IMAGO: Project Proposal

Xining Li
Department of Computing and Information Science
University of Guelph

©XINING LI 2002
Introduction

Mobile Agents are mainly intended to be used for network computing - applications distributed over large scale computer networks. The motto of Mobile Agents is: move the computations to the data rather than the data to the computations. 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, multi-threaded, 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

The final and fully developed stage of an insect after all metamorphoses, or an idealized mental picture of oneself or others.

*OXFORD Dictionary*

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 (autonomous 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 Prolog and its 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 Prolog is a simplified Prolog with extensions for Mobile Agents. The IMAGO Prolog API consists of a set of primitives that allows programmer to create mobile agent applications. Generally speaking, a mobile agents system provides primitives for agent management (creation, dispatching, migration), agent communication/synchronization, agent monitoring (query, recall, termination), etc. In logic programming languages which support concurrency and multi-programming, communication primitives take the form of message passing, remote procedure calls, or blackboard-based. For example, SICStus MT[15] uses the asynchronous message-passing mechanism whereas Bin-Prolog[10] adopts the blackboard-based model. On the other hand, IMAGO Prolog explores a novel model: instead of passing messages among agents through send/receive primitives, the IMAGO Prolog 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, will be discussed in this document, whereas some other issues, such as security, services, etc, will be investigated further.
A long term goal is to complete an Intelligent Mobile Agents System which
will be an infrastructure for a wide range of distributed applications, to examine
the possibility of integrating IMAGO with existing Mobile Agents systems, and to
explore the potential application areas, such as distributed data mining, electronic
commerce, network information retrieval, mobile monitoring and notification, etc.

2 IMAGO Prolog

IMAGO Prolog is a simplified Prolog with an extended Application Programming
Interface (API) for Intelligent Mobile Agents.

An IMAGO Prolog program consists of a set of imago definitions and module
definitions. Imago definitions serve to specify autonomous entities. An imago defi-
nition provides an implementation framework from which intelligent (mobile) agents
can be created. A procedure defined in an imago must be a complete procedure and
private to its name space. This means that such procedures are not accessible anywhere outside of the imago. Modules serve to partition the name space and support encapsulation for the purpose of constructing large applications from a library of smaller components. A module definition in IMAGO Prolog follows the specification of standard Prolog. A procedure defined in a module must be a complete procedure and accessible by imagoes or other modules through export/import mechanism. Procedure definitions that do not explicitly reside in modules or imagoes shall be prepared and gathered as the body of the built-in module named user, that is, user defined procedures which are not defined in any imago or particular module shall belong to the module user. Thus, conceptually, there are no standalone procedures in an IMAGO Prolog program. A procedure either belongs to an imago privately, or a particular module, or the default module user. In addition, procedures defined in module user are exported implicitly, i.e., accessible anywhere.

2.1 Imago Directives

In standard Prolog, module text is defined as one or more user-defined modules and
the required module user. A module consists of a single interface and zero or more
corresponding bodies. For example:

    :- module(util).
    :- export([length/2, reverse/2]).
    :- end_module(util).
    :- body(util).

    length(List, Len) :- length1(List, 0, Len).
    length1([], N, N).
    length1([H|T], N, L):-
N1 is N +1,
length1(T, N1, L).
reverse(List, Rev) :-
    rev1(List, [], Rev).
rev1([], [R, R], R).
rev1([H|T], Acc, R) :-
    rev1(T, [H|Acc], R).
:- end_body(utils).

The IMAGO Prolog introduces imago directives to specify the body of an imago. Since all procedures defined in an imago belong to the imago privately, there is no interface required. There are three kinds of imagoes: stationary imago, worker imago, and messenger imago. Corresponding, three pairs of directives are specified:

- stationary(Name)/end_stationary(Name),
- worker(Name)/end_worker(Name), and
- messenger(Name)/end_messenger(Name).

These directives are used for imago definitions, and they share a same syntax. An imago definition provides the framework (like a Java class) for creating active imagoes (like Java objects of the class). For example, the worker directive worker(buyer), where buyer is an atom giving the name of an imago definition, specifies that the Prolog text bracketed between this directive and the matching closing directive end_worker(buyer) belongs to the imago definition buyer.

:- worker(buyer).
    buyer(Arg) :-
        buyer_body, ...
:- end_worker(buyer).

The name of an imago definition serves three purposes:

- It is used to specify the beginning and termination of an imago text.
- It is a predicate indicator which gives the execution entry of the imago. From Prolog convention, a predicate indicator is a compound term P/N where P is the procedure identifier and whose arity is N. Here we define that the entry procedure is identified by the name and whose arity is 1 (an argument list). That is, there must be a procedure in the imago text which bears the name as its identifier with one argument, and this procedure is called the entry procedure. The entry procedure will be called by the IMAGO runtime system automatically. In other words, the runtime engine provides a goal toward the entry procedure after an imago text has been prepared for execution.
• It is used to form the file name of the imago bytecode file. Even though several imago definitions can be placed in a single source file, the IMAGO Prolog 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 an extension .ima), such as buyer.ima.

2.2 Imago Instances: (Mobile) Agents

Imago instances, i.e., (mobile) agents, are created from imago definitions. From now on, we simply use imago to be synonymous with imago instance. 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].

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. As there is only one stationary imago in an application, we reserve a special name queen for it. We can find the similarity that there is only one queen in a colony of bees. An imago Prolog application must contain one and only one stationary imago in its context. The following gives a minimum Imago application.

```prolog
:- stationary(foo).
   foo(_).
:- end_stationary(foo).
```

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 predicate. 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. Names (identifiers) of worker imagoes must be presented at the time they are created, and are immutable throughout execution.

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 trace 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 trace a messenger imago. However, it can move or even clone itself if necessary.

2.3 Imago API

Like other logic programming systems, IMAGO Prolog Application Programming Interface 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, imago API predicates are context sensitive, i.e., the eligibility and effect of such predicates depend on the calling context in which they are activated. In general, the usage of agent management predicates depends on the type of imagoes. Table 2.1 shows a brief list of predicates legal to each imago type. This table is far from complete, but should be sufficient to describe my project proposal.

<table>
<thead>
<tr>
<th>Imago Type</th>
<th>Builtin Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>stationary imago</td>
<td>create, accept, wait, accept, dispatch, terminate, workers</td>
</tr>
<tr>
<td>worker imago</td>
<td>move, clone, back, accept, wait, accept, dispatch, dispose</td>
</tr>
<tr>
<td>messenger imago</td>
<td>move, clone, back, attach, dispose</td>
</tr>
</tbody>
</table>

Table 2.1: Builtin Predicates for Imagos

In principle, all these predicates are not re-executable. Furthermore, they can be used in imago definitions only, that is, invocations of agent predicates are not allowed in any modules. Different kinds of errors, such as type error, resource error, system error, etc., might happen during their execution. 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 while leaving the detailed specification to Appendix A.

**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. In addition, the argument to a messenger imago is a list composed by [Receiver, Msg].

**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. A matching messenger is defined by Table 2.2. 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.

**workers(Template_id, Status, W):** This predicate is used to gather a list W of id’s of worker imagoes which match with the given Template_id and Status. This predicate can be invoked by the stationary imago only.
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 Table 2.2: it succeeds if one of the cases in the table is satisfied, or it fails/waits otherwise. A failed accepting predicate does not cause any side effect and the messenger queue remains unchanged.

<table>
<thead>
<tr>
<th>Sender</th>
<th>Msg</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>var</td>
<td>the first message</td>
</tr>
<tr>
<td>var</td>
<td>nonvar</td>
<td>the first message matching Msg</td>
</tr>
<tr>
<td>nonvar</td>
<td>var</td>
<td>the first message matching Sender</td>
</tr>
<tr>
<td>nonvar</td>
<td>nonvar</td>
<td>the first message matching both Msg and Sender</td>
</tr>
</tbody>
</table>

Table 2.2: Behavior of Accepting Predicates

Based on our discussion, Fig. 1 summarizes the state transition of imagoes. From the transition figures 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 waitAccept 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 resumed when its receiver evaluates one of the following predicates: move, clone, back, dispose, and accept if the messenger matches.

2.4 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 oneway_messenger. It is worth to note that this name is the imago definition name, rather than the imago instance name, because messenger imagoes are anonymous.

:- messenger(oneway_messenger).
oneway_messenger([Receiver, Msg]);- // the entry procedure
deliver(Receiver, Msg).
deliver(Receiver, Msg);-
Figure 1: Imago State Transition
attach(Receiver, Msg, Result),
check(Receiver, Msg, Result).
check(_, _ received):- !,
   dispose.
check(_, _ deceased):- !,
   dispose.
check(Receiver, Msg, cloned(Clone)):- !,
   clone(_, R),
   R == clone ->
      deliver(Clone, Msg);
   deliver(Receiver, Msg).
check(Receiver, Msg, moved(Server)):- !,
   move(Server), !,
   deliver(Receiver, Msg).
check(Receiver, Msg, _):- // if move/1 fails
   deliver(Receiver, Msg). // try to attach again
:- end._messenger(oneway._messenger).

When the oneway._messenger is started, it tries to attach itself to the given receiver. Only two possible cases make the attach succeeds 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 trace a moving receiver. Briefly, it has the intelligence to deliver a message to its receiver in a changing, dynamic mobile world.

Another commonly used communication scheme is the *send-receive-reply*. In this scheme, the sender is expecting a reply, i.e., it could be blocked until the receiver has completed the processing of the message and replied back with a result. We could use *oneway_messenger* to simulate this scheme by the following steps:

1. the sender dispatches a messenger, and wait a messenger from the receiver,
2. the messenger disposes after its carried message has been received,
3. the receiver handles the message and generates results,
4. the receiver dispatches another messenger to deliver the result, and
5. the sender accepts the result and the messenger disposes.

Although this simulation works, it requires two messenger imagoes being deployed to carry out the send-receive-reply sequence. Can we use just one messenger to do the same work? The next example gives a *roundtrip_messenger* imago which implements asynchronous round-trip communication. This scheme is similar to the Aglet’s *Future-Type Messaging*. Namely, the sender does not need to block its execution, instead, it is free to do anything until it is willing to accept the reply. However, if the sender does want a pure synchronous communication, it can issue a *wait_accept* immediately after dispatching a *roundtrip_messenger*.

In order to make a round trip, the messenger must be able to change its role: the initial trip on behalf of the sender to deliver the message and the return trip on behalf of the receiver to deliver the reply.

```prolog
:- messenger(roundtrip_messenger).
roundtrip_messenger([Receiver, Msg]):-
    deliver(Receiver, Msg, initial_trip).
deliver(Receiver, Msg, Type):-
```

13
attach(Receiver, Msg, Result),
check(Receiver, Msg, Type, Result).
check(Receiver, initial_trip, received):- !, // message received
deliver(Receiver, Msg, replying). // deliver a var Msg for holding reply
check(Receiver, Msg, replying, received):- !, // reply constructed
sender(Sender), // switch the role: sender becomes receiver
reset_sender(Receiver), // receiver becomes new sender
deliver(Sender, Msg, return_trip).
check(_, return_trip, received):- !, // reply received
dispose.
check(Receiver, Msg, Type, cloned(Clone)):- !,
clone(_, R),
R == clone ->
deliver(Clone, Msg, Type);
deliver(Receiver, Msg, Type).
check(_, deceased):- !,
dispose.
check(Receiver, Msg, Type, moved(Server)):-
move(Server), !,
deliver(Receiver, Msg, Type).
check(Receiver, Msg, Type, _):-
deliver(Receiver, Msg, Type).
:- end_messenger(roundtrip_messenger).

To achieve the send-receive-reply behavior, the roundtrip_messenger should make three rounds of attachment. The first attachment is to deliver the initial_trip message. After that message has been received, the second attachment builds up a variable in order to hold the replying message. This is the unique feature in Prolog that an argument could be used as either input or output. Therefore, this variable message will be matched (unified) with the receiver’s reply. As soon as the replying message has been constructed, the messenger changes its role by calling the swap_sender, and then delivers the return_trip message, i.e., the reply. During the process of constructing a reply, if the receiver moves, clones, etc, the roundtrip_messenger takes the corresponding action to trail the receiver. A new predicate is used in the roundtrip_messenger: swap_sender(S, R). Since every messenger implicitly carries its sender’s name, swap_sender(Sender, Receiver) will copy the sender’s name to Sender and use Receiver to replace the sender’s name, as if the messenger was dispatched by the Receiver.
2.5 Application Examples

In this section, I show three possible IMAGO applications. For the sake of simplicity, the examples are presented with assumptions of services and user interfaces.

The first example simulates a mobile agent sniffing the price changes in an imaginary TSE_server. In this program, the stationary imago creates a worker imago with the name sniffer and a list argument which gives 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.

/* Example 1: Stationary Imago */
:- stationary(stock_monitor).
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(stock_monitor).

/* Example 1: Worker Imago */

:- worker(sniffer).
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(Stock, Limit)|L], Act):-
    database('SELECT PRICE', Stock, Value), // assumed service
    check(Stock, Limit, Value, Act),
    query(L, Act).
check(_, _, Value, _):-
    var(Value), !, // if unbound, market closed
    dispatch(oneway_messenger, queen, complete),
    dispose.
check(Stock, Limit, Value, buy) :-
    Limit > Value, !,
    dispatch(oneway_messenger, queen, knock(buy, Stock, Value)).
check(Stock, Limit, Value, sale) :-
    Limit < Y1, !,
    dispatch(oneway_messenger, queen, knock(sale, Stock, Value)).
check(_, _, _). // otherwise, take no action
:- end_worker(sniffer).

When the sniffer starts execution, it moves from the home host to the TSE_server. After moving, 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.

Now, the sniffer will make queries to the stack 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.
Our second example demonstrates the usage of the `$roundtrip_messenger$. In this program, the stationary imago creates a worker imago, dispatches a `$roundtrip_messenger$ for each listed stock in its query list and then waits for a reply. It is worth to note that each stock symbol is accompanied with the name of its market server, and the worker is thus requested to move to the destined server for making the stock price query. When the query list runs out, the stationary imago launches an `$oneway_messenger$ notifying the worker to complete.

/* Example 2: Stationary Imago */
:- stationary(stock_query).
stock_query( ) :-
    create(./callee ima', agent, )
    query([["NT", 'TSE_server'], ['LU', 'NYSE_server']].
query([]) :-
    dispatch(oneway_messenger, agent, finish),
    terminate.
query([S,L]) :-
    dispatch(roundtrip_messenger, agent, S),
    wait_accept(agent, Price),
    display(S, Price),
    query(L).
    display([Stock, Server], Price) :-
        // print Stock at Server market costs Price
        beep.
:- end_stationary(stock_query).

The worker imago evaluates oncall to wait for a messenger from its queen. If the incoming messenger delivers finish, then the worker completes its job. Otherwise, the incoming messenger must be a roundtrip_messenger which not only delivers the stock symbol to be investigated but also will carry the reply back to the queen. Thus, the worker first moves to the designed server, then makes a query to the stock database, and issues the reply. From our earlier discussion, this reply is in fact a wait_accept call which will match the variable message and resume the second attachment of the roundtrip_messenger. As soon as one stock has been processed, the worker starts its next round of iteration.

/* Example 2: Worker Imago */
:- worker(callee).
callee( ) :-
    oncall.
    oncall :-
        wait_accept(queen, Msg),
check(Msg),
oncall.
check(finish) :-
    dispose.
check([Stock, Server]) :-
    move(Server),
    database('SELECT PRICE', Stock, Price), // assumed service
    reply(queen, Price).
reply(R, M) :- wait_accept(R, M).
:- end_worker(callee).

In the above examples, the system predicate terminate not only stops the execution of the stationary imago, but also eliminates all worker/messenger imagoes of the application if they are still alive on remote hosts.

Our last example shows a user-designed messenger for multicasting. The stationary_multicaster will deliver a message to all worker imagoes whose name match with that in the given Receiver - a partially instantiated term. For example, suppose that a set of N workers has been created by the stationary imago, where each worker has a name node(i), 1 ≤ i ≤ N. Thus the stationary imago may issue a goal:

```prolog
...,
dispatch('stationary_multicaster.ima', node(_), hello),
...,
```

to send hello to all node(i)'s. Other workers, if any and if their names do not match with node(_), will not see this message. As the matter of fact, the stationary_multicaster may also be used as a broadcaster simply by naming the receiver as a void variable, i.e., a wild name which matches with any existing imago's name. It is worth noting that the stationary_multicaster can only be launched by the stationary imago, because it a user-defined messenger.

/* Example 3: A User-defined Multicasting Messenger */

```prolog
:- messenger(stationary_multicaster).
stationary_multicaster([Receiver, Msg]) :-
    multicast(Receiver, Msg).
multicast(Receiver, Msg) :-
    workers(Receiver, alive, L),
    spawn(L, Msg).
spawn([], _) :-
    dispose.
```
spawn([Receiver], Msg) :- !,
    deliver(Receiver, Msg).
spawn([Receiver | L], Msg) :-
    clone( _, R),
    R == clone ->
    deliver(Receiver, Msg);
    spawn(L, Msg).
deliver(Receiver, Msg):-
    attach(Receiver, Msg, Result),
    check(Receiver, Msg, Result).
check(_, received):- !,
    dispose.
check(Receiver, Msg, cloned(Clone)) :- !,
    clone( _, R),
    R == clone ->
    deliver(Clone, Msg);
    deliver(Receiver, Msg).
check(_, deceased):- !,
    dispose.
check(Receiver, Msg, moved(Server)):-
    move(Server), !,
    deliver(Receiver, Msg).
check(Receiver, Msg, _):-
    deliver(Receiver, Msg).
:- end_messenger(stationary_multicaster).

In the body of stationary_multicaster, predicate workers(N, alive, L) is called which will return a list L containing all worker imago names matching with N and being currently alive.

2.6 System-builtin Messengers

Clearly, the concept of messengers offers flexibility to simulate different patterns of agent communication. For example, we can define a postman messenger to deliver mails to the addressed users (agents), we may also design a paperboy messenger to dispatch a copy of message (like a copy of newspaper) to different subscribers. Those messengers follow the basic pattern of the oneway_messenger in design with a slight difference. As an example, we can have a system-defined multicaster messenger just by slightly modifying the stationary_multicaster in previous section, and thus this system messenger can be dispatched by any worker at anywhere.
:- messenger(multicasting_messenger).
multicasting_messenger([Receiver, Msg]):-
   back, // go back to stationary server
   multicast(Receiver, Msg).
multicast(Receiver, Msg):-
   // the same as the user-defined multicaster
:- end_messenger(multicasting_messenger).

In this section, I list possible system messengers and discuss their behavior and usage. The discussion is based upon the assumption that all imagoes participating in the communication are correctly named (addressed) and there is no error occurred during execution.

oneway_messenger:
   • The sender dispatches an oneway_messenger with argument [R, M] and continues its execution.
   • The messenger attaches itself to receiver R.
   • The receiver receives the message M which makes the messenger resumed.
   • The messenger disposes itself.

croundtrip_messenger:
   • The sender dispatches a roundtrip_messenger with argument [R, M].
   • The sender then waits for a reply from R.
   • The messenger attaches itself to receiver R.
   • The receiver receives the message M which makes the messenger resumed.
   • The messenger re-attaches itself to R with a variable M1.
   • The receiver binds M1 with a reply which makes the messenger resumed again.
   • The messenger changes its role and attaches itself to the sender.
   • The sender receives the reply M1 which resumes the messenger.
   • The messenger disposes itself.

multicasting_messenger:
   • The sender dispatches a multicasting_messenger with argument [P, M] and continues its execution.
• The messenger moves to the stationary server and requests for a list of receivers which match the given name P (partially bound or a variable for broadcasting).
• The messenger then clones itself for each receiver $R_i$ in the list and leaves the last receiver for itself.
• Each (cloned) messenger attaches itself to receiver $R_i$.
• The receiver receives the message M which makes the messenger resumed.
• The messenger disposes itself.

**postman_messenger:**

• The sender dispatches a *postman_messenger* with argument [Rlist, Mlist] and continues its execution.
• The messenger attaches itself to the first receiver in Rlist.
• The receiver receives the first message in Mlist which makes the messenger resumed.
• The messenger then attaches itself to the second receiver in Rlist.
• The receiver receives the second message in Mlist which makes the messenger resumed.
• Repeat this attach-receive process until Rlist is exhausted.
• The messenger disposes itself.

**paperboy_messenger:**

• The sender dispatches a *paperboy_messenger* with argument [Rlist, M] and continues its execution.
• The messenger attaches itself to the first receiver in Rlist.
• The receiver receives message M which makes the messenger resumed.
• The messenger then attaches itself to the second receiver in Rlist.
• The receiver receives message M which makes the messenger resumed.
• Repeat this attach-receive process until Rlist is exhausted.
• The messenger disposes itself.

**collecting_messenger:**

• The sender dispatches a *collecting_messenger* with argument [Rlist, Vlist] and waits for a messenger whose sender is itself.
• The messenger attaches itself to the first receiver in Rlist.
• The receiver binds a message with the first variable in Vlist and makes the messenger resumed.
• The messenger then attaches itself to the second receiver in Rlist.
• The receiver binds a message with the second variable in Vlist and makes the messenger resumed.
• Repeat this attach-receive process until Rlist is exhausted.
• The messenger then attaches itself to the original sender.
• The sender receives the returned Vlist and makes the messenger resumed.
• The messenger disposes itself.
3 MLVM Design and Implementation

The IMAGO system is an infrastructure that implements the agent paradigm. An IMAGO server resides at a host machine intending to host imagoes (mobile agents) 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. Fig. 2 gives a graphic view of an IMAGO server (the daemon is not presented).

The MLVM provides a framework for implementing imago servers. The IMAGO project requires two kinds of servers: stationary server and mobile imago server.

A stationary server hosts the stationary imago as well as worker/messenger imagoes from the same application. Namely, imagoes from one application are strictly forbidden to enter the stationary server invoked by another application. A stationary server co-exists with its invoking application, i.e., it terminates when its hosted application completes. This kind of server should provide rich services to allow the stationary imago accessing local resources. In addition, it is the default Imago Name Server for its hosted application.

A mobile imago server (or just imago server) may host worker/messenger imagoes
of different applications. Like a web server, an imago server must have either a well
known name or a name searchable through the internet name binder. Also it must
be active all the time to host incoming imagoes. This kind of servers should provide
services for network computing, or interfaces to other internet servers, such as www
servers, database servers, etc.

In this section, we will discuss the major issues of designing imago servers. As
the framework of MLVM is originated from the LVM, most implementation details
can be found in [19], and only new issues will be addressed in this document.

3.1 Multithreading and thread-level mobility

The LVM is a sequential logic virtual machine for Prolog-like languages. To accom-
modate the LVM to the mobile agents paradigm, we must introduce concurrency
into the LVM model. Concurrent processes are the basic computational units in
most operating systems. They are asynchronous and each has its own logical ad-
dress space. The concept of process blends the concepts of resource ownership and
current execution together. Modern systems tries to distinguish these potentially
independent concepts, and thus has led to the development of a computational con-
cept known as the thread.

Multithreading refers to the ability of a system to support multiple threads of
execution within a single process. In this paradigm, each concurrent process behaves
like a virtual machine that supports concurrent execution of threads, forming the
second level concurrency within the same address space.

A MLVM-based imago server is a multi-threading process. Each MLVM-based
server behaves like a virtual machine that supports concurrent execution of imagoes.
An imago is represented by a thread which is a lightweight process that runs strictly
sequentially.

In designing a MLVM emulator, it is a critical factor to determine the stra-
egy of multi-threading implementation. Generally speaking, there are two possible
schemes:

Kernel level multithreading: kernel level multithreading has the advantage that
blocking and scheduling of threads are treated with greater flexibility and effi-
ciency. For example, threads can be preempted easily, a thread issuing a sys-
tem call can be blocked without blocking its host process, and each thread can
get a fair share of processor time. Solaris is an instance which introduces the
c kernel level multithreading called a Light Weight Process (LWP). LWP’s are
recognized by the Solaris kernel as the basic unit that can be scheduled. The
POSIX application programming interface (API) is another example which
provides preemptive operating system level threads. However, the shortcom-
ings are that the lighter context switching overhead of threads is lost, and this
paradigm does not fit with the idea that mobile agent servers should be able to run on different operating systems.

**User level multithreading:** thread packages implemented as a software layer in the user space are straightforward and portable without kernel modification. The key issues of this paradigm are how to handle blocking system calls from a thread and how to schedule threads for execution in a process. For example, the run time procedure must trap blocking system calls to prevent blocking all threads in the process, and the processor time allocated to this process must be multiplexed among existing threads. The context switching of threads requires very little overhead because it involves saving and restoring only the program counter and stack pointers.

I intend to explore user level multithreading because switching threads is at least at an order of magnitude faster than trapping to the kernel. However, in the absence of clock interrupts, user level multithreading has the difficulty of distributing cpu time among runnable threads. In other words, if a thread starts running, no other threads in the MLVM process will ever run unless the running thread voluntarily gives up the cpu. There are four possible solutions to solve this problem:

- MLVM requests a clock signal and captures this signal to switch threads.
- MLVM counts procedure calls of each running thread and uses a certain call limit to switch threads.
- MLVM preempts cpu from a thread at the point of garbage collection.
- Or let a thread run and believe that most threads will block often.

At this moment, I have not decided which solution or a combination of solutions to use. The above possible solutions will be investigated further and be experimented in the MLVM implementation.

### 3.2 Memory Management

Memory management is a central issue in the design of the MLVM. It play a determining role not only in shaping multithreading strategy, but also in establishing computation mobile policy. The merged stack/heap architecture of the MLVM makes it much easier and simpler to control autonomous agent threads and implement agent migration by capturing execution state at thread-level. Thus, the destination host can reactivate the thread at precisely the point where migration was initiated. A good memory management scheme should be efficient all the time and should use little extra space for its own bookkeeping. Unfortunately no such
scheme is known for the general problem of dynamic memory allocation. We compromise and use simple strategies to handle memory expanding/contracting as well as garbage collection, hoping that would be efficient enough in most cases.

The MLVM allocates two memory blocks for each thread: a program block and an environment block. The memory regions for these blocks does not need to be contiguous. The MLVM does not provide the programmer or the user control over the initial sizes of these blocks, instead, they are dynamically expanded or possibly contracted as required by the computation. During execution, a thread might need more program space to cope with dynamically generated clauses or symbolic constants, or more environment space to hold dynamic data. Further, for a Mobile Agents application, a thread might move to a remote host and continue execution from the moving point. Thus, both blocks should be able to be expanded (or possibly contracted) during run time, or to be moved together to another (remote) virtual machine and then be restored at different block spaces. The expandability and mobility of a thread require efficient algorithms to relocate addresses from old to new.

The program block of a thread contains program code and a symbol table. The MLVM separates the program block from the environment block because it exhibits different expansion and management patterns. First, this block is not subject to garbage collection. Secondly, it is unlikely to be expanded as often as the environment block. The code portion might be expanded only if the directive dynamic/1 occurs in the program text. A symbol hash table is used for symbolic resolution - a unique entity (offset) produced by the initial resolution of an atom entry will be shared by all the later resolutions of the same atom entry. The symbol table and its associated hash table might be expanded if new atoms are created during execution (through input or atom constructing predicates). The basic rule is that all existing atoms will remain in their original offsets to the symbol table base after expansion. However, they must be rehashed with respect to an expanded hash table.

The management algorithms and strategies associated with the environment block is critical to the system performance. The environment block is controlled by two crucial pointers: the current trail top and the current stack top, where the trail is used to save pointers for backtracking and garbage collection, and the stack contains all kinds of data during execution. The environment block is exhausted if these two pointers threaten to cross. Clearly, monitoring the growth of both pointers must be very expensive. We only check the threaten to cross condition at a few points - stack allocation instructions. When an stack allocation instruction is executed, the MLVM emulator will first check if a round of garbage collection can be invoked and thus come up with the following cases:

1. no GC and two pointers are not threaten to cross: allocate the frame and continue.
2. no GC, but two pointers are threaten to cross: expand the environment block.

3. GC and have enough copying area: do garbage collection.

4. GC but no enough copying area: expand the environment block and do GC.

The environment block needs to be expanded in cases 2 and 4. Generally speaking, there are two common ways of dynamic memory expanding: Unix dynamic memory allocation functions, such as malloc, realloc and free, and Unix memory mapping functions, such as mmap, mremap and munmap. The former scheme is adopted by XSB [16] - a Prolog emulator, and the latter is used in SableVM [17] - a Java virtual machine. The latter scheme has some advantages over the former. For example, using mremap to replace realloc may save many copying operations when the memory block needs to be relocated somewhere to grow. Unfortunately, mremap is not supported by some platforms. Thus we borrow ideas from XSB for environment expansion. In brief, a small environment block is allocated through malloc to a new born thread, the block might be expanded by realloc dynamically to meet the needs of the thread, and finally, the block will be returned to the system pool at the thread death.

If there is a reasonable amount of accumulated garbage being detected, an inline CGC function will be invoked. CGC is mainly a generational copying collector[12]. However, it has new features different from the traditional generational collectors[18]. First, CGC introduces a concept of chronological generation - a dynamic way to divide generations. The size of a generation is bound by the machine cache size, and the number of generations vary from program to program. Second, CGC controls the frequency of collector invocations by capturing the continuation point (an approximation of local minima of stack) and a factor of cache size. Therefore it collects garbage (in most cases) incrementally with a trivial pause time. Third, most generational collectors promote survivors from the youngest generation into old generations. On the other hand, CGC discards the young generations, and makes the survivors temporarily tenured as a part of the generation which becomes the youngest after collection. Contracting only applies to the environment block. A possible point to reduce the size of the environment block is a big chunk of memory being reclaimed after a round of garbage collection.

3.3 Name Resolution

There are two layers of naming services: the server layer and the application layer. To contact a server, the server must be identifiable. Generally speaking, three possible ways could be considered:

- Identify by symbolic name of a server.
• Identify by either URL or physical address.

• Or, identify by service that the servers provide.

Although I have not decided yet which scheme to use for defining or looking for servers, I intend to accept the commonly used method such as those adopted for web servers, i.e., identifying servers by URL’s (the Uniform Resource Locator). In addition, I intend to use the method proposed by the Aglet project, namely, this URL should provide the host and domain names plus the protocol (Imago Transfer Protocol) to be used for transferring the imago over the network.

On the other hand, each agent must have a unique name so that its owner and other agents can communicate with it over the networks. Some systems, such as Distributed Oz and Voyager, adopt location-transparent names at the application level. Systems such as Aglets and Agent Tcl, assign location-dependent names. For example, an aglet is associated with a unique identifier so that every alget in the network can be uniquely addressed by combining its identifier with its context URL.

I believe, however, that identifying an agent by the combination of its identifier and its location does not suit well with the mobility of agents. As an agent may move any time to an arbitrary host server, making the location uncertain. Therefore, I intend to use the location-transparent scheme because it not only avoids the conflict between fixed locations and moving agents, but also allows easier agent programming and portability of agent applications.

The current IMAGO design adopts a closed world naming scheme to identify imagoes. An identifier is bound to each worker imago and it is unique in the application (a closed world) where the imago is created and immutable throughout the lifetime of the imago. In other words, imagoes among different applications can not see (identify) each other. Should we leave a door to the open world is to be investigated, because it is quite difficult to avoid name clashes in an open world.

From the discussion of IMAGO API, we can find that there is no direct control to a worker imago except creation and termination. That is, a worker imago can not be forced to move, clone, back, or dispose unless it is willing to do so. The only way that stationary/worker imagoes cooperate with each other is by the means of dispatching messengers. Therefore, messengers are responsible to look for moving worker imagoes.

In order to locate a moving imago, imago servers must maintain enough information. However, it is virtually impossible to have the precise information of a changing world, because an application may involve imagoes which are moving or cloning all the time. To cope with such a dynamic configuration, the IMAGO system adopts two levels of imago name look-up, and maintains heuristic information through registration and logging.

The stationary server is the default imago name server. It maintains an entry for each worker current alive in its application. An entry is created when an imago
is created or cloned and eliminated when the imago is disposed. A new born worker, no matter by creating or cloning and no matter born from the stationary host or a remote host, will register its birth place automatically. A deceased worker will also notify its death information to the stationary server. In addition, the stationary server will keep its local logging information. For example, an entry is updated when the imago is moved from the stationary host to a new remote host.

On the other hand, an imago server maintains its local logging information only. An entry is created for a worker imago when it enters this server and modified when it moves out of this server. This entry will eventually be deleted when the imago disposes itself or its associated application terminates.

Searching for an imago is done by server’s lookup facility. First, a local search takes place. It will consult the local logging information to find where the imago possibly resides in. If the local search fails, the IMAGO system conducts the second level name search, i.e., a system message is send to the imago’s stationary server (the default imago name server of this application) asking for where the imago is possibly located. The reason of using “possibly” is that the information recorded in a registration or a logging table is heuristic. There is no guarantee that the imago in search is still working at the location returned by lookup, because an imago (except the stationary one) has no absolute host address - it is free to move to any legal host at any time. However, it is guaranteed that successive lookup’s at subsequent heuristic hosts will eventually trap the imago which is being searched for. This is why a messenger will invoke attach each time it moves to a new place. A messenger claims that “I can track the receiver down provided I have the trail of the receiver”, whereas the IMAGO lookup says that “the location I found is where most likely the receiver resides at, or at least the receiver has lived”. In other words, the IMAGO’s lookup facility provides the trail of the receiving imago while leaves the tracking-down job to the messenger.

3.4 Security issues

Mobile agent security is an open research area. Several security related problems, such as agent privacy and integrity, authentication, authorization, and access control, are to be investigated.

3.5 Services

An agent server must provide services to its hosted agents. One scheme is to offer a generic service by exporting a parametric interface which facilitates a large spectrum of requirements from all the potential agents. Another approach is to provide a set of primitives which can be invoked by agents for a desired service. This issue will be studied further.
4 Conclusion

Appendix A

For each IMAGO predicate, we try to follow Prolog Standard document to describe its definition (except item 6 which is specific for IMAGO system).

1 Description: Assuming no errors,
   • The logic condition for the built-in predicate to be true; and
   • A procedural description of what happens as a goal is executed and whether the goal succeeds or fails.

2 Template and modes: A specification for both the type of arguments and the mode of arguments. The mode is one of the following:
   +: instantiated,
   ?: either instantiated or a variable,
   -: a variable
   @: remaining unaltered.

3 Errors: An error is a special event which causes the normal process of execution to be interrupted (how to handle an error is implementation dependent).

4 Examples: A predication executing the built-in predicate as a goal, together with a statement saying whether the goal succeeds or fails or there is an error.

5 Bootstrapped builtin predicates: One or more bootstrapped built-in predicates, if any, are defined as special cases of a more general built-in predicate.

6 Eligibility: Imago predicates are context sensitive. This is specification of which type of imagees is eligible to invoke the predicate. There are three types of imagees: stationary, worker and messenger.

IMAGO Predicates

1. create/3

1.1 Description:
   • create(Worker_file, Imago_id, Argument) is true.
   • procedurally, create(Worker_file, Imago_id, Argument) is executed as follows:
a) Loads the worker imago specified by Worker_file,
b) Allocates a new thread to execute the worker imago,
c) Sets up the Imago_id and initial Argument,
d) Inserts this new thread into the ready queue,
e) The goal succeeds.

1.2 Template and modes:
create(@source_sink, +nonvar, ?term)

1.3 Errors:

a) Work_file is a variable: instantiation_error
b) Imago_id is a variable: instantiation_error
c) The file specified by Worker_file does not exist or cannot be opened: i/o_error
d) Imago_id is a duplicate (the same imago already exists): ** this seems a hard problem: how to check this?
e) The file specified by Worker_file is not an imago (worker) program: imago_type_error
f) No more resource for a new thread: out_of_space_error

1.4 Examples:
create(’./callee.ima’, w1, [123, abc]).
Succeeds.
// It loads the imago file ’./callee.ima’, allocates a new imago thread,
// assigns this thread with identifier w1, passes the argument list
// [123, abc] to this thread, and inserts this thread into the ready queue.

1.5 Bootstrapped predicates:
create(Worker_file, Imago_id :-
create(Worker_file, Imago_id, _).

1.6 Eligibility:
Stationary imago only

2. dispatch/3

2.1 Description:
• dispatch(Messenger_file, Receiver_id, Msg) is true.
• procedurally, dispatch(Messenger_file, Receiver_id, Msg) is executed as follows:
  a) Loads the messenger imago specified by Messenger_file. There are two cases: internal file such as $oneway_messenger, or external file such as './my_messenger.imaf',
  b) Allocates a new thread to execute the messenger imago,
  c) Sets up sender_id (the caller's id), Receiver_id, Msg,
  d) Inserts this new thread into the ready queue,
  e) The goal succeeds.

2.2 Template and modes:
   dispatch(@source_sink, +nonvar, ?term)

2.3 Errors:
   a) Messenger_file is a variable: instantiation_error
   b) Receiver_id is a variable: instantiation_error
   c) The file specified by Messenger_file does not exist or cannot be opened: i/o_error
   d) A worker imago tries to dispatch an external Messenger_file: eligibility_error
   e) The file specified by Messenger_file is not an imago (messenger) program: imago_type_error
   f) No more resource for a new thread: out_of_space_error

2.4 Examples:
   dispatch($oneway_messenger, sniffer, nt(23.2)).
   Succeeds.
   // It spawns a new messenger thread from the internal file $oneway_messenger
   // asking the messenger to deliver message nt(23.2) to imago sniffer

   dispatch($oneway_messenger, sniffer, _).
   Succeeds.
   // It spawns a new messenger thread from the internal file $oneway_messenger
   // asking the messenger to synchronize with imago sniffer

32
dispatch('./my_messenger.im', node(1), foo).
Succeeds.
// It spawns a new messenger thread from the external file './my_messenger.im'
// asking the messenger to deliver message foo to imago node(1)

2.5 Bootstrapped predicates:

no

2.6 Eligibility:

A worker imago dispatches internal (system defined) messengers only. Stationary imago may dispatch messengers defined either internally or externally.

3. attach/3

3.1 Description:

- attach(Receiver_id, Msg, Result) is true.
- procedurally, attach(Receiver_id, Msg, Result) is executed as follows:
  a) Searches for the imago identified by Receiver_id through its server’s log,
  b) If the receiver is not found at the current host, instantiates Result to ’moved(queen)’, the goal succeeds.
  c) If the receiver is found in the ready queue at the current host, then proceeds to 3.1.e),
  d) Else if the receiver is found currently blocked (waiting for a messenger), removes the receiver from the waiting queue and inserts the receiver into the ready queue,
  e) Deactivates the caller, and appends the caller to the receiver’s messenger queue. As soon as the caller has been attached to the receiving imago, its thread is suspended.
  f) The goal will succeed when the caller is resumed by its receiver with one of the following Result instantiations:
     (i) ’moved(Server)’ if the receiver has moved to another Server, or
     (ii) ’deceased’ if the receiver is dead, or
     (iii) ’received’ if the receiver has received the message carried by the caller, or
     (iv) ’cloned(Clone)’ if the receiver has cloned itself such that the new cloned imago is identified as Clone.
3.2 Template and modes:
   attach(+nonvar, ?term, -term)

3.3 Errors:
   a) The imago identified by Receiver_id does not exist: imago_id_error
   b) Receiver_id is a variable: instantiation_error

3.4 Examples:
   attach(node(1), date('April', 20), R).
   Succeeds.
   // This is the most complicated predicate in IMAGO API.
   // It tries to deliver the message date(april, 20) to receiver node(1).
   // However, the receiver might generate one of the following events: it
   // could move to another host, it could clone itself, it could receive
   // the messenger, or it could terminate its execution. If any event occurs,
   // R is bound to the corresponding event value, and the attach/3 is considered
   // to succeed.

3.5 Bootstrapped predicates:
   no.

3.6 Eligibility:
   Messenger imagoes only.

4. move/1

4.1 Description:
   • move(Server_id) is true iff the calling imago has been transferred to the
     remote server specified by Server_id and ready to continue its execution.
   • procedurally, move(Server_id) is executed as follows:
     a) Deactivates the caller and captures its state,
     b) If there are pending messengers in the caller’s messenger queue, all
        these suspended messengers will be resumed and 'moved(Server_id)'
        will be instantiated to the Result variable of each blocked attach/3
        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.)

c) Transmits it to the given remote server specified by Server_id,
d) If the caller is ready to continue at the new host, the goal succeeds.
e) Else the goal fails at the host the caller resided at before moving.

4.2 Template and modes:

move(@destination_sink)

4.3 Errors:

a) Server_id is a variable: instantiation error
b) Server identified by Server_id does not exist: existence_error
c) Server rejects the moving imago: security_error, or ...

NOTE: This predicate might cause side effects, because it resumes all attached messengers. Suppose the predicate fails upon some transmission failure, such as gateway error, server shutdown, etc. Consider the following code:

```prolog
p([]).
p([S|L]):- move(S), !,
    do_something, // if move/1 succeeds
    p(L).
P([|L]):- p(L). // if move/1 fails
```

This will allow an imago to try working on a list of possible servers. However, it causes a serious problem for its attached messengers (if any): they might terminate with a failure. A solution to this problem is that messengers must be designed to cope with failure of move/1, i.e., they must redo attach/3 upon failure of move/1.

All system messengers should be designed to cope with this problem. How to handle this in a user-designed messenger is out of our hand.

4.4 Examples:

move(http://imago-s1.uoguelph.ca)
// If the identified server accepted this moving imago,
// Succeeds, the control goes to the next goal at the new server.
// If the identified server is not alive or some other reasons
// Fails, the imago backtracks at the original server.

4.5 Bootstrapped predicates:
   back :- stationary_host(S),
       move(S).

4.6 Eligibility:
   Worker and Messenger imagoes.

5. clone/2

5.1 Description:
   • clone(Imago_id, Result) is true.
   • procedurally, clone(Imago_id, Result) is executed as follows:
      a) Allocates a new thread to execute the cloned imago,
      b) Duplicates the caller’s entire state to this new thread,
      c) Sets up this new thread as Imago_id,
      d) Binds the caller’s Result to ‘origin’ and the clone’s Result to ‘clone’,
      e) Insert this new thread into the ready queue with a program counter to the next goal,
      f) If the caller is a worker and there are pending messengers in its messenger queue, all these suspended messengers will be resumed and the term ’cloned(Imago_id’ will be instantiated to the Result of each blocked attach/3 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),
      g) The goal succeeds.

5.2 Template and modes:
   clone(-term)
   clone(+nonvar, -term)

5.3 Errors:
   a) Imago_id is a variable: instantiation_error
   b) Imago_id is a duplicate (the same imago already exists): ** this seems a hard problem: how to check this?

36
c) no more resource for a new thread: out_of_space_error

5.4 Examples:

clone(R).
Succeeds.
// called by a messenger to clone an anonymous messenger

clone(twin, R).
Succeeds.
// the caller clones itself and name the clone as ’twin’

5.5 Bootstrapped predicates:

clone(Result) :- clone(anonymous, Result).

5.6 Eligibility:

Worker and messenger imagoes.

6. accept/2

6.1 Description:

- accept(Sender_id, Msg) is true iff both Sender and Msg unify with the sender’s id and the message carried by a messenger in the current messenger queue.
- procedurally, accept(Sender_id, Msg) is executed as follows:
  a) Let H be the pointer to the caller’s associated messenger queue.
  b) If the queue is exhausted (H→nil), the goal fails.
  c) Else let M be the messenger pointed by H,
  d) If Sender_id and Msg unify with M’s sender_id and M’s message,
  e) Removes M from the messenger queue, instantiates M’s Result (attach/3) to ’received’ and inserts M into ready queue,
  f) The goal succeeds,
  g) Else advances H to the next messenger, then proceeds to 6.1.b).

6.2 Template and modes:

accept(\(?\)term)
accept(\(?\)term, \(?\)term)
6.3 Errors:
   no.

6.4 Examples:

accept(M).
   // If the contents of current messenger queue is
   // [M1(node(1), foo), M2(node(5), bar), ...]
   // Succeeds, unifying M with 'foo' and the current messenger queue
   // is left as [M2(node(5), bar), ...]

accept(S, M).
   // If the contents of current messenger queue is
   // [M1(node(1), foo), M2(node(5), bar), ...]
   // Succeeds, unifying S with 'node(1), M with 'foo' and the current
   // messenger queue is left as [M2(node(5), bar), ...]

accept(node(5), M).
   // If the contents of current messenger queue is
   // [M1(node(1), foo), M2(node(5), bar), ...]
   // Succeeds, unifying M with 'bar' and the current
   // messenger queue is left as [M1(node(1), foo), ...]

accept(node(5), foo).
   // If the contents of current messenger queue is
   // [M1(node(1), foo), M2(node(5), bar)]
   // Fails, the current messenger queue is unchanged

6.5 Bootstrapped predicates:
   accept(Msg) :-
      accept(\_, Msg).

6.6 Eligibility:
   Stationary and worker imagoes.
7. wait_accept/2

7.1 Description:

- wait_accept(Sender_id, Msg) is true.
- procedurally, wait_accept(Sender_id, Msg) is executed as follows:
  a) Let H be the pointer to the caller’s associated messenger queue.
  b) If the queue is exhausted (H→nil),
  c) Rolls back the caller’s program counter to the beginning of this goal, such that it seems the goal has not been executed yet,
  d) Deactivates the caller’s thread, inserts it into the waiting queue, In this case, we say that the wait_accept/2 predicate is blocked.
  e) Else let M be the messenger pointed by H,
  f) If Sender_id and Msg unify with M’s sender_id and M’s message,
  g) Removes M from the messenger queue, instantiates M’s Result (attach/3) to ’received’ and inserts M into ready queue,
  h) The goal succeeds,
  i) Else advances H to the next messenger, then proceeds to 7.1.b).

7.2 Template and modes:

wait_accept(?term)
wait_accept(?term, ?term)

7.3 Errors:

no.

7.4 Examples:

wait_accept(M).
// If the contents of current messenger queue is
// [M1(node(1), foo), M2(node(5), bar), ...]
// Succeeds, unifying M with ’foo’ and the current messenger queue
// is left as [M2(node(5), bar), ...]

wait_accept(S, M).
// If the contents of current messenger queue is
// [M1(node(1), foo), M2(node(5), bar), ...]
// Succeeds, unifying S with 'node(1), M with 'foo' and the current
// messenger queue is left as [M2(node(5), bar), ...]

wait_accept(node(5), M).
// If the contents of current messenger queue is
// [M1(node(1), foo), M2(node(5), bar), ...]
// Succeeds, unifying M with 'bar' and the current
// messenger queue is left as [M1(node(1), foo), ...]

wait_accept(node(5), foo).
// If the contents of current messenger queue is
// [M1(node(1), foo), M2(node(5), bar)]
// The goal is blocked, the current messenger queue is unchanged

7.5 Bootstrapped predicates:
   wait_accept(Msg) :-
       wait_accept(_, Msg).

7.6 Eligibility:
   Stationary and worker imagoes.

8. dispose/0

8.1 Description:
   • dispose is true.
   • procedurally, dispose is executed as follows:
     a) All the pending messengers in caller’s messenger queue, if any, will be
        resumed with their Result bound to ‘deceased’,
     b) Deallocates the caller’s thread and modifies the server’s log,
     c) The goal succeeds.

8.2 Template and modes:
   dispose

8.3 Errors:
   no.

40
8.4 Examples:
   no.

8.5 Bootstrapped predicates:
   no.

8.6 Eligibility:
   Worker and messenger imagoes.

9. terminate/0

9.1 Description:
   - terminate is true.
   - procedurally, terminate is executed as follows:
     a) Sends system messages to all involved servers to eliminate all imagoes
        spawned (cloned, created, or dispatched) from this application,
     b) Terminates the caller’s thread.
     c) The goal succeeds.

9.2 Template and modes:
   terminate

9.3 Errors:
   no.

9.4 Examples:
   no.

9.5 Bootstrapped predicates:
   no.

9.6 Eligibility:
   Stationary imago only.

10. current_host/1

10.1 Description:
   - current_host(H) is true iff H unifies with the name of the current host.
   - procedurally, current_host(H) is executed as follows:
a) If H is a variable, instantiates H to the name of current host and the
goal succeeds.
b) Else if H unifies with the name of the current host, the goal succeeds.
c) Else the goal fails.

10.2 Template and modes:
   current_host(?term)

10.3 Errors:
   no.

10.4 Examples:
   no.

10.5 Bootstrapped predicates:
   no.

10.6 Eligibility:
   All imagoes.

11. stationary_host/1

11.1 Description:
   • stationary_host(H) is true iff H unifies with the name of the stationary
     host.
   • procedurally, stationary_host(H) is executed as follows:
     a) If H is a variable, instantiates H to the name of stationary host and
        the goal succeeds.
     b) Else if H unifies with the name of the stationary host, the goal suc-
        ceeds.
     c) Else the goal fails.

11.2 Template and modes:
   stationary_host(?host_name)

11.3 Errors:
   no.

11.4 Examples:
   no.
11.5 Bootstrapped predicates:
   no.

11.6 Eligibility:
   All imagoes.

12. workers/3

12.1 Description:
   - workers(Template, Status, Worker_list) is true.
   - procedurally, workers(Template, Status, Worker_list) is executed as follows:
     a) Let H be the pointer to the registration log of the current server.
     b) If the log is exhausted, *i.e.*, H→nil,
     c) the goal succeeds.
     d) Else let W be the worker’s registration record pointed by H,
     e) If Template and Status match with W’s name and current status,
     f) Inserts W into Worker_list.
     g) Advances H to the next registration record, proceeds to 12.1.b).

12.2 Template and modes:
   workers(?term, +atom, -var)

12.3 Errors:
   a) Status is not an legal status (see Note 2): domain_error
   b) Worker_list is not a variable: type_error

12.4 Examples:
   workers(_,_ W).
   Succeeds.
   // W is the list of all imago names which are recorded in the current server
   and belong to the same application of the caller.

   worker(node(_), alive, W).
   Succeeds.
// W is the list of imago names in the form of node(_) and they are alive at
the current server and are from the same application as the caller.

12.5 Bootstrapped predicates:

no.

12.6 Eligibility:

All imagoes.

Note(1): a status is an atom which defines in a call of workers/3 the
current status that a matching worker should be included in the output list.
The current version recognizes the following status:

- alive: in search of workers which are still alive (but could be blocked);
- deceased: in search of workers which are terminated;
- all: in search of workers which belong to this application.

Note (2): although this predicate can be invoked by any imago at any-
where, only the stationary server holds the complete registration log, while
other remote servers may hold partial logging records. Thus, there is no guar-
antee that the returning list includes all matching imago names if the predicate
is not executed at the stationary host.

13. sender/1

13.1 Description:

- sender(Sender) is true.
- procedurally, sender(Sender) is executed as follows:
  a) Bind Sender to the sender’s identifier (carried by this messenger),
  b) The goal succeeds.

13.2 Template and modes:

sender(-var)

13.3 Errors:

a) Sender is not a variable: type_error

13.4 Examples:

no.
13.5 Bootstrapped predicates:
   no.

13.6 Eligibility:
   Messengers only.

14. reset_sender/1

14.1 Description:
   - reset_sender(New_sender) is true.
   - procedurally, reset_sender(New_sender) is executed as follows:
     a) Replace the hiding sender's identifier to New_sender,
     b) The goal succeeds.

14.2 Template and modes:
   reset_sender(+atom)

14.3 Errors:
   a) New_sender is a variable: instantiation_error

14.4 Examples:
   no.

14.5 Bootstrapped predicates:
   no.

14.6 Eligibility:
   Messengers only.
Appendix B

In this section, a table is given which defines the eligibility of built-in predicates with respect to different imago types. We use section numbers of standard Prolog for each group of built-in predicates. There are possible cases to define eligibility:

**NO:** not eligible

**YES:** eligible

**TBD:** to be decided

<table>
<thead>
<tr>
<th>Group</th>
<th>Stationary</th>
<th>Worker</th>
<th>Messenger</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2 Term Unification</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.3 Type Testing</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.4 Term Comparison</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.5 Term Creation</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.6 Arithmetic</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.7 Arithmetic Comparison</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.8 Clause Retrieval</td>
<td>TBD</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.9 Clause Creation</td>
<td>TBD</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.10 All Solution</td>
<td>YES</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>8.11 Stream Selection</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.12 Char I/O</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.13 Byte I/O</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.14 Term I/O</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>8.15 Logic &amp; Control</td>
<td>YES</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>8.16 Atomic Term Processing</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>8.17 Implementation Defined</td>
<td>YES</td>
<td>TBD</td>
<td>TBD</td>
</tr>
</tbody>
</table>

Table 1: Eligibility of built-in predicates

References


46


