IMAGO: A Prolog-based System for Intelligent Mobile Agents

Xining Li

Department of Computing and Information Science
University of Guelph, Canada
xli@snowhite.cis.uoguelph.ca

Abstract. This paper presents the preliminary design of the IMAGO project. This project consists of two major parts: the IMAGO Application Programming Interface (API) - an agent development kit based on Prolog, and the MLVM - a multithreading agent server framework. We focus on the IMAGO API and its communication model - a novel mechanism to automatically track down agents and deliver messages in a dynamic, changing world. Examples are given to show the expressive power and simplicity of the programming interface as well as possible applications of the proposed system.

1 Introduction

Mobile Agents are mainly intended to be used for network computing - applications distributed over large scale computer networks. In general, a mobile agent is a self-contained process that can autonomously migrate from host to host in order to perform its task on behalf of a (human) user. Numerous Mobile Agents systems have been implemented or are currently under development. System-level issues and language-level requirements that arise in the design of Mobile Agents systems are well discussed in [1].

Most of the Mobile Agents systems are based on scripting or interpreted programming languages that offer portable virtual machines for executing agent code, as well as a controlled execution environment featuring a security mechanism that restricts access to the host's private resources. Some Mobile Agents systems are based on Java [2] [3] [4] [5] [6], and some are based on other object oriented programming languages or scripting languages [7] [8] [9]. As the primary identifying characteristic of a mobile agent is its ability to migrate from host to host, support for agent mobility is a fundamental requirement of a Mobile Agents system. An agent is normally composed of three parts: code, execution thread (stack), and data (heap). All these parts move with the agent whenever it moves. However, most of the Mobile Agents systems (especially those built on top of Java) only support weak migration - an agent moves with its code and data without its stack of the execution thread. Thus, the agent has to direct the control flow appropriately when its state is restored at the destination. For example, a Java-based agent captures/restores its execution state through the

Java's *serialising/de-serialising* feature which provides a means for translating a graph of objects into a byte-stream and thus achieves migration at a coarse granularity. This implies that an agent restarts execution from the beginning each time it moves to another host. As a result, the agent has to include some tracing code in order to find its continuation point upon each migration.

Another mobile agent framework which embeds a logic programming component is pioneered by Distributed Oz[11] - a multi-paradigm language (functional, logic, object-oriented, and constraint), and Jinni[10] - a lightweight, multithreaded, Prolog-based language (supporting mobile agents through a combination of Java and Prolog components). Distributed Oz does not support thread-level mobility, instead, it provides protocols to implement mobility control for *objects*. In a user program, the mobility of an object must be well-defined under the illusion of a single network-wide address space for all entities (include threads, objects and procedures). Jinni implements computation mobility by capturing continuations (describing future computations to be performed at a given point) at the thread-level. A live thread will migrate from Jinni to a faster remote BinProlog engine, do some CPU intensive work and then come back with the results.

This paper will discuss the design of the IMAGO project. The origin of the word *imago* means that

> An insect in its final, adult sexually mature, and typically winged state, or an idealized mental image of another person or the self.
>
> *WEBSTER's Dictionary*

In my proposal, imagoes are programs written in a variant of Prolog that can fly from one host on the Internet to another. That is, an imago is characterized as an entity which is mature (automonous and self-contained), has wings (mobility), and bears the mental image of the programmer (intelligent agent). From computer terminology point of view, the term IMAGO is an abbreviation which stands for Intelligent Mobile Agents Gliding On-line.

The IMAGO project consists of two major parts: the IMAGO Application Programming Interface (API) - an agent development kit based on Prolog, and the MLVM - a multithreading agent server framework based on a sequential logic virtual machine LVM [12].

The IMAGO API consists of a set of primitives that allows programmer to create mobile agent applications. In general, a mobile agents system provides primitives for agent management (creation, dispatching, migration), agent communication/synchronization, agent monitoring (query, recall, termination), *etc.* In most concurrent programming languages, communication primitives take the form of message passing, remote procedure calls, or blackboard-based. For example, SICStus MT[13] uses the asynchronous message-passing mechanism whereas Bin-Prolog[10] adopts the blackboard-based model. On the other hand, IMAGO explores a novel model: instead of passing messages among agents through send/receive primitives, the IMAGO implements agent communication through *messengers* - special mobile agents dedicated to deliver messages on the network.

The goal of MLVM is to present a logic-based framework in the design space of Intelligent Mobile Agents server. To achieve this, we need to extended the LVM to cope with new issues, such as explicit concurrency, code autonomy, communication/synchronization and computation mobility. In designing the MLVM, some practical issues, such as multithreading, garbage collection, code migration, communication mechanism, *etc*, have been specified, whereas some other issues, such as security, services, *etc*, will be investigated further.

2 Overview of Imagoes

The IMAGO system is an infrastructure that implements the agent paradigm. An IMAGO server resides at a host machine intending to host imagoes and provide a protected imago execution environment. An IMAGO server consists of three components: a network *daemon* to accept incoming imagoes, a security manager to deal with privacy, physical access restrictions, application availability, network confidentiality, content integrity, and access policy, and a MLVM engine to schedule and execute imago threads.

Generally speaking, an imago is composed of three parts: its identifier which is unique to distinguish with others, its code which corresponds to a certain algorithm, its execution thread which is maintained by a single memory block (a merged stack/heap with automatic garbage collection)[12].

There are three kinds of imagoes: *stationary imago*, *worker imago*, and *messenger imago*. An agent application starts from a stationary imago. It looks like that the wings of a stationary imago have degenerated, so that it has lost its mobility. In other words, a stationary imago always executes on the host where it begins execution. However, a stationary imago has the privileges to access resources of its host machine, such as I/O, files, GUI manager, *etc*. A stationary imago can create worker or messenger imagoes, but it can not clone itself. There is only one stationary imago in an application. We can find the similarity that there is only one queen in a colony of bees.

Worker imagoes are created by the stationary imago of an application. A worker imago is able to move such that it looks like a worker bee flying from place to place. A worker imago can clone itself. A cloned worker imago is an identical copy of the original imago but with a different identifier. A worker imago can not *create* other worker imagoes, however, it may launch messenger imagoes (system built-in imagoes) to deliver messages. When a worker imago moves from one host to another, it continues its execution on the destination host at the instruction which immediately follows the invocation of the move primitive. As mobile agents are a potential threat to harm the remote hosts that they are visiting, the IMAGO system enforces a tight access control on worker imagoes: they have no right to access any kind of system resources except the legal services provided by the server. A messenger queue is associated with each worker imago which holds all attached messenger imagoes waiting to deliver messages.

Messenger imagoes are agents dedicated to deliver messages. The reason of introducing such special purpose imagoes is that the peer to peer communication mechanism in traditional concurrent (distributed) programming languages does not fit the paradigm of mobile agents. This is because mobile agents are autonomous - they may decide where to go based on their own will or the information they have gathered. Most mobile agents systems either do not provide the ability of automatically tracing moving agents, or try to avoid discussing this issue. For example, Aglet API does not support agent tracking, instead, it leaves this problem to applications. On the other hand, the IMAGO system allows messenger imagoes to track worker imagoes and therefore achieves reliable message delivery. The system provides several builtin messenger imagoes. Programmer designed messenger imagoes are possible but this kind of imagoes can only be created by the stationary imago. A messenger imago is anonymous so that there is no way to track a messenger. However, it can move or even clone itself if necessary.

3 Imago API

The code of an imago is enclosed in a pair of directives. Here we follow the Prolog convention such that a directive specifies properties of the procedure defined in Prolog text. Three pairs of directives are used for imago definitions, and they share the same syntax. For example, the following code gives a syntactical pattern of a messenger imago:

```
:- begin_messenger
my_messenger(Receiver, Msg) ::-
        messenger_body, ...
:- end_messenger
```

In each imago, one and only one clause is defined by ::- which indicates the starting entry of the imago, and the rest clauses, if any, are defined by the Prolog convention. The entry clause can not be explicitly called, instead, the IMAGO runtime system automatically provides a goal toward the entry clause after an imago text has been prepared for execution.

Even though several imago definitions can be placed in a single source file, the IMAGO compiler will compile them independently and save the bytecode of each imago into a separate file (the file name is composed by the name of its entry clause with a postfix *.ima*). For the above code pattern, its bytecode file is named as *my_messenger.ima*.

Messenger imagoes are anonymous. As there is only one stationary imago in an application, we reserve a special name *queen* for it. Names of worker imagoes must be presented at the time they are created.

Like other logic programming systems, the IMAGO API is presented as a set of builtin predicates. This set consists of builtin predicates common to most Prolog-based systems and new builtin predicates extended for mobile agent applications. As we mentioned before, resource access predicates and user-machine interface predicates can be used only in a stationary imago. In addition, the usage of agent management predicates depends on the type of imagoes, such as illustrated in Table 1 which lists predicates legal to each imago type. This table is far from complete, but should be sufficient to describe my project proposal.

Imago Type	**Builtin Predicates**
stationary imago	create, accept, wait_accept, dispatch, terminate
worker imago	move, clone, back, accept, wait_accept, dispatch, dispose
messenger imago	move, clone, back, attach, dispose

Table 1: Builtin Predicates for Imagoes

In principle, all these predicates are not re-executable. Different kinds of errors, such as type error, resource error, system error, *etc.*, might happen during their execution. However, for the sake of simplicity, we discuss these predicates in an informal manner, *i.e.*, we only present a brief procedural description for each predicate.

create(Worker_file, Name, Argument): *Create* is used only by the stationary imago. It will load the *Worker_file*, spawn a new thread to execute the worker imago, put this new thread into the ready queue, and set up the imago's *Name* and initial *Argument*.

dispatch(Messenger_file, Receiver, Msg): *Dispatch* is used to create a messenger imago which is responsible to search for the *Receiver* and deliver *Msg*. A worker imago can only dispatch system builtin messengers (which will be automatically created by imago servers), whereas the stationary imago can dispatch either system builtin messengers or programmer designed messengers (which can be loaded from the local file system). A messenger will implicitly carry the sender's name (name of the imago which invokes the messenger) which is accessible by some other predicates.

attach(Receiver, Msg, Result): *Attach* is used only by messenger imagoes and probably the most complicated predicate in the IMAGO API. It will first search for the *Receiver* through its server's log or probability through the IMAGO name server, instantiate *Result* to *moved(S)* if the receiver has moved to another host *S*, or *deceased* if the receiver could not be found. On the other hand, if the receiver is found currently alive, it will deactivate the calling messenger, and attach the caller to the receiver's messenger queue. As soon as a messenger has been attached to the receiving imago, its thread is suspended until the receiver executes certain predicate to resume its execution. In this case, we say that the *attach* predicate is blocked.

move(Server): Invoking *move* allows a worker or a messenger to migrate to another imago server. This predicate deactivates the caller, captures its state, and transmits it to the given remote *Server*. When a worker issues *move* and there are pending messengers in its messenger queue, all these suspended messengers will be resumed and the term *moved(Server)* will be instantiated to the *Result* of each blocked *attach* predicate. This does not apply to a moving messenger, because messengers are anonymous and thus there is no way to attach a messenger to another messenger. However, a resumed messenger should follow the moving worker to the new host in order to deliver its message.

clone(Name, Result): *Clone* will duplicate the caller (either a worker or a messenger) as a new imago thread with the given *Name* (anonymous for a messenger). The behavior of *clone* resembles the *fork()* in C where two imagoes continue their execution at the instruction immediately following the *clone* predicate but each has a different *Result* instantiation: *origin* to the caller imago and *clone* to the duplicated imago. When a worker issues *clone* and there are pending messengers in its messenger queue, all these suspended messengers will be resumed and the term *cloned(Clone)* will be instantiated to the *Result* of each blocked *attach* predicate. Under this case, a resumed messenger must clone itself and then the original messenger re-attaches itself to the original receiver and the cloned messenger attaches itself to the cloned worker imago. A messenger example can be found in next section.

back: An imago calling *back* will move itself back to the host where the stationary imago resides in. The same as the *move*, this predicate will resume all pending messengers of a worker and bind *Result* to *moved(stationary_server)*. Thus a resumed messenger should follow the receiver back to their home station.

accept(Sender, Msg): Stationary and worker imagoes can issue an *accept* to receive a message. It will succeed if a matching messenger has been found and the messenger will be resumed with an instantiation *received* to the *Result* argument, or it will fail if either the messenger queue is empty or no matching messenger can be found. *Accept* will never block, and is powerful enough to achieve indeterministic message receiving.

wait_accept(Sender, Msg): *Wait_accept* will cause its caller to be blocked (from the ready queue to a waiting queue) if either the caller's messenger queue is empty, or no matching messenger is found. It will succeed immediately if there is a pending matching messenger. An imago being blocked by this predicate will become ready when a new messenger attaches to it. A resumed imago will automatically redo this predicate: it succeeds if the new attached messenger matches, or it blocks the imago again otherwise. In other words, a *wait_accept* will never fail. It either succeeds or becomes blocked waiting for a matching messenger.

dispose: *Dispose* terminates the calling imago. All the pending messengers, if any, will be resumed with a *Result* bound to *deceased*. It is up to messengers to determine if they also dispose themselves or move back to notifying their senders.

terminate: This predicate is called by the stationary imago to terminate the application and eliminate all imagoes spawned (cloned) from this application.

A messenger attached to a worker imago is ready to be searched by *accept* or *wait_accept*. The behavior of an accepting predicate is determined by the unification of its arguments against pending messengers: it succeeds if a matching messenger is found, or it fails/waits otherwise. A failed accepting predicate does not cause any side effect and the messenger queue remains unchanged.

From the state transition description, we can find that a stationary or a worker imago becomes blocked only if its messenger queue is empty or no matching messenger is found at the time a *wait_accept* is invoked, and it is resumed to ready when a messenger attachment occurs. On the other hand, a messenger becomes blocked when it is attached to a receiver, and it is unblocked when its receiver evaluates one of the following predicates: *move*, *clone*, *back*, *dispose*, or *accept* if the messenger matches.

4 Messenger Imago

The IMAGO system provides a set of builtin messenger imagoes as a part of the IMAGO API. These messengers should be robust and sufficient for most imago applications. They may be dispatched by either a stationary imago or a worker imago. For the sake of flexibility, a stationary imago may also dispatch user designed messengers. In this case, the system will load the user designed messenger code from the local host, create a thread and add the messenger thread into the ready queue for execution.

In this section, we will discuss the design pattern of system builtin messengers. Each system builtin messenger has a given code name. The following example shows an asynchronous messenger named as *$oneway_messenger*. It is worth to note that this name is the *code* name, rather than the imago's name, because messenger imagoes are anonymous.

```
:- begin_messenger
$oneway_messenger(Receiver, Msg)::- deliver(Receiver, Msg).
deliver(Receiver, Msg):- attach(Receiver, Msg, Result),
        check(Receiver, Msg, Result).
check(_, _, received):- !, dispose.
check(Receiver, Msg, moved(Server)):-!, move(Server),
        deliver(Receiver, Msg).
check(Receiver, Msg, cloned(Clone)):- !, clone(_, R),
        R == clone →
            deliver(Clone, Msg);
            deliver(Receiver, Msg).
check(_, _, deceased):-!, dispose.
:- end_messenger
```

When the *$oneway_messenger* is started, it tries to attach itself to the given receiver. Only two possible cases make the *attach* succeed immediately: either the receiver has moved or the receiver has deceased (here we consider the receiver dead if it could not be found through the IMAGO name resolution). For the former case, this messenger will follow the receiver by calling *move* and then try to deliver its message at the new host; for the later case, the messenger simply disposes itself. Otherwise, the receiver must be alive at the current host, thus the messenger attaches to this receiver and makes the receiver ready if the receiver was blocked by a *wait_accept*.

After having attached to its receiver, the messenger is suspended. There is no guarantee that the receiver will release this attached messenger by calling an accept-type predicate, because the receiver is free to do anything, such as *move*, *back* or *clone* before issuing an accept, or even *dispose* without accepting messengers. For this reason, a resumed messenger must be able to cope with different cases and try to re-deliver the message if the message has not been received yet and the receiver is still alive.

An interesting case is when the receiver imago clones itself while it has pending messengers. In order to follow the principle that a cloned imago must be an identical copy of its original, all attached messengers must also clone themselves and then attach to the cloned imago. From the *$oneway_messenger* program, we can find that after knowing that the receiver has been cloned, the resumed messenger invokes *clone* and then an *if-then-else* goal is executed: the original messenger re-attaches to the original receiver and the cloned messenger attaches to the cloned imago. The word *identical copy* refers to the "as is" semantics, that is, at the time an imago issues a *clone* predicate, it takes a snapshot (stack, messenger queue, *etc.*) to create the *identical* copy. Therefore, a cloned imago will have the same messenger queue as its original, but messengers pending in the queue are new threads representing cloned messengers.

The *$oneway_messenger* is the most basic system builtin messenger imago. It is simple and easy to understand. The overhead of its migration from host to host is only slight higher than the cost of peer to peer message communication, because the amount of its bytecode and execution stack is very small. It implements asynchronous communication between a sending imago and a receiving imago. It has the ability to automatically track down a moving receiver. Briefly, it has the intelligence to deliver a message to its receiver in a changing, dynamic mobile world.

Other system builtin messengers for *send-receive-reply*, *multicasting* and *broadcasting* can be designed in the similar pattern. Unfortunately, space does not allow for further discussion of these issues.

5 An Example

In this section, I show a possible IMAGO application which simulates a mobile agent sniffing the price changes in an imaginary *TSE_server*. For the sake of simplicity, this example is presented with assumptions of services and user interfaces. The program starts from the stationary imago *stock_monitor* which creates a worker imago with the name *sniffer* and an argument involving lists of stocks to be monitored for sale or buy, and then it waits for messengers. Upon receiving a message, the application terminates if the message indicates that the market is closed, or it displays the message otherwise.

When the *sniffer* starts execution, it moves from the home host to the *TSE_server*. Upon arriving, the *sniffer* continues execution by calling *split/2* which will examine the given argument list to determine whether a clone is necessary. If the argument involves both *Buy* and *Sale* stocks, the *sniffer* clones itself such that the original sniffs *Buy* list whereas the clone sniffs the *Sale* list.

```
/* Example: Stationary Imago */
:- begin_stationary
stock_monitor ::- create('./sniffer.ima', sniffer,
                [[s('NT', 26.00), s('RY', 43.00)], [s('SW', 53.00)]]),
        monitor.
monitor :- wait_accept(W, Msg),
        display(W, Msg),
        monitor.
display(_, complete) :- // print "market closed"
        terminate.
display(W, Msg) :- // print W and Msg
        beep.
:- end_stationary

/* Example: Worker Imago */
:- begin_worker
sniffer([Buy, Sale]) ::- move('TSE_server'),
        split(Buy, Sale).
split([], []) :- dispatch($oneway_messenger, queen, complete),
        dispose.
split([], Sale) :- !, sniff(Sale, sale).
split(Buy, []) :- !, sniff(Buy, buy).
split(Buy, Sale) :- clone(twin, R),
        R == clone →
                sniff(Sale, sale);
                sniff(Buy, buy).
sniff(L, Act):- query(L, Act),
        sleep(2000),
        sniff(L, Act).
query([], _):- !.
query([s(X, Y)|L], Act):- database('FIND PRICE', X, Y1), // assumed service
        check(X, Y, Y1, Act),
        query(L, Act).
check(_, _, Y1, _) :- var(Y1), // if unbound, market closed
        dispatch($oneway_messenger, queen, complete),
        dispose.
check(X, Y, Y1, buy) :- Y > Y1, !,
        dispatch($oneway_messenger, queen, knock(buy, X, Y1)).
check(X, Y, Y1, sale) :- Y < Y1, !,
        dispatch($oneway_messenger, queen, knock(sale, X, Y1)).
check(_, _, _).
:- end_worker
```

Now, the *sniffer* will make queries to the stock database periodically until the stock market is closed (a variable is returned to a query). For each stock listed in its argument, the *sniffer* checks if the new price is less than the user's limit. If so, an *$oneway_messenger* is dispatched to knock the stationary imago up, otherwise, the next stock will be investigated. The clone, if there is one, will do the same work as described above, except it checks for the condition on sale. Clearly, it is possible that no knock-up messengers would be dispatched if the stock prices could not meet the conditions for sale or buy.

6 Conclusion

The major feature of the IMAGO API is its novel communication model - to deploy messengers to automatically track down agents and deliver messages in a dynamic, changing world. Research on this subject involves two ongoing projects: a detailed specification of the IMAGO API and the implementation of MLVM. Although this study concentrates on the design of the IMAGO system, results will be also useful in related disciplines of network/mobile computing and functional/logic programming community.

Finally, I would like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

References

1. N. M. Karnik and A. R. Tripathi, Design Issues in Mobile-Agent Programming Systems. *IEEE Concurrency*, July-Sept., 1998, pp. 53-61.
2. D. B. Lange and M. Oshima, Programming and Deploying Java Mobile Agents with Aglets. *Addison-Wesley*, August, 1998.
3. ObjectSpace: ObjectSpace Voyager Core Package Technical Overview. *Technical Report, ObjectSpace Inc.*, 1997, http://www.objectspace.com/.
4. Odyssey. *Technical Report, General Magic Inc.*, http://www.genmagic.com/agents.
5. Concordia. *Mitsubishi Electric*, http://www.meitca.com/HSL/Projects/Concordia.
6. N. Karnik and A. Tripathi, Agent Server Architecture for the Ajanta Mobile-Agent System. *In Proc. of PDPTA'98, CSREA Press*, 1998, pp. 62-73.
7. J. E. White, Mobile Agents. *Technical Report, General Magic Inc.* , 1995.
8. D. Johansen, R. van Renesse, and F. B. Schnelder, Operating System for Mobile Agents. *In Proc. of HotOS-V'95, IEEE Computer Society Press*, 1995, pp. 42-45.
9. R. S.. Gray, Agent Tcl: A Flexible and Secure Mobile-Agent System. *In Proc. Fouth Ann. Tcl/Tk Workshop*, 1996, pp. 9-23.
10. P. Tarau, Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. *In Proc. of PAAM'99*, 1999, pp. 109-123.
11. P. van Roy *et al.*, Mobile Objects in Distributed Oz. *ACM Trans. on Programming Languages and Systems*, (19)5, 1997, pp. 805-852.
12. X. Li, Efficient Memory Management in a Merged Heap/Stack Prolog Machine. *ACM-SIGPLAN 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, 2000, pp. 245-256
13. J. Eskilson and M. Carlsson, SICStus MT - A Multithreaded Execution Environment for SICStus Prolog. *In Proc. of PLILP/ALP'98*, 1998, pp. 36-53.