents

Timothy K. Shih
Multimedia Information NEtwork (MINE) Lab
Department of Computer Science and Information Engineering
Tamkang University
Tamsui, Taipei Hsien, Taiwan 251, R.O.C.
email: TSHIH@CS.TKU.EDU.TW

Abstract

The ecosystem is an evolutionary result of natural laws. Food Web (or Food Chain) embeds a set of computation rules of natural balance. Based one the concepts of Food Web, one of the laws that we may learn from the natural besides neural networks and genetic algorithms, we propose a theoretical computation model for mobile agent evolution on the Internet. We define an agent niche overlap graph and agent evolution states. We also propose a set of algorithms, which is used in our multimedia search programs, to simulate agent evolution. Agents are cloned to live on a remote host station based on three different strategies: the brute force strategy, the semi-brute force strategy, and the selective strategy. Evaluations of different strategies are discussed. Guidelines of writing mobile agent programs are proposed. The technique can be used in distributed information retrieval which allows the computation load to be added to servers, but significantly reduces the traffic of network communication.

1 Introduction

Mobile agents are software programs that can travel over the Internet. Mobile search agents find the information specified by its original query user on a specific station, and send back search results to the user. Only queries and results are transmitted over the Internet. Thus, unnecessary transmission is avoided. In other words, mobile agent computing distributes computation loads among networked stations and reduces network traffic.

The environment where mobile agents live is the Internet. Agents are distributed automatically or semi-automatically via some communication paths. Therefore, agents meet each other on the Internet. Agents have the same goal can share information and cooperate. However, if the system resource (e.g., network bandwidth or disk storage of a station) is insufficient, agents compete with each other. These phenomena are similar to those in the ecosystem of the real world. A creature is born with a goal to live and reproduce. To defense their natural enemies, creatures of the same species cooperate. However, in a perturbation in ecosystems, creatures compete with or even kill each other. The natural world has built a law of balance. Food web (or food chain) embeds the law of creature evolution. With the growing popularity of Internet where mobile agents live, it is our goal to learn from the natural to propose an agent evolution computing model over the Internet. The model, even it is applied only in the mobile agent evolution discussed in this paper, can be generalized to solve other computer science problems. For instance, the search problems in distributed Artificial Intelligence, network traffic control, or any computation that involves a large amount of concurrent/distributed computation. In general, an application of our Food Web evolution model should have the following properties:

- The application must contain a number of concurrent events.
- Events can be simulated by some processes, which can be partitioned into a number of groups according to the properties of events.
- There must exists some consumer-producer relationships among groups so that dependencies can be determined.
- The number of processes must be large enough.

For instance, with the growing popularity of Internet, Web-based documentation are retrieved via some search engine. Search processes can be conducted as several concurrent events distributed among Internet stations. These search events of the same kind (e.g., pursuing the same document) can be formed in a group. Within these agent groups, search agents can provide information to each other. Considering the amount

0-7695-0253-9/99 $10.00 © 1999 IEEE

of Web sites in the future, the quantity of concurrent search events is reasonably large.

We have surveyed articles in the area of mobile agents, personal agents, and intelligent agents. The related works are discussed in section 2. Some terminologies and definitions are given in section 3, where we also introduce the detail concepts of agent communication network. In our model, an agent evolves based on state transitions, which are also discussed. A graph theoretical model describes agent dependencies and competitions is also given. Agent evolution computing algorithms are addressed in section 4. And finally, we discuss our conclusions in section 5.

2 Related Works

The concept of mobile agent is discussed in several articles [3, 4]. Agent Tcl, a mobile-agent system providing navigation and communication services, security mechanisms, and debugging and tracking tools, is proposed in [1]. The system allows agent programs move transparently between computers. A software technology called Telescript, with safety and security features, is discussed in [7]. The mobile agent architecture, MAGNA, and its platform are presented in [3]. Another agent infrastructure is implemented to support mobile agents [4]. A mobile agent technique to achieve load balancing in telecommunications networks is proposed in [6]. The mobile agent programs discussed can travel among network nodes to suggest routes for better communications. Mobile service agent techniques and the corresponding architectural principles as well as requirements of a distributed agent environment are discussed in [2].

3 Definitions

Agents communicate with each other since they can help each other. For instance, agents share the same search query should be able to pass query results to each other so that redundant computation can be avoided. An *Agent Communication Network* (ACN) serves this purpose. Each node in an ACN represents an agent on a computer network node, and each link represents a logical computer network connection (or an agent communication link). Since agents of the same goal want to pass results to each other, agent communication relations can be described in a complete graph. Therefore, an ACN of agents hold different goals is a graph of complete graphs. Since agents can have multiple goals (e.g., searching based on multiple criteria), an agent may belong to different complete graphs.

We define some terminologies used in this paper. A *host station* (or *station*) is a networked workstation on which agents live. A *query station* is a station where a user releases a query for achieving a set of goals. A station can hold multiple agents. Similarly, an agent can pursue multiple goals. An *agent society* (or *society*) is a set of agents fully connected by a complete graph, with a common goal associated with each agent in the society. A goal belongs to different agents may have different priorities. An agent society with a common goal of the same priority is called a *species*. Since an agent may have multiple goals, it is possible that two or more societies (or species) have intersections. A *communication cut set* is a set of agents belong to two distinct agent societies, which share common agents. The removing of all elements of a communication cut set results in the separation of the two distinct societies. An agent in a communication cut set is called an *articulation agent*. Since agent societies (or species) are represented by complete graphs and these graphs have communication cut sets as intersections, articulation agents can be used to suggest a shortest network path between a query station and the station where an agent finds its goal. Another point is that an articulation agent can hold a *repository*, which contains the network communication statuses of links of an agent society. Therefore, network resource can be evaluated when an agent checks its surviving environment to decide its evolution policy.

An agent evolves. It can react to an environment, respond to another agent, and communicate with other agents. The evolution process of an agent involves some internal states. An agent is in one of the following states after it is born and before it is killed or dies of natural:

- **Searching**: the agent is searching for a goal
- **Suspending**: the agent is waiting for enough resource in its environment in order to search for its goal
- **Dangling**: the agent loses its goal of surviving, it is waiting for a new goal
- **Mutating**: the agent is changed to a new species with a new goal and a possible new host station

An agent is born to a *searching state* to search for its goal (i.e., information of some kind). All creatures must have goals (e.g., search for food). However, if its surviving environment (i.e., a host station) contains no enough resource, the agent may transfer to a *suspending state* (i.e., hibernation of a creature). The searching process will be resumed when the environment has better resources. But, if the environment is lack of resources badly (i.e., natural disasters occur), the agent might be killed. When an agent finds its goal, the agent will pass the search results to other agents of the same kind (or same society). Other agents will abort their search (since the goal is achieved) and transfer to a *dangling state*. An agent in a dangling state can not survive for a long time. It will die after some days

(*i.e.*, a duration of time). Or, it will be re-assigned to a new goal with a possible new host station, which is a new destination where the agent should travel. In this case, the agent is in a *mutating state* and is reborn to search for the new goal. Agent evolution states keep the status of an agent. In order to maintain the activity of agents, in a distributed computing environment, we use message passing as a mechanism to control agent state transitions.

Agents can suspend/resume or even kill each other. We need a general policy to decide which agent is killed. By our definition, a species is a set of agents of the same goal with a same priority. It is the priority of a goal we base on to discriminate two or more species.

We need to construct a direct graph which represents the dependency between species. We call this digraph an *species food web* (or *food web*). Each node in the graph represents a species. All species of a connected food web (i.e., a graph component of the food web) are of the same goal with possibly different priorities. We assume that, different users at different host stations may issue the same query with different priority. Each directed edge in the food web has an origin represents a species of a higher goal priority and has a terminus with a lower priority. Since an agent (and thus a species) can have multiple goals which could be similar to other agents, each goal of an articulation agent should have an associated food web. Therefore, the food web is used as a competition base of agents of the same goal in the same station.

Each food web describes goal priority dependencies of species. Form a food web, we can further derive an *niche overlap graph*. In an ecosystem, two or more species have an *ecological niche overlap* (or *niche overlap*) if and only if they are competing for the same resource. A *niche overlap graph* can be used to represent the competition among species. The niche overlap graph is used in our algorithm to decide agent evolution policy and to estimate the effect when certain factors are changed in an agent communication network. Based on the niche overlap graph, the algorithm is able to suggest strategies to re-arrange policies so that agents can achieve their highest performance efficiency. This concept is similar to the natural process that recover from perturbations in ecosystems.

4 Agent Evolution Computing

The algorithms proposed in this section use the agent evolution states and the niche overlap graphs discussed for agent evolution computing. An agent wants to search for its goal. At the same time, since the searching process is distributed, an agent wants to find a destination station to clone itself. Searching and cloning are essentially exist as a *co-routing relation*. A co-routine can be a pair of processes. While one process serves as a producer, another serves as a consumer. When the consumer uses out of the resource, the consumer is suspended. After that, the producer is activated and produces the resource until it reaches an upper limit. The producer is suspended and the consumer is resumed. In the computation model, the searching process can be a consumer, which need new destinations to proceed search. On the other hand, the cloning process is a producer who provides new URLs.

Agent evolution on the agent communication network is an asynchronous computation. Agents live on different (or the same) stations communicate and work with each other via agent messages. The searching and the cloning processes of an agent may run as a co-routine on a station. However, different agents are run on the same or separated stations concurrently. We use a formal specification approach to describe the logic of our evolution computation. Formal specifications use first order logic, which is precise. In this paper, we use the *Z* specification language to describe the model and algorithms.

Each algorithm or global variable in our discussion has two parts. The expressions above a horizontal line are the signatures of predicates, functions, or the data types of variables. Predicates and functions are constructed using quantifiers, logic operators, and other predicates (or functions). The signature of a predicate also indicate the type of its formal parameters. For instance, $Agent \times Goal \times Host\_Station$ are the types of formal parameters of predicate *Agent_Search*. The body, as the second part of the predicate, is specified below the horizontal line.

We use some global variables through the formal specification. The variable *goal_achieved* is set to *TRUE* when the search goal is achieved, *FALSE* otherwise. We also use two watermark variables, α and β, where α is the basic system resource requirement and β is the minimal requirement. Note that, α must be greater than β so that different levels of treatment are used when the resource is not sufficient.

Global Variables and Constants

$$\begin{array}{|l} goal\_achieved : Goal\_Achieved \\ \alpha : REAL \\ \beta : REAL \\ \hline \alpha > \beta \end{array}$$

Algorithm *Agent_Search* is the starting point of agent evolution simulation. If system resource meets a basic requirement (i.e., α), the algorithm activates an agent in the searching state within a local station. If the search process finds its goal (e.g., the requested information is found), the goal is achieved. Goal abortion of all agents in a society results in a dangling

state of all agents in the same society (including the agent who finds the goal). At the same time, the search result is sent back to the original query station via *Query_Return_URL*. Suppose that the goal can not be achieved in an individual station, the agent is cloned in another station (agent propagation). The *Agent_Clone* algorithm is then used. On the other hand, the agent may be suspended or even killed if the system resource is below the basic requirement (i.e., $Resource\_Available(A, G, X) < \alpha$). In this case, algorithms *Agent_Suspend* is used if the resource available is still feasible for a future resuming of the agent. Otherwise, if the resource is below the minimal requirement, algorithm *Agent_Kill* is used.

Agent Searching Algorithm

$$
\begin{array}{|l}
Agent\_Search : Agent \times Goal \times Host\_Station \\
\hline
\forall A : Agent, G : Goal, X : Host\_Station \bullet \\
\quad Agent\_Search(A, G, X) \Leftrightarrow \\
\qquad Resource\_Available(A, G, X) \geq \alpha \Rightarrow \\
\qquad\quad [G \in Local\_Search(A, X) \Rightarrow \\
\qquad\qquad Abort\_All(A \uparrow Agent\_Society) \wedge \\
\qquad\qquad send\_result(X.URL, \\
\qquad\qquad\quad G.Query\_Return\_URL) \wedge \\
\qquad\qquad goal\_achieved = TRUE \\
\qquad\quad \vee\ G \notin Local\_Search(A, X) \Rightarrow \\
\qquad\qquad Agent\_Clone(A, G, \\
\qquad\qquad\quad A \uparrow Agent\_Society)] \\
\qquad \vee\ Resource\_Available(A, G, X) \geq \beta \Rightarrow \\
\qquad\quad Agent\_Suspend(A, G, X) \\
\qquad \vee\ Resource\_Available(A, G, X) < \beta \Rightarrow \\
\qquad\quad Agent\_Kill(A, G, X)
\end{array}
$$

Agent cloning is achieved by the *Agent_Clone* algorithm. When the cloning process wants to find new stations to broadcast an agent, two implementations can be considered. The first is to collect all URLs of stations found by one search engine. But, considering the network resource available, the implementation may check for the common URLs found by two or more search engines. New URLs are collected by the *Search_For_Stations* algorithm, which is invoked in the agent cloning algorithm. Agent propagation strategy decides the computation efficiency of our model. In this research, we propose three strategies:

- the brute force agent distribution
- the semi-brute force agent distribution, and
- the selective agent distribution.

The first strategy simply clone an agent on a remote station, if the potential station contains information that helps the agent to achieve its goal. The semi-brute force strategy, however, finds another agent on a potential station, and assigns the goal to that agent. The selective approach not only try to find a useful agent, but also check for the goals of that agent. Cloning strategies affect the size of agent societies thus the efficiency of computation.

Agent Cloning Algorithm: the Brute Force Strategy

$$
\begin{array}{|l}
Agent\_Clone : Agent \times Goal \times Agent\_Society \\
\hline
\forall A : Agent, G : Goal, S : Agent\_Society \bullet \\
\quad Agent\_Clone(A, G, S) \Leftrightarrow \\
\qquad [\forall X : Host\_Station \bullet \\
\qquad\quad X \in Search\_For\_Stations(G) \Rightarrow \\
\qquad\quad (\exists A' : Agent \bullet A' = copy(A) \wedge \\
\qquad\quad X.Agent\_Set = X.Agent\_Set \cup \{\ A'\ \} \wedge \\
\qquad\quad S = S \cup \{\ A'\ \} \wedge \\
\qquad\quad Agent\_Search(A', G, X))] \\
\qquad \vee\ [Search\_For\_Stations(G) = \emptyset \Rightarrow \\
\qquad\quad goal\_achieved = FALSE]
\end{array}
$$

The brute force agent distribution strategy makes a copy of agent A, using the *copy* function, in all stations returned by the *Search_For_Stations* algorithm. Agent set in each station is updated and the society S where agent A belongs is changed. Agent A', a clone of agent A is transmitted to station X for execution.

Agent Cloning Algorithm: the Semi-brute Force Strategy

$$
\begin{array}{|l}
Agent\_Clone : Agent \times Goal \times Agent\_Society \\
\hline
\forall A : Agent, G : Goal, S : Agent\_Society \bullet \\
\quad Agent\_Clone(A, G, S) \Leftrightarrow \\
\qquad [\forall X : Host\_Station \bullet \\
\qquad\quad X \in Search\_For\_Stations(G) \Rightarrow \\
\qquad\quad [\exists A' : Agent \bullet A' \in X.Agent\_Set \Rightarrow \\
\qquad\quad (A'.Goal\_Set = A'.Goal\_Set \cup \\
\qquad\quad\quad \{\ G\ \} \wedge \\
\qquad\quad\quad S = S \cup \{\ A'\ \} \wedge \\
\qquad\quad\quad Agent\_Search(A', G, X))]] \\
\qquad \vee\ [Search\_For\_Stations(G) = \emptyset \Rightarrow \\
\qquad\quad goal\_achieved = FALSE]
\end{array}
$$

The semi-brute force agent distribution approach is similar to the brute force approach, except that it does not make a copy of the agent but give the goal to an agent on its destination station. The agent which accepts this new goal (i.e., A') is activated for the new goal in its belonging station.

Agent Cloning Algorithm: the Selective Strategy

$$
\begin{array}{l}
Agent\_Clone : Agent \times Goal \times Agent\_Society \\
\hline
\forall A : Agent, G : Goal, S : Agent\_Society \bullet \\
\quad Agent\_Clone(A, G, S) \Leftrightarrow \\
\quad\quad [\forall X : Host\_Station \bullet \\
\quad\quad\quad X \in Search\_For\_Stations(G) \Rightarrow \\
\quad\quad\quad [\exists A' : Agent \bullet A' \in X.Agent\_Set \Rightarrow \\
\quad\quad\quad\quad [G \in A'.Goal\_Set \Rightarrow \\
\quad\quad\quad\quad\quad S = S \cup A' \uparrow Agent\_Society \\
\quad\quad\quad\quad \vee\ G \notin A'.Goal\_Set \Rightarrow \\
\quad\quad\quad\quad\quad (A'.Goal\_Set = \\
\quad\quad\quad\quad\quad\quad A'.Goal\_Set \cup \{\ G\ \} \\
\quad\quad\quad\quad\quad \wedge\ S = S \cup \{\ A'\ \})] \\
\quad\quad\quad\quad \wedge\ Agent\_Search(A', G, X)] \\
\quad\quad\quad \vee\ [X.Agent\_Set = \emptyset \Rightarrow \\
\quad\quad\quad\quad [\exists A'' : Agent \bullet A'' = copy(A) \wedge \\
\quad\quad\quad\quad X.Agent\_Set = \{\ A''\ \} \wedge \\
\quad\quad\quad\quad S = S \cup \{\ A''\ \} \wedge \\
\quad\quad\quad\quad Agent\_Search(A'', G, X)]]] \\
\quad\quad\quad \vee\ [Search\_For\_Stations(G) = \emptyset \\
\quad\quad\quad\quad \Rightarrow goal\_achieved = FALSE]
\end{array}
$$

The last approach is more complicate. The selective approach of cloning algorithm must check whether there is another agent in the destination station (i.e., X). If so, the algorithm checks whether the agent (i.e., A') at that station shares the same goal with the agent to be cloned. If two agents share the same goal, there is no need of cloning another copy of agent. Basically, the goal can be computed by the agent at the destination station. In this case, the union of the two societies is necessary (i.e., $S = S \cup A' \uparrow Agent\_Society$). On the other hand, if the two agents do not have a common goal, to save computation resource, we may ask the agent at the destination station to help searching for an additional goal. This case makes a re-organization of the society where the source agent belongs. The result also ensure that the number of agents on the ACN is kept in a minimum. Whether the two agents share the same goal, the *Agent_Search* algorithm is used to search for the goal again. In this case, Agent A' is physically transmitted to station X for execution. When there is no agent running on the destination station, we need to increase the number of agents on the ACN by duplicating an agent on the destination station (i.e., the invocation of $A'' = copy(A)$). The society is reorganized. And the *Agent_Search* algorithm is called again. In the acse that no new station is found by the *Search_For_Stations* algorithm, the goal is not achieved.

The agent search and agent clone algorithms use some auxiliary algorithms, which are discussed as follows. The justification of system resource available depends on agent policy, as defined in *A.Policy*. Agent policy is a set of factors indicated by name tags (e.g., *NETWORK_BOUND*). The estimation of resources is represented as a real number, which is computed based on *X.Resource* of station X. Note that, in the algorithm, $w1$ and $w2$ are weights of factors ($w1 + w2 = 1.0$). We only describes some cases of using agent policies. Other cases are possible but omitted. Moreover, we consider the priority of goal G. If the priority is lower than some watermark (i.e., $G.Priority < \theta$), we let $r1$ be a constant less than 1.0. Therefore, resources are reserved for other agents. On the other hand, if the priority is high, we consider the value returned by *Resource_Available* should be high. Thus the potential agent can proceed its computation immediately. The values of θ and ω depend on agent applications.

Auxiliary Algorithms

$$
\begin{array}{l}
Resource\_Available : Agent \times Goal \times Host\_Station \rightarrow \\
\quad REAL \\
\hline
\forall A : Agent, G : Goal, X : Host\_Station, R : REAL \bullet \\
\quad \exists w1, w2, r1, r2 : REAL \bullet \\
\quad Resource\_Available(A, G, X) = R \Leftrightarrow \\
\quad\quad [NETWORK\_BOUND \in A.Policy \Rightarrow \\
\quad\quad\quad R = X.Resource.Network \\
\quad\quad \vee\ CPU\_BOUND \in A.Policy \Rightarrow \\
\quad\quad\quad R = X.Resource.CPU \\
\quad\quad \vee\ MEMORY\_BOUND \in A.Policy \Rightarrow \\
\quad\quad\quad R = X.Resource.Memory \\
\quad\quad \vee\ CPU\_BOUND \in A.Policy \wedge \\
\quad\quad MEMORY\_BOUND \in A.Policy \Rightarrow \\
\quad\quad\quad R = X.Resource.CPU * w1 + \\
\quad\quad\quad\quad X.Resource.Memory * w2 \wedge \\
\quad\quad\quad w1 + w2 = 1.0 \\
\quad\quad \vee\ ...] \\
\quad\quad \wedge\ \exists \theta, \omega : Priority \bullet \\
\quad\quad\quad [G.Priority < \theta \Rightarrow \\
\quad\quad\quad\quad (R = R * r1 \wedge r1 < 1.0) \\
\quad\quad\quad \vee\ G.Priority > \omega \Rightarrow \\
\quad\quad\quad\quad (R = R * r2 \wedge r2 > 1.0)]
\end{array}
$$

The above algorithms describe how an agent evolves from a state to another. How agents affect each other depends on the system resource available. However, in an ACN, it is possible that agents suspend or even kill each other, as we described in previous sections. The niche overlap graphs of each goal play an important role. We use the *Agent_Suspend* and *Agent_Kill* algorithms to take the niche overlap graphs of a goal (i.e., $niche\_compete(G)$) into consideration. In the *Agent_Suspend* algorithm, if there exists a goal that has a lower priority comparing to the goal of the searching agent, a suspend message is sent to the goal to delay its search (i.e., via $suspend(G' \uparrow Agent)$). The searching agent may be resumed after that since system resources may be released from those goal suspension. In the *Agent_Kill* algorithm, however, a kill message is sent instead (i.e., via $terminate(G' \uparrow Agent)$). The system resource is checked against the minimum requirement β. If resuming is feasible, the *Agent_Search* algorithm in invoked. Otherwise, the system should terminate the searching agent.

$$Agent\_Suspend : Agent \times Goal \times Host\_Station$$

$$\begin{aligned}
&\forall A : Agent, G : Goal, X : Host_Station \bullet \\
&\quad Agent_Suspend(A, G, X) \Leftrightarrow \\
&\quad \exists GS : Goal_Set \bullet \\
&\qquad GS = niche_compete(G) \\
&\qquad \wedge (\forall G' : Goal \bullet G' \in GS \wedge \\
&\qquad\quad G'.Priority < G.Priority \Rightarrow \\
&\qquad\qquad suspend(G' \uparrow Agent)) \\
&\qquad \wedge (Resource_Available(A, G, X) \geq \beta \Rightarrow \\
&\qquad\quad Agent_Search(A, G, X) \\
&\qquad \vee Resource_Available(A, G, X) < \beta \Rightarrow \\
&\qquad\quad suspend(A))
\end{aligned}$$

$$Agent\_Kill : Agent \times Goal \times Host\_Station$$

$$\begin{aligned}
&\forall A : Agent, G : Goal, X : Host_Station \bullet \\
&\quad Agent_Kill(A, G, X) \Leftrightarrow \\
&\quad \exists GS : Goal_Set \bullet \\
&\qquad GS = niche_compete(G) \\
&\qquad \wedge (\forall G' : Goal \bullet G' \in GS \wedge \\
&\qquad\quad G'.Priority < G.Priority \Rightarrow \\
&\qquad\qquad terminate(G' \uparrow Agent)) \\
&\qquad \wedge (Resource_Available(A, G, X) \geq \beta \Rightarrow \\
&\qquad\quad Agent_Search(A, G, X) \\
&\qquad \vee Resource_Available(A, G, X) < \beta \Rightarrow \\
&\qquad\quad terminate(A))
\end{aligned}$$

The other auxiliary algorithms are relatively less complicated. Function *Local_Search* takes as input an agent and a station. It returns a set of goals found by the agent in that station. A *match* predicate is used. This match predicate is application dependent. It could be a search program which locates a key word in a Web page, or a request of information from a user (e.g., a survey questionnaire). The *Abort_All* predicate takes as input an agent society and terminates all agents within that society. The *Search_For_Stations* function takes as input a goal and returns a set of host stations. The stations should be selected depending on the *candidate_station* function, which estimates the possibility of goal achievement in a station. This function can be implemented as a Web search engine which looks for candidate URLs. We have omitted some detailed definitions of the above auxiliary algorithms, as well as some primitive functions which are self-explanatory.

$$Local\_Search : Agent \times Host\_Station \rightarrow Goal\_Set$$

$$\begin{aligned}
&\forall A : Agent, X : Host_Station, GS : Goal_Set \bullet \\
&\quad Local_Search(A, X) = GS \Leftrightarrow \\
&\qquad GS = \{\ G : Goal \mid G \in A.Goal_Set \wedge \\
&\qquad\quad match(G.Query, \\
&\qquad\qquad X.Resource.Information)\ \}
\end{aligned}$$

$$Abort\_All : Agent\_Society$$

$$\begin{aligned}
&\forall S : Agent_Society \bullet \\
&\quad Abort_All(S) \Leftrightarrow \\
&\qquad \forall A : Agent \bullet A \in S \Rightarrow terminate(A)
\end{aligned}$$

$$Search\_For\_Stations : Goal \rightarrow \mathbb{P}\ Host\_Station$$

$$\begin{aligned}
&\forall G : Goal, X_Set : \mathbb{P}\ Host_Station \bullet \\
&\quad Search_For_Stations(G) = X_Set \Leftrightarrow \\
&\qquad X_Set = \{\ X : Host_Station \mid \\
&\qquad\quad candidate_station(G, X)\ \}
\end{aligned}$$

5 Conclusions

Mobile agent based software engineering is interesting. However, in the literature, we did not find any other similar theoretical approach to model what mobile agents should act on the Internet, especially how mobile agents can cooperate and compete. A theoretical computation model for agent evolution was proposed in this paper. Algorithms for the realization of our model were also given.

References

[1] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko, "Agent Tcl: targeting the needs of mobile computers," IEEE Internet Computing, Vol. 1, No. 4, July 1997, pp. 58 – 67.

[2] S. Krause and T. Magedanz, "Mobile service agents enabling intelligence on demand in telecommunications," in Proceedings of the 1996 IEEE Global Telecommunications Conference, London, UK, 1996, pp 78 – 84.

[3] Sven Krause, Flavio Morais de Assis Silva, and Thomas Magedanz, "MAGNA - a DPE-based platform for mobile agents in electronic service markets," in Proceedings of the 1997 3rd International Symposium on Autonomous Decentralized Systems (ISADS'97), Berlin, Germany, 1997, pp. 93 – 102.

[4] Anselm Lingnau and Oswald Drobnik, "Making mobile agents communicate: a flexible approach," in Proceedings of the 1996 1st Annual Conference on Emerging Technologies and Applications in Communications, Portland, OR, USA, 1996, pp. 180 – 183.

[5] Michael Pazzani and Daniel Billsus, "Learning and Revising User Profiles: The Identification of Interesting Web Sites", Machine Learning, Vol. 27, 1997, pp. 313 – 331.

[6] Ruud Schoonderwoerd, Owen Holland, and Janet Bruten, "Ant-like agents for load balancing in telecommunications networks," in Proceedings of the 1997 1st International Conference on Autonomous Agents, Marina del Rey, California, U.S.A., 1997, pp. 209 – 216.

[7] Joseph Tardo and Luis Valente, "Mobile agent security and telescript," in Proceedings of the 1996 41st IEEE Computer Society International Conference (COMPCON'96), Santa Clara, CA, USA, 1996, pp. 58 – 63.