

Distributed coordination with MESSENGERS

Munehiro Fukuda, Lubomir F. Bic*, Michael B. Dillencourt,
Fehmina Merchant

Department of Information and Computer Science, University of California, Irvine, CA 92717, USA

Abstract

MESSENGERS is a paradigm for the programming of distributed systems. It is based on the principles of autonomous messages, called Messengers, which carry their own behavior in the form of a program. This enables them to navigate freely in the underlying computational network, communicate with one another, and invoke compiled node-resident C functions in the nodes they visit. Hence, a distributed application is viewed as a collection of C functions whose invocation and interoperation is orchestrated by Messengers. This provides for a clear separation between computations, as expressed by the individual node functions, and coordination, which is the order of function invocations and the transport of information among them as prescribed by Messengers. This separation allows each layer to be designed and implemented separately. It also supports the reuse of the coordination structures and the interactive and incremental development and use of distributed applications. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Coordination; Mobile agents; Navigation; Distributed Computing; Simulation

1. Coordination and the MESSENGERS model

The notion of separating concurrent programming into computation and coordination was introduced in [5]. Coordination includes the synchronization, communication, and creation/destruction of computational activities required to orchestrate individual computations into a coherent system.

One approach to coordination utilizes *channel-based* communication. Processes communicate directly with each other by reading from and writing to *ports*. Ports of different processes are connected to each other via *channels*. This approach leads to a clean separation of computation and coordination functions. A process performs its own computation by computing values to be written to its output ports from values read from its input ports. The coordination requirements of a process, such as connecting its ports to ports of other processes through channels, are handled by other processes.

* Corresponding author. E-mail: bic@ics.uci.edu.

Examples of the channel-based approach are the IWIM model [1] and the ConCoord programming environment [12].

Another approach to coordination is *medium-based* coordination. At a very abstract level, all medium-based systems work on the same principle. There is a common medium, or state space, shared by the processes. Processes can modify the state space, and these modifications affect the behavior of other processes. Computation is performed by the processes, and coordination is achieved through the shared state space.

One of the most prominent examples of the medium-based approach is Gamma [2], based on a chemical reaction metaphor. The state space is a multiset of objects. Gamma programs consist of matched pairs of *reaction conditions* and *actions*. Execution proceeds by replacing a collection of objects that satisfy a reaction condition by the result of applying the corresponding action. As programs are executed, they may cause multiset transformations that create the reaction conditions necessary to allow other programs to execute, thus advancing the computation.

Another well-known example of coordination through a shared state space is the Linda system [5]. The state space is a pool of data called a *tuple space*. Processes may insert, read, and remove tuples from the tuple space using various primitives. They may also spawn new activities that leave new tuples in the tuple space upon their termination. Processes select tuples associatively, by issuing requests for tuples that match certain templates.

The MESSENGERS coordination model, the subject of this paper, is also based on an underlying shared state space through which processes communicate. The underlying state space contains *both* functions and data. Processes may read or write data and dynamically incorporate functions into their behavior. The shared space is structured by being mapped onto a logical network. Every data item or function in the shared state space resides at some node in this network.

Individual processes, called *Messengers*¹ have state information, consisting of local variables and control data. Messengers are able to navigate freely and autonomously through the logical network. They migrate to nodes containing data that they wish to read, write, or modify, and to nodes containing functions that they wish to invoke.

The MESSENGERS coordination model is similar to Linda in that both provide a structured global state space, shared by all the processes. In Linda, the state space is implicitly structured by tuple contents and the associative pattern matching. In MESSENGERS, the state space is explicitly partitioned by its mapping onto the logical network, and this structuring is supported by the navigational features described in Section 2. Thus the MESSENGERS model is most similar to variants of Linda, such as PoliS [6], that support explicit partitioning of the tuple space into multiple tuple spaces. We will say more about this later in the paper (Section 4.1), after we have described MESSENGERS in more detail.

¹ We use small caps (MESSENGERS) to denote the entire system, and mixed case (Messengers) to denote the individual processes.

A major distinguishing characteristic between Messengers and other coordination models is the nature of the programming task, particularly insofar as it involves specifying coordination. In models based on coupled coordination, the coordination specification is *structural*, primarily concerned with which pairs of processes are directly communicating. In models based on a shared state space, the programmer's primary view may be either *declarative* (i.e., concerned primarily with specifying which data objects should interact and the nature of the interaction) or *operational* (i.e., concerned primarily with the evolution of the state space from its initial configuration to its final configuration). In MESSENGERS, the programmer's view is *navigational*, concerned with how to move through the logical network, and when to interact with the data or functions stored at a node. This paradigm has a number of advantages over other approaches at both the programming and the implementation levels. We will discuss these in Section 3, after having made the navigational view of coordination concrete by describing the main principles of MESSENGERS. Section 4 then discusses related research to put the present work in a larger context.

2. The MESSENGERS paradigm

2.1. Basic philosophy of MESSENGERS

MESSENGERS is a coordination paradigm for distributed systems based on the concept of autonomous messages [3], which we refer to as Messenger to differentiate them from ordinary passive messages. A Messenger is a self-contained object, consisting of a program and current state, including a program counter and local variables. The program, which is a combination of interpreted MESSENGERS code and native C code, describes the Messenger's behavior, which allows it to navigate autonomously through the underlying communication network and carry out computation in the nodes it visits.

MESSENGERS is based on the following three fundamental concepts:

- *Navigational autonomy* – A computation under MESSENGERS involves the dynamic construction of a logical network that is mapped onto the underlying physical network. The construction (and possible subsequent destruction) of this logical network is performed by one or more Messengers. Any Messenger is also capable of navigating through the logical network, replicating itself by following multiple links, and deciding what actions to take at each node. The commands that permit the construction/destruction of and the movement through the local network are based on a navigational calculus, described in Section 2.3.
- *Dynamic composition* – In addition to constructing and moving around the logical network, each Messenger is capable of invoking and controlling the execution of ordinary C functions accessible on various nodes visited by the Messenger. Hence, each application may be viewed as a collection of ordinary C functions, whose execution (i.e. invocation, termination, synchronization, and

communication) is coordinated in both time and space by Messengers propagating autonomously through the computational network.

- *Temporal and spatial interaction* – Concurrent Messengers can coordinate their activities through a collection of specialized synchronization and communication mechanisms in both time and space to accomplish their mission.

We note that none of the above three concepts is unique to MESSENGERS. There is, however, no system that uses all three together for the sole purpose of application coordination. Notably, autonomous navigation is used by a number of systems to create “mobile agents” (see Section 4). While their primary purpose is to roam the Internet, MESSENGERS uses navigation as a primary structuring tool, which, in conjunction with dynamic functional composition, yields a powerful coordination paradigm for general-purpose distributed computing.

The following sections elaborate on the above basic principles of navigation, dynamic composition, and coordination.

2.2. MESSENGERS architecture

The MESSENGERS system is implemented as a collection of daemons instantiated on all physical nodes participating in the distributed computation. The daemon’s task is to continuously receive Messengers injected from the user interface or arriving from other daemons, to interpret the Messengers’ behaviors described by their scripts, and to send them on to their next destinations as appropriate. Thus, the MESSENGERS system involves three levels of networks:

1. *Physical network*: This is the lowest level network (a LAN or WAN), which constitutes the underlying computational resource.
2. *Daemon network*: This network is superimposed over the physical layer. Each daemon node is a UNIX process running a MESSENGERS language interpreter. The daemon network topology is described by the user at system’s initialization. This involves selecting an arbitrary subset of the physical nodes to run the daemon process on (which may be done automatically by the system based on current loads) and specifying arbitrary inter-daemon links. Each such link is mapped onto a path of one or more links in the physical network and is used to guide the mapping of logical links and nodes at run time. The main purpose of the daemon network is to provide a virtual computational resource, thus isolating the applications from the details of the physical network topology.
3. *Logical network*: This is an application-specific computation network created on top of the daemon network. Multiple logical network nodes may be created on the same daemon network nodes, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology. It is this network that is used by Messengers when navigating through the system.

The general organization of the MESSENGERS interpreter is as follows. It alternates between two phases: interpretation of Messengers code and native-mode function execution. During the first phase, it makes a complete pass through the ready queue

of Messengers and continues interpreting each until one of the following occurs: the Messenger terminates; the Messenger executes a navigational statement causing it to move to another node; or the Messenger invokes a compiled C function using the *func()* or *exec()* statements as discussed below. Hence, the interpretation of a Messenger is serialized between any two navigational or function-invocation statements, thus automatically enforcing a critical section. During the second phase, the interpreter calls all functions that have been scheduled as “future events” thus far. The above two phases are repeated until one of the following occurs: both the Messengers ready queue and the function scheduling queue are empty; an interval timer expires; or the local virtual time proceeds beyond a certain threshold. At that point, the interpreter enters an inter-daemon communication phase, which physically exchanges Messengers between workstations using Unix sockets. Thereafter, the interpreter continues with the Messengers interpretation phase as described above.

2.3. The navigational calculus

Movement of a Messenger through the logical network as well as the creation and destruction of this network is expressed in terms of a navigational calculus. We first define this calculus using an abstract network, N , where each node n has a user-defined name and a system-wide unique ID (called address); each link also has a user-defined name and an orientation. Names and orientations are optional. If omitted, they are considered blank (empty).

Within such an abstract network, the navigational calculus is based on a single operation, called *FIND_DEST*, which, given a network N , a current node c , and a specification of destinations *node_dest*, finds a set of destination nodes

$$DEST = FIND_DEST(dest_specs).$$

That is, *DEST* is the subset of all nodes $n \in N$ that match the destination specifications, or $DEST = \{n \mid match(n, dest_specs)\}$.

The destination specifications are described using BNF notation as shown in Fig. 1. Each is a triple, consisting of a node name, link name, and link orientation (direction). The node can be explicitly named (using an address, a variable, or a constant). The symbol “*” denotes a “wild card”, i.e., a symbol that matches any name. The symbol “.” denotes “this node”, i.e., the current node. The link can be specified using a variable, a constant, or “*”. The symbol “**” denotes a “virtual link”, i.e., it pretends that the network is fully connected and hence the current node c is connected to all other nodes in the network. The symbols “+”, “−”, and “*” used for *dir* denote the link direction: “forward”, “backward”, or “any” (i.e., forward, backward, or undirected).

The semantics of the function “match”, which determines whether a node n satisfies the destinations specifications, is as follows: it returns true if (1) the node’s name or address matches the specified name or address and (2) the node n is connected to the current node c via a link that matches the specified link name and direction.

$$\begin{aligned}
\text{dest_specs} &= (\text{node}, \text{link}, \text{dir}) \\
\text{node} &= \text{addr} \mid \text{variable} \mid \text{constant} \mid * \mid . \\
\text{link} &= \text{variable} \mid \text{constant} \mid * \mid ** \\
\text{dir} &= + \mid - \mid *
\end{aligned}$$

Fig. 1. BNF notation for Navigational Calculus destination specifications.

Table 1

The meanings of the possible *node/link/dir* parameter combinations in the MESSENGERS navigational calculus

	node	link	dir	Set of destination nodes selected
(1)	n_i	l_i	d_i	All nodes n_i connected to c by link l_i in direction d_i
(2)	*	l_i	d_i	All nodes connected to c by link l_i in direction d_i
(3)	n_i	*	d_i	All nodes n_i connected to c in direction d_i
(4)	*	*	d_i	All nodes connected to c in direction d_i
(5)	n_i	**	d/c	All nodes n_i (other than c)
(6)	*	**	d/c	All nodes (other than c)
(7)	.	d/c	d/c	Current node c

Operationally, the above semantics may be expressed as follows: From the current node c , follow all links that match the specification. That is, in the case of a name, follow the named link; in the case of a “*”, follow all incident links; in the case of “**”, hop directly to all nodes in the network. In each of the nodes reached, determine if it matches the specified name (including “.”) or address. This yields a (possibly empty) set of destination nodes.

Table 1 lists all possible combinations of the above *node/link/dir* parameters, together with their intuitive meaning. The symbol n_i denotes a specific node name, i.e., a variable, a constant, or an address. Similarly, l_i denotes a specific link name, i.e., a variable or a constant, and d_i denotes the link’s direction. The symbol “d/c” denotes a “don’t care” condition.

The above navigational calculus is the basis for navigating through the *logical* network as well as for the construction of this network, which requires a form of navigation through the *daemon* network.

2.4. Language specification

MESSENGERS is based on the principles of autonomous objects navigating through a network and coordinating the ongoing computation. Specifically, individual Messengers must be capable of creating and subsequently destroying logical nodes and links and map these onto the underlying daemon network. They also must be capable of navigating through the logical network using a variety of criteria and replicating themselves as necessary. In this section we describe MESSENGERS-C, the programming language

in which individual Messengers programs, called *scripts*, are written. As the name suggests, MESSENGERS-C is based on C. Its most important extensions fall into three categories as outlined in Section 2.1: (1) navigational statements to permit Messengers to create, destroy, and navigate in the logical network; (2) an interface to permit Messengers to dynamically load and invoke native-mode C functions; and (3) constructs to support interaction in time and space.

2.4.1. Navigation

A Messenger may create new logical links and nodes, change or delete existing ones, and move arbitrarily through the network by following links or jumping to specific nodes. The navigational calculus presented in Section 2.3 permits one to specify a set of destination nodes relative to a given node and is the basis for a potentially wide spectrum of possible navigational statements. In this section we present three specific statements, *create*, *hop*, and *delete*, which reflect the current MESSENGERS implementation and have been shown to be powerful enough to express a wide range of tasks.

The *hop* Statement. The *hop* statement permits a Messenger to move around the logical network and is based directly on the navigational calculus. Its syntax is as follows:

$$\text{hop}(ln = n_1, \dots, n_k; ll = l_1, \dots, l_k; ldir = d_1, \dots, d_k)$$

where each triple (n_i, l_i, d_i) is a destination specification in the logical network, as defined by the navigational calculus (Fig. 1).

The semantics of *hop* are then as follows: for each destination specification (n_i, l_i, d_i) determine $DEST_i = FIND_DEST(n_i, l_i, d_i)$; for each destination node $n \in DEST_i$, replicate the current Messenger and send it to n ; the original Messenger in the current node c , i.e., the one executing the *hop* statement, ceases to exist.

Examples. Fig. 2 shows a portion of a logical network, consisting of a current node c and several other nodes connected to c by named or unnamed logical links. The number inside each of the node circles represents its unique system-wide address. Dotted links represent virtual links, i.e., nodes 7 and 8 are not directly connected to node 0.

Table 2 shows a series of *hop* statements with different combinations of parameters. Assuming a Messenger would execute these statements in node 0 of the logical network

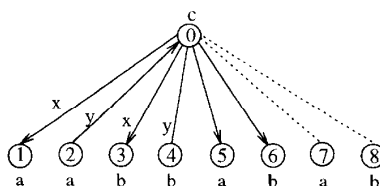


Fig. 2. Navigating in a logical network.

Table 2

Examples of the effect of various *hop* parameter combinations in the logical network of Fig. 2

<i>hop</i> statement	Destinations	Explanation
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = <i>x</i> ; <i>ldir</i> = *)	1	Nodes <i>a</i> connected by link <i>x</i>
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = *; <i>ldir</i> = *)	1, 2, 5	Nodes <i>a</i> connected by any link
<i>hop</i> (<i>ln</i> = *; <i>ll</i> = <i>x</i> ; <i>ldir</i> = *)	1, 3	Nodes connected by link <i>x</i>
<i>hop</i> (<i>ln</i> = *; <i>ll</i> = *; <i>ldir</i> = *)	1–6	All neighbor nodes
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = <i>x</i> ; <i>ldir</i> = +)	1	Nodes <i>a</i> connected by forward link <i>x</i>
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = *; <i>ldir</i> = +)	1, 5	Nodes <i>a</i> connected by any forward link
<i>hop</i> (<i>ln</i> = *; <i>ll</i> = <i>x</i> ; <i>ldir</i> = +)	1, 3	Nodes connected by forward link <i>x</i>
<i>hop</i> (<i>ln</i> = *; <i>ll</i> = *; <i>ldir</i> = +)	1, 3, 5, 6	Nodes connected by any forward link
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = <i>x</i> ; <i>ldir</i> = –)	none	Nodes <i>a</i> connected by backward link <i>x</i>
<i>hop</i> (<i>ln</i> = <i>a</i> ; <i>ll</i> = *; <i>ldir</i> = *)	1, 2, 5, 7	Nodes <i>a</i> (connected or disconnected)
<i>hop</i> (<i>ln</i> = *; <i>ll</i> = *; <i>ldir</i> = *)	1–8	All Nodes (connected or disconnected)
<i>hop</i> (<i>ln</i> = <i>a</i> , <i>b</i> ; <i>ll</i> = <i>x</i> , <i>y</i> ; <i>ldir</i> = *, *)	1, 4	Nodes <i>a</i> connected by link <i>x</i> and nodes <i>b</i> connected by link <i>y</i>
<i>hop</i> (<i>ln</i> = <i>a</i> , <i>b</i> ; <i>ll</i> = *, <i>y</i> ; <i>ldir</i> = *, *)	1, 2, 4, 5	Nodes <i>a</i> connected by any link and nodes <i>b</i> connected by link <i>y</i>
<i>hop</i> (<i>ln</i> = <i>a</i> , <i>b</i> ; <i>ll</i> = *, *; <i>ldir</i> = –, –)	2	Nodes <i>a</i> and <i>b</i> connected by back link

(Fig. 2), it would be replicated and *hop* to zero or more other nodes. The middle column of the table (labeled “destinations”) lists the nodes that would be reached by the given *hop* statement.

Defaults. The defaults for the *hop* parameters are as follows:

n_i	l_i	d_i	ln	ll	$ldir$
*	*	*	*, ..., *	*, ..., *	*, ..., *

That means that any node, link, or direction specification of the form “..., *, ...” is equivalent to “*, ..., *”. When any of the parameters *ln*, *ll*, or *ldir* has the form “*, ..., *”, it may be omitted. Semantically, this implies that the selection of target nodes considers all neighbors of the current node, i.e., those connected by any logical link.

Examples. *hop*() $\equiv$ *hop*(*ln* = *) $\equiv$ *hop*(*ln* = *; *ll* = *) $\equiv$ *hop*(*ln* = *; *ll* = *; *ldir* = *)
Hop to neighboring logical nodes along all incident links regardless of their direction.

The create Statement. The *create* statement permits a Messenger to create new logical nodes and/or links. The selection of the daemon node(s) on which the new logical node(s) will be mapped is based on the navigational calculus (Section 2.3). The syntax is as follows:

$$create(ln = n_1, \dots, n_k; ll = l_1, \dots, l_k; ldir = d_1, \dots, d_k;$$

$$dn = N_1, \dots, N_k; dl = L_1, \dots, L_k; ddir = D_1, \dots, D_k)$$

where each triple (n_i, l_i, d_i) specifies a *new* logical node, n_i , connected to the current node by a (possibly directed) link, l_i . The node n_i is created on the daemon node specified by the triple (N_i, L_i, D_i), which is a destination specification in the daemon network, as defined by the navigational calculus (see Fig. 1). The possible options for

the specification of the new logical node (n_i, l_i, d_i) are as follows:

$$\begin{aligned} n_i &= \text{variable} \mid \text{constant} \mid \sim \\ l_i &= \text{variable} \mid \text{constant} \mid \sim \mid ** \\ d_i &= + \mid - \mid \sim \end{aligned}$$

The semantics of *create* are then as follows: For each destination specification (N_i, L_i, D_i) determine the set of daemon nodes $DEST_i = FIND_DEST(N_i, L_i, D_i)$. From this (possibly empty) set of potential destinations, choose one and create the logical node n_i on it. n_i can be named (using a variable or constant) or unnamed ($\sim$), and can be connected to the current node c by the link l_i , which could be named (using a variable or constant) or unnamed ($\sim$). This link may be directed ($+$ or $-$) or undirected ($\sim$). When l_i is specified as “**”, the link is virtual, i.e., the new node n_i is not connected to c . In all cases, the Messenger executing the *create* statement moves to the newly created node n_i . When no node was created (because the set $DEST_i$ of potential daemon node destinations was empty), the Messenger ceases to exist.

Note that, unlike the *hop* statement where *all* nodes of the destination set are chosen, the *create* statement only chooses *one*. The rules for selecting a single daemon node $N \in DEST_i$ are not specified as part of the statement or the navigational calculus. This is to give different implementations the freedom to provide different rules based on their specific needs, for example, to support non-determinism or various fairness policies in mapping.

Examples. Assume that a Messenger is executing in a logical node c mapped onto a daemon node C . The following series of *create* statements shows the creation of a new logical node a connected to c by an undirected link x .

- *create*($ln = a; ll = x; ldir = \sim; dn = .$)
create a on the current daemon C ; (dl and $ddir$ are ignored and hence need not be specified)
- *create*($ln = a; ll = x; ldir = \sim; dn = N1; dl = *; ddir = *$)
create a on a neighboring daemon $N1$, i.e., an $N1$ connected to C by any daemon link
- *create*($ln = a; ll = x; ldir = \sim; dn = *; dl = *; ddir = *$)
create a on any neighboring daemon, i.e., one connected to C by any daemon link
- *create*($ln = a; ll = x; ldir = \sim; dn = *; dl = **$)
create a on any daemon, i.e., connected or disconnected ($ddir$ is ignored)

The following statements create two logical nodes, a and b , connected to c by undirected links x and y , respectively.

- *create*($ln = a, b; ll = x, y; ldir = \sim, \sim; dn = *, *; dl = L1, L2; ddir = *, *$)
create a on neighboring daemon connected by $L1$ and b on neighboring daemon connected by $L2$
- *create*($ln = a, b; ll = x, y; ldir = \sim, \sim; dn = ., *; dl = *, *; ddir = *, *$)
create a on current daemon C and b on a neighboring daemon

Duplicates. The *create* statement checks for the existence of the nodes and links it is to create. If neither the node nor the link exists, it creates both as specified above. If the node exists, it only creates the link as specified and hops to the node. If both the node and the link exist, it only hops to the node. Note that duplicate nodes and/or link names may exist in general but the following situations are prevented: (1) two nodes cannot be connected to each other by indistinguishable links, i.e., links carrying the same name and direction, and (2) two nodes with the same name cannot be connected to a common node using indistinguishable links.

Defaults. The defaults for the *create* parameters are as follows:

n_i	l_i	d_i	ln	ll	$ldir$
$\sim$	$\sim$	$\sim$	$\sim, \dots, \sim$	$\sim, \dots, \sim$	$\sim, \dots, \sim$

The above means that any logical node, link, or direction specification of the form “ $\dots, \sim, \dots$ ” is equivalent to “ $\dots, \dots$ ”. When any of the parameters ln , ll , or $ldir$ has the form “ $\sim, \dots, \sim$ ”, it may be omitted. Semantically, this implies that, unless a name or direction is explicitly specified, the newly created node is unnamed and connected by an unnamed and undirected link to the current node c .

N_i	L_i	D_i	dn	dl	$ddir$
*	*	*	$*, \dots, *$	$*, \dots, *$	$*, \dots, *$

Similar to *hop*, the above means that the search for potential target daemons considers all neighbors of the current daemon, i.e., those connected by any daemon link.

Examples. The following two *create* statements are equivalent; each creates a new unnamed logical node connected to the current node by an unnamed link. The new node is created on one of the neighboring daemons:

$$create() \equiv create(ln = \sim; ll = \sim; ldir = \sim; dn = *; dl = *; ddir = *)$$

The following two *create* statements are also equivalent, each creating a new node a not connected to the current node by any logical link; a is created on a neighboring daemon:

$$create(ln = a; ll = ***) = create(ln = a; ll = ***; dn = *; dl = *; ddir = *)$$

The delete Statement. The *delete* statement is similar to *hop* in that it permits a Messenger to move around the logical network. Unlike *hop*, however, *delete* also removes logical links and nodes as it traverses the network. Its syntax is analogous to *hop*:

$$delete(ln = n_1, \dots, n_k; ll = l_1, \dots, l_k; ldir = d_1, \dots, d_k)$$

where each triple (n_i, l_i, d_i) is a destination specification defined as for the *hop* statement.

The semantics of *delete* are then as follows: For each destination specification (n_i, l_i, d_i) determine $DEST = FIND_DEST(n_i, l_i, d_i)$. For each destination node

$n \in DEST_i$, replicate the current Messenger, send it to n , and remove the corresponding link. The original Messenger in the current node c , i.e., the one executing the *delete* statement, ceases to exist. If after all links have been removed, c becomes a singleton, it is also deleted.

Defaults. The defaults for the delete parameters are the same as for *hop*.

Examples. The following statement deletes all links directed away from the current node. If there are no incoming (or undirected) links, the node becomes a singleton and is also deleted.

*delete(lidir = +) $\equiv$ delete(ln = *, ll = *, lidir = +)*

2.4.2. Computation

MESSENGERS distinguishes three types of variables, referred to as *messenger*, *node*, and *network* variables. Messenger variables are private to and carried by each Messenger as it propagates through the logical computational network. Node variables are resident in nodes and shared by all Messengers currently running on the same logical node. Network variables are predefined at each logical node and give each Messenger access to the network information local to the current node, (i.e. the current logical and daemon node's address and name, the number of logical and daemon links incident on the current node, and the last traversed logical and daemon link's name.)

MESSENGERS-C supports all of the common assignment and control statements provided by C, except those involving pointers. Specifically, a Messenger residing in a given logical node c may: (1) read and update c 's node variables; (2) read and update its own messenger variables; and (3) read c 's network variables. Arbitrary expressions are permissible in the assignment. A Messenger may also perform all the common control statements supported in C, including if-else, while, do-while, break, and exit. These may access the same types of variables as assignment statements.

In addition to the above general computation statements, all of which are interpreted by the MESSENGERS daemons, a Messenger script may include statements to dynamically link and invoke C functions, precompiled and executed in native-mode on the current node. This permits the behavior of each Messenger, and hence the functionality of the entire application, to be composed dynamically.

The following two statements provide an interface to executable C functions via the Unix file system:

1. *exec(file = filename [; in = argument] [; out = argument;] [nowait])*: The parameter *filename* is a Unix path name that specifies a file containing an executable C program. This must contain a *main()* function, which is spawned as a separate Unix process, i.e., operating concurrently with the invoking Messenger. The arguments specified by *in* are passed to the new process at the time of creation and results are returned to the invoking Messenger via the *out* argument. If *nowait* is specified, the process may continue operating even after the parent Messenger has left the node (or has terminated). With this option, no results are returned to the invoking Messenger.

2. *func(file=filename; name=func_name [; in=argument] [; out=argument])*: Similar to *exec*, the parameter *filename* is a Unix path name specifying a file. This must contain one or more executable C functions (but not necessarily a complete program with a *main()* function). The function specified by the parameter *func_name* is invoked and the arguments specified by *in* are passed to it. Unlike with *exec*, the function executes as part of the current Messenger, that it, as part of the MESSENGERS interpreter process. The invoking Messenger is blocked until the function terminates and returns its results to the Messenger script via the *out* argument.

Note that the Unix path name *filename* is resolved relative to the file system accessible by the current node. The loading of the function is triggered dynamically when it is invoked for the first time. Instead of loading a function from the file system, a compiled function image can also be carried by a Messenger at run time by encapsulating it into a character array of a Messenger variable. This can then be invoked by jumping to this data area once the Messenger has reached its desired destination node.

2.5. Interaction

To accomplish their mission, Messengers must generally interact with one another in both time and space. Spatial interaction is accomplished via shared node variables defined in every logical node. Every Messenger script contains a node variables declaration that uses the same style and conventions as C. This provides the necessary mapping between the Messenger code and the actual node-resident variables. Using the declared names, a Messenger can read and write any of the node variables of the node in which it currently resides. Consequently, any collection of Messengers can interact with each other by hopping to a common node and reading/writing agreed-upon node variables.

To support temporal interaction, MESSENGERS provide support for *virtual time*, i.e., a global time line consistent over all physical nodes [16]. Conceptually, this is a globally shared variable that simulates the passage of time. It allows any Messenger to suspend itself until a certain point in the virtual time has been reached. This is accomplished using two possible functions:

- *m_sched_time_dlt(delta_time)* suspends the calling Messenger for a virtual time interval specified in *delta_time*.
- *m_sched_time_abs(abs_time)* suspends the Messenger until an absolute virtual time *abs_time*.

The virtual time is advanced to the next scheduled event whenever all events at the current time have been completed and hence all Messengers have suspended themselves to await the arrival of their respective points in virtual time.

MESSENGERS supports both conservative and optimistic approaches to virtual time maintenance, which require periodic exchanges of timing information among all participating daemons, followed by possible roll-backs of local computations, to guarantee that the virtual time is consistent across all nodes [16, 8].

3. Advantages of the navigational paradigm

In this section, we address the advantages of the navigational paradigm. We first show how MESSENGERS can be used to construct a Toxicology simulation model, and we discuss the advantages of this approach. We then discuss some of the other advantages of using MESSENGERS as a coordination paradigm (Section 3.3).

3.1. A coordination example

We illustrate how a distributed simulation can be structured as a collection of functions, residing on different nodes of a network, where the invocation of these functions and the exchange of data among them is orchestrated by Messengers. Here we consider a class of biomedical simulations where the organism is modeled as a circulatory system, consisting of boxes, each representing a distinct organ or a group of organs having similar physiological characteristics. These are then interconnected by directed links, which represent the flow of fluids, in particular, blood. The information exchanged along the links depends on the particular application. For example, in a cardio-vascular simulation, values such as blood pressure and volume would be exchanged [19]. In Toxicology, research has produced a number of useful models that allow the distribution and metabolism of various chemical substances (e.g., drugs or environmental toxins) to be simulated.

Fig. 3 shows a specific Toxicology simulation model that we have implemented in collaboration with UCI's College of Medicine [21, 10]. Under this model, the lung node consists of two compartments – the alveolar space represents the area into which outside air is inhaled. Oxygen as well as other chemical compounds are then passed to the lung blood compartment and from there they are carried to all the other organ groups in the body. Each organ group retains some portion of the chemicals based on their size, the volume of incoming blood, and their tissue-specific physiological constants. The concentration of the chemical in the blood, which also plays an important role in determining how much of the chemical is retained in a given organ, is a variable and must be carried between the organs according to the blood flow. *C_{art}* in Fig. 3 represents the arterial concentration of the chemical being modeled. Each organ changes this concentration based on its specific parameters and passes it along its corresponding *C_{ven}* link back to the lung, where the individual values are accumulated to form the new arterial concentration.

Some organs, notably the liver, also carry out metabolic functions, which may change some portion of the original chemical compounds into other chemicals. For example, styrene, which is inhaled from the air is metabolized into styrene oxide and hence the model must be able to account for multiple chemicals circulating through the body and continuously changing from one to another.

All of the above processes, i.e., the metabolism and the retention of chemicals in organs, are described by sets of differential equations. The objective of the simulation is to solve these equations over a given simulated time interval to predict the levels

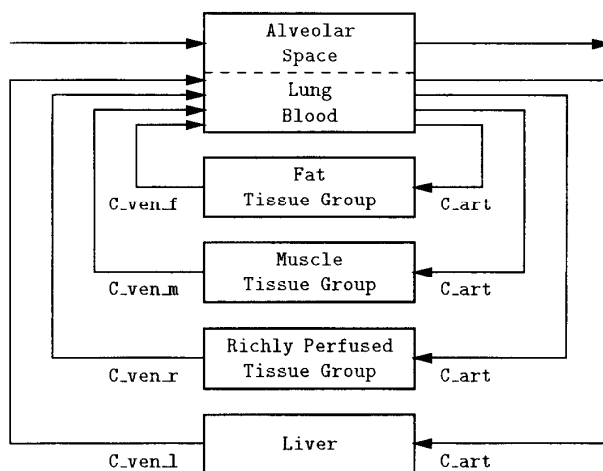


Fig. 3. A toxicology model.

of certain variables, such as the concentrations in the various organs, as functions of time. Traditionally, the models have been implemented using imperative programming languages, which do not lend themselves well to performance improvements through parallelization. Furthermore, it is impossible to interact with the simulation process at run time, for example, to alter some of the equations or constants.

Our implementation uses **MESSAGERS** as a control language to coordinate the operation and interaction of compiled node-resident C functions, which carry out the actual computations of the model. The basic approach is to map each organ onto a separate node, which contains the necessary sets of constants and differential equations as C functions. The toxin-carrying fluids, such as blood, are implemented as waves of consecutive Messengers, which cycle through the organism along the predefined paths, thus mimicking the actual flow through the body over time. As they pass through the organs, they trigger the execution of the appropriate functions to compute the new concentrations and other values for the current simulated time increment.

Fig. 4 shows the complete Messenger script for this simulation. The corresponding Messenger is injected into the lung node, where it first invokes the *new_lung_conc* function (line 4) to compute the new styrene concentration using *inhaled_sty* and *c_veinal* (initially zero). The circulation is accomplished using the two *hop* statements; the first (line 6) replicates the Messenger to all organ nodes, where the corresponding *new_org_conc* computes the new styrene concentration; the second *hop* statement (line 8) then takes the Messenger back to the lung node along the veinal links, where it adds its *c_ven_o* value (which represents the organ's contribution to the incremental change in the veinal styrene concentration) to the cumulative node variable *c_veinal* (line 9). To ensure that all Messengers have arrived at the lung node before the next

```

(1) styrene_sim(n_iter, inhaled_sty) {
(2)   c_veinal == 0;
(3)   for (i = 0; i < n_iter; i++) {
(4)     func(name="new_lung_conc"; in=c_veinal, inhaled_sty; out = c_art);
(5)     n_msgrs = number_of_organs;
(6)     hop(link="arterial");
(7)     func(name="new_organ_conc"; in=c_art; out=c_ven_o);
(8)     hop(link="veinal");
(9)     c_veinal += c_ven_o;
(10)    if (--n_msgrs > 0)
(11)      exit;
(12)  }
(13) }

```

Fig. 4. Messenger script for styrene inhalation.

cycle begins, the shared node variable *n_msgrs* in the lung node is used as a synchronization counter. At the beginning of each cycle it is set to the number of organs (line 5), thus counting the number of Messenger replicas. Upon returning to the lung node, each Messenger decrements this variable and, if this is greater than 0, it terminates (lines 10–11). Only the last Messenger survives the test and starts the next cycle.

3.2. Discussion of the example

The above simple example illustrates the principles of coordination under MESSENGERS. One of the most important observations is that it provides for a clear separation between computations, as expressed by the individual node functions, and coordination, which is the order of function invocations and the transport of information among them; the latter is accomplished by Messengers. This separation allows each layer to be designed and implemented separately. The Messengers do not need to know anything about the functions they invoke and, consequently, these can be supplied or changed at runtime. Similarly, the computational functions need not know when and by whom they will be invoked to perform their tasks.

This has several important implications for the design and use of the simulation application. First, it is possible to *reuse* the coordination layer in the design of other similar models. For example, the above Toxicology model can be turned into a cardiovascular simulation model by replacing the functions computing toxin concentrations by corresponding functions computing blood pressures and/or volumes, and carrying those values between the organ nodes in the same way as the current toxin concentrations. Second, unlike most existing simulation and modeling tools, MESSENGERS does not require the entire experiment to be set up a priori. Rather, each simulation run is composed dynamically as Messengers propagate through the network and invoke the various compute functions. Hence, the simulation model is *open-ended* in terms of its functionality and can arbitrarily be modified or extended incrementally and interactively

at runtime. It allows both the definition and the state of a simulation experiment to be modified at runtime by either substituting different functions to be invoked or by replacing the circulating Messengers by different ones as necessary. Hence, intermediate results observed from the current simulation run may be used as immediate feedback to steer the ongoing experiment.

3.3. Other advantages of self-migrating objects

One of the main virtues of Messengers is that they are capable of self-migration, i.e., autonomous navigation through both the logical and the physical networks. This has significant benefits for both conceptual program development as well as implementation. We discuss some of these below:

Programming. Self-migration is very natural for expressing certain classes of applications. For example, it has been observed that groups of individuals are capable of performing complex maneuvers without any particular individual taking on the role of a leader. The complex behavior of the entire group is the results of local interactions among neighboring individuals. Similarly, complex natural phenomena, like turbulence or heat conduction, can effectively be modeled by only describing the local interactions of neighboring particles. Such models are called individual-based and have been applied successfully in a number of different domains, including interactive battle simulations [7], particle-level simulations in physics [11], traffic modeling [22], evolution and behavior in biology/ecology [13, 24, 14], artificial life [17], and advanced graphics and animation [23]. Messengers, due to their self-migrating capabilities, are a natural paradigm for these types of applications [9]. Each entity type corresponds to a separate Messenger script, that defines its behavior and its potential interactions with other entities. Hence, all programs are written from the point of view of an autonomously operating individual. This is much more natural than using communicating processes or high-level declarative programming. Other examples of applications that benefit from the navigational perspective are circulatory simulations, such as the Toxicology model described in Section 3.1.

Scalability due to reduced communication overhead. An important implication of the MESSENGERS paradigm is that the individual entities move not only within the simulated space but actually migrate among processors in the network. This occurs when an entity crosses the boundary of a logical region. The corresponding Messenger performs a *hop* statement, which causes it to automatically migrate into the processor responsible for the corresponding logical region. Consequently, most interactions among logically neighboring entities will occur within the same processor. Only entities that are near the boundary will require the processor to communicate with the corresponding neighbor. Hence, inter-processor communication is restricted to only near-neighbor exchanges which makes the architecture arbitrarily scalable. In contrast, systems without physical migration require that each processor either communicates with all other processors to detect the proximity of other objects, or that some complex tracking mechanisms be implemented.

Load balancing by changing areas of responsibility. Self-migration offers a unique approach to load balancing based on the navigational autonomy of individual Messengers. This is applicable in situations where each logical node represents an area of responsibility within which application entities move by recomputing their respective positions. When the new position is outside of a node's area of responsibility, the entity automatically *hops* to the appropriate node. This can be exploited for load balancing by dynamically adjusting the decomposition of the logical space within which entities navigate. When a node's area of responsibility is decreased, some of its entities will automatically be forced to *hop* to neighboring nodes, thus decreasing its load. This can be performed periodically or in response to changing loads on the processors. We are currently investigating the suitability of various spatially-oriented decomposition strategies to improve the performance of the individual-based models mentioned above.

4. Related research

In this section, we compare MESSENGERS with PoliS, which is the most similar coordination model. Since the navigational features of MESSENGERS also share some similarities with mobile agents, network browsers, and systems that support process migration, we summarize these systems and contrast them with MESSENGERS.

4.1. Comparison with the PoliS model

As we mentioned in the introduction, MESSENGERS is similar to Linda. One of the main differences is that in MESSENGERS, the state space is explicitly partitioned. Here we compare MESSENGERS to PoliS [6], an enhancement to the Linda model intended to simplify the design of distributed systems by incorporating explicit partitioning of the tuple space. PoliS allows multiple named tuple spaces, called *places*, such that each tuple belongs to exactly one tuple space. The collection of all tuple spaces is called the *polispace*. The polispace in PoliS corresponds to the MESSENGERS logical network, and the tuple spaces in PoliS correspond to the nodes in the MESSENGERS logical network. In MESSENGERS, a link between any two logical nodes may or may not be present. In PoliS there are no explicit links; conceptually, the named tuple spaces form the nodes of a fully connected network. The execution threads in PoliS are autonomous active tuples, called *agents*. Because an agent is a tuple, an agent belongs to exactly one tuple space. An agent can read tuples inside its own tuple space and can write tuples to any tuple space. We compare these two models along three dimensions: dynamicity, mobility, and communication.

Dynamicity: Both MESSENGERS and PoliS support the ability to dynamically add new partitions to the state space (new logical nodes and new tuple spaces, respectively), and to create new processes. In PoliS, process creation is performed by writing an agent tuple into the appropriate tuple space. In MESSENGERS, this is done by injecting a new

Messenger. MESSENGERS also supports dynamic composition by allowing C functions to be stored at nodes and carried from node to node by Messengers; these functions are passive objects that become active when they are invoked through a *func* or *exec* statement. There is no direct support for a similar operation in PoliS, but the effect could be achieved by combinations of other operations.

Process mobility: The ability of a Messenger to move from one node to another is directly supported in MESSENGERS through the *hop* statement. The same effect can be achieved in PoliS by having an agent write a copy of itself into another tuple space and terminating itself in the current tuple space. In MESSENGERS, multiple destination nodes may be specified. The specification can be made implicitly, using the navigational calculus described in Section 2.3. In PoliS, the destination tuple space is specified by name.

Communication: In PoliS, the task of data exchange in a given tuple space is achieved by one agent writing a tuple into the tuple space and another agent reading it. The same effect is achieved in MESSENGERS by having two Messengers rendezvous at the node. The data in a PoliS tuple space is content addressable, while the data in a MESSENGERS logical node are structured and addressed using conventional C structuring and addressing techniques.

In summary, while the two systems are functionally comparable in terms of their coordination capabilities, MESSENGERS' point of view is strongly navigation-oriented. Messengers scripts are written from the point of view of a moving entity. They are used for all aspects of coordination, including dynamic creation and destruction of spaces (logical nodes), creation and destruction of activities (Messengers), and inter-Messengers communications.

4.2. Mobile agents

The term (intelligent) mobile agents has been used by a number of recent projects to describe autonomous programs, capable of physically moving through wide-area communication networks (notably the Internet), and performing a variety of service tasks on behalf of their users. One of the first and best known representatives of this paradigm is Telescript [25]. Different aspects of mobile agents have been explored in a multitude of recent projects [15, 18].

While MESSENGERS uses the same basic principles of autonomous navigation, it differs from mobile agent systems in two significant ways. First, it uses C to express both computations and navigation. Hence, MESSENGERS is only a new programming *paradigm*, rather than a new programming *language*. Second, it strives for *general purpose* distributed computing by attempting to harness the power of local area networks to speedup some computational task. Consequently, it provides explicit support for the incremental construction and use of an arbitrary logical network as a means of structuring and coordinating an application. Mobile agents, on the other hand, aim at providing a vehicle for "electronic commerce" or more efficient support for utilizing the various geographically distributed services provided on the Internet.

4.3. Network browsers

There is a variety of network browsers, all of which are capable of following hyper-text links and downloading various pieces of information via communication networks. Dynamic browsers are capable of downloading not only data but also functions (e.g., “applets”) and invoking them as part of the ongoing program execution. This dynamic composability (dynamic linking) is similar to MESSENGERS and is possible because the underlying languages (e.g., Java) are interpreted and hence portable across platforms. The main distinction between MESSENGERS and network browsers, however, is in navigation. Unlike a Messenger, a browser program does not have the ability to actively move out of its current node; programs can only be downloaded by the action of some other remote host program. Hence, network browsers can be characterized as “import-oriented” while MESSENGERS applications are “export-oriented” in that each Messenger can decide when and where it wishes to migrate.

4.4. Process migration

Process migration has been investigated by a number of projects. This can be supported by the underlying operating system, i.e., by supporting the migration of address spaces and process states between machines, or at the object level. Emerald [20] provides a number of specific primitives to efficiently locate and move fine-grain objects (including processes). Similarly, Obliq [4] permits threads to transfer themselves between physical sites while retaining their original semantics. These forms of migration differ from MESSENGERS in two significant ways. First, with a few notable exceptions, migration typically takes on a *passive* form: the migrating process has no autonomy – it is *being* moved by a command executed by some other object. A Messenger, in contrast, moves as the result of its *own* actions. Second, the purpose of migration in most systems is to improve performance. With MESSENGERS, navigating through the network is a concept inherent to the basic programming paradigm and is an important and explicit part of any application, even when running on a single processor.

5. Conclusions

We have presented a programming paradigm, called MESSENGERS, which is based on the concept of autonomous messages, each carrying a complete program to describe its behavior. These programs are interpreted by daemons running in each physical node of a network, which allow Messengers to navigate through the network, to communicate with one another, and to invoke compiled native-mode C functions. The main characteristic that distinguishes MESSENGERS from other approaches using autonomous objects is its intended application domain, which focuses on the dynamic composition of performance-oriented distributed applications, rather than providing agents for the electronic market place or a geographically distributed communication infrastructure.

In this paper we have focused on MESSENGERS' capabilities as a coordination paradigm. A Messenger's behavior depends not only on the interpretive script it carries but also on the functions it may invoke and utilize when visiting a node. Hence, the behavior of any given Messenger is determined by the composition of functions it invokes at run time. This capability, which we refer to as dynamic composition, allows us to view an application as a collection of distributed node-resident functions, whose invocation and data exchange is coordinated by the Messengers. Their navigational capabilities address the spatial aspect of coordination, in that each Messenger determines where it wishes to perform a given activity by invoking a compiled C function and where information exchange takes place. The temporal aspect is addressed by the inherently sequential execution of any given Messenger, which automatically serializes all functions it invokes, and through special scheduling functions, capable of ordering the various events with respect to global virtual time maintained by the system.

MESSENGERS is currently operational on Sun workstations. The software is available in source code free of charge for research purposes. For additional information, the reader is referred to our WWW page:

<http://www.ics.uci.edu/~bic/messengers>.

References

- [1] F. Arbab, The IWIM model for coordination of concurrent activities, in: P. Ciancarini, C. Hankin (Eds.), *Coordination Languages and Models*, Cescna, Italy, April 1996, *Lecture Notes in Computer Science*, Vol. 1061, Springer, Berlin, pp. 34–56.
- [2] J.-P. Banâtre, D. Le Métayer, Programming by multiset transformation, *Comm. ACM* 36 (1) (1993) 98–111.
- [3] L.F. Bic, M. Fukuda, M. Dillencourt, Distributed computing using autonomous objects, *IEEE Comput.* 29 (8) (1996).
- [4] L. Cardelli, Obliq: A language with distributed scope, *Comput. Systems* 8 (1) (1995) 27–59.
- [5] N. Carriero, D. Gelernter, Linda in context, *Comm. ACM* 32 (4) (1989).
- [6] P. Ciancarini, Distributed programming with logic tuple spaces, *New Generation Computing* 12 (3) (1994) 251–284.
- [7] DIS Steering Committee, The DIS vision: A map to the future of distributed simulation, Institute for Simulation and Training, 1994.
- [8] R.M. Fujimoto, Optimistic approaches to parallel discrete event simulation, *Trans. Soc. Comput. Simulation* 7 (2) (1990) 153–191.
- [9] M. Fukuda, L.F. Bic, M. Dillencourt, F. Merchant, Intra- and inter-object coordination with MESSENGERS, in: P. Ciancarini, C. Hankin (Eds.), *Coordination Languages and Models*, Cescna, Italy, April 1996, *Lecture Notes in Computer Science*, Vol. 1061, Springer, Berlin, 1996, pp. 179–196.
- [10] M. Fukuda, K.L. Morse, L.F. Bic, M. Dillencourt, E. Lee, D. Menzel, A novel approach to toxicology simulation based on autonomous objects, SCS Western MultiConference, Simulation in the Medical Sciences, January 1996.
- [11] R.W. Hockney, J.W. Eastwood, *Computer Simulations using Particles*, IOP Publishing Ltd, Bristol, Great Britain, 1988.
- [12] A.A. Holzbacher, A software environment for concurrent coordinated programming, in: P. Ciancarini, C. Hankin (Eds.), *Coordination Languages and Models*, Cescna, Italy, April 1996, *Lecture Notes in Computer Science*, Vol. 1061, Springer, Berlin, 1996, pp. 249–226.
- [13] M. Huston, D. DeAngelis, W. Post, New computer models unify ecological theory, *BioScience* 38 (10) (1988) 682–691.
- [14] A. Huth, C. Wissel, The simulation of the movement of fish schools, *J. Theoret. Biol.* 156 (1992) 365–385.

- [15] IEE, IEE Coll. on Intelligent Agents and their applications, No. 1996/101, London, UK, April 1996.
- [16] D.R. Jefferson, Virtual time, *ACM TOPLAS* 7 (3) (1985) 404–425.
- [17] C. Langton (Ed.), *Artificial Life III*, Addison-Wesley, Reading, MA, 1994.
- [18] MA '97: 1st Internat. Conf. on Mobile Agents, *Lecture Notes in Computer Science*, Springer, Berlin, 1997.
- [19] S.L. Mabry, L.F. Bic, Distributed biomedical simulation in SPEEDES, in: *Proc. of SCS Simulation Multiconference, High Performance Computing Symp.*, April 1996, pp. 36–41.
- [20] R.K. Raj, E.D. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, E. Jul, Emerald: A general-purpose programming language, *Software – Practice & Experience* 21 (1) (1991) 91–118.
- [21] J.C. Ramsey, M.E. Andersen, A physiologically based description of the inhalation pharmacokinetics of styrene in rats and humans, *Toxicol. Appl. Pharmacol.* 73 (1984) 159–175.
- [22] M. Resnick, Changing the centralized mind, *Technol. Rev.* (July 1994) 33–40.
- [23] C.W. Reynolds, Flocks, herds, and schools: A distributed behavioral model, *Comput. Graphics* 21 (4) (1987) 25–34.
- [24] F. Villa, New computer architectures as tools for ecological thought, *Trends Ecology Evolution (TREE)* 7 (6) (1992) 179–183.
- [25] J.E. White, Telescript technology, Technical Report, General Magic, Inc., Mountain View, CA 94040, 1994.