Programming Language Concepts for Global Computing

Ulisses Ferreira

Abstract — *The present paper introduces a number of existing concepts about the programming language Plain that are useful for mobile agents and for the Internet programming, among other applications. This paper explores the ability to program with the unknown value, by preventing samples.*

Index Terms — *global computer, internet, programming language*

1. INTRODUCTION

For writing the present paper, the present author observed that, in order to provide a reasonable paradigm for general Internet programming, some combined and non-traditional programming language concepts need to be introduced aiming at specific applications over the Internet.

Code mobility is a relatively new field of research that has inspired intriguing ideas on programming techniques to improve related software[8]. New programming languages have been presented and discussed in workshops and conferences aiming at providing better standards and good examples for future language designs, commercially or otherwise.

In comparison with the state-of-the-art technology, to propose a better paradigm for programming mobile agents for the World Wide Web, for example, the language designer should consider that the underlying connections often fail or delay. A neutral state, which would represent the lack of result, could be assigned to the variable that takes part in the request in such a way that the program carries on running safely. Mobile agents need to be robust and make their own decisions remotely. Two very different and alternative approaches to agent are [25], [33].

Manuscript received June 3, 2004. This piece of work corresponds to a chapter of the present author´s Ph.D. Thesis (2001), at the University of Dublin, Dublin 2, Ireland. Updated author´s e-mail: josefer@philosophers-fcs.org.

Although the present programming language features are not necessarily specific for mobile code languages, the underlying environment can be the same. Here the present author explains] how this additional value can be useful in programming on a global network[35] where the mobile agent paradigm and technology have become increasingly important. In logics, this value has been traditionally referred to as *uu* for providing an alternative value to *true* (*tt*) and *false* (*ff*). Like many imperative features, the present ones also apply to functional languages.

Broadly, some pieces of work, such as [30], [31], have indicated similarities between technologies of code mobility and persistence, and some persistent languages are being explored at some universities. In the present approach, because the present author is looking for generality and efficiency, persistence is not provided directly by the language, but instead by programmable constructs. Because of that, this approach here is at least as applicable to mobile agents as persistent languages. As well as persistence, communication is another very active topic of research and much work has been done regarding fault tolerance and communication between mobile agents. However, relatively few satisfactory results have been achieved in terms of language facilities and abstractions.

Here we concentrate on the use of programming features in the context of a global environment and mobile agents programming.

The ability to represent and reason with partial information is well understood in the artificial intelligence and logic communities. However, very little of this work has been related to programming techniques. An exception is Extended Logic Programming, introduced by Gelfond and Lifschitz[16], [17], that can be used for the same purpose as that the present author is discussing here. Extended Logic Programming makes use of two forms of negation. In [23], the author suggests that an important practical problem in Extended Logic Programming is how the programmer distinguishes whether a negative condition is to be interpreted as explicit negation, or as negation due to the absence of any clause in any closed world as an assumption.

The *unknown* value, *uu*, extends the semantics of other logics, such as classical and intuitionistic logic, according to the Lukasiewicz [38], [26] 3-valued logic. Here the present author extends this value for each data type in programming for a global environment or for mobile agents. In arithmetic and relational expressions, *uu* as a resulting operand implies the] expression to result in *uu*. Accordingly, statements have to be adapted to make use of this value.

Most Expert System Shells made use of an *unknown* symbol to represent lack of information in Boolean variables, in a very restrictive way however. The present author generalizes this concept to programming languages in general, although he applies it here to mobile agents programming.

Agents have to be robust and, because of this, when connections fail or delay, programs should carry on running despite the lack of information. uu is a constant in programming languages that can be assigned to any variable of any datatype. This new constant guarantees both safety and robustness at the same time, because variables are never committed to any value that is not in the problem domain. An introduction on types for functional programming languages can be found in [34].

In section 2 the present author reviews some recent programming languages for such an environment, while section 3 is dedicated to an illustrative programming language. Section 4 introduces the concept of *unknown* together with other related concepts, and explains how they can be used to achieve the proposed goals. As a consequence, section 5 complements those concepts by stressing the importance of any form of lazy evaluation in programming, as well as timeouts, no matter the adopted paradigm. Section 6 contains other relevant features that are relevant enough to be mentioned. Section 7 contains the conclusion.

The examples in this paper are written in a programming language. In section 3, the present author discusses the syntax of the relevant subset of this language.

2. Some Current Mobile Code Languages

In the past few years researchers have seen the Internet as a popular environment for systems. Some of us would like to program and compute using this structure, i.e. to view parts of an Internet-like network as *global computers* [15]. Many companies, for example, are starting to have their own internal global computers for their specific purposes.

One of the most interesting ideas is not only to move code, as **Tcl**[32] and **Java**[3] do, but also computation (code along with context) over the network, that is, a computation which starts at some location may continue to execute at some other location. Synchronous connections to the original site may be set while a program is running remotely in such a way that any change in some variables transparently causes the value to be stored in the original site. Alternatively, new values of variables can be sent to the original site with no need for synchronous connections. Other paradigms of mobile computation already exist and they depend on the kind of entities that are transferred over the network, with respect to what is moved (code, data, connections, etc).

When the code is moved, what happens if the names it contains are bound to resources in the source virtual machine? This issue defines two classes of strategy, *replication* and *sharing*. The former strategy may be either static or dynamic. Concerning static replication strategy, constants, system variables and libraries, for example, are regarded as *ubiquitous resources*[27] and they can adopt such strategy, where bindings are deleted and set after arrival. As for dynamic replication strategy, the code migrates to another virtual machine along with bound resources and the original bindings are deleted. The original resource in the source virtual machine may be either deleted (*replication by move*) or kept (*replication by copy*). In the *sharing strategy*, the original resource is kept and remotely accessed through network references and connections.

In both strong and weak mobility[10], security[39],[40] is a very important matter. Locations must check for authorization and capabilities in order to prevent malicious software running. However, as long as that is ensured by the system, a global network can be a very interesting and natural platform for computation. Thus, a new challenge emerges: how to provide these facilities and prevent the related problems?

Mobility should also be considered not only during the execution of programs but also during the elaboration of software. The emphasis in performance is no longer in the run-time code generated by compilers, but in the (dynamic) compilation process itself (when applicable), transmission and additional overheads to guarantee security and other requirements.

Acharya, Ranganathan and Saltz[1],[2] during the design of **Sumatra**, an extension of **Java**, consider three requirements for **individuals**: *awareness*, which is the need to monitor the level and quality of resources in their operating environment; *agility*, which is the ability to react to changes in resource availability, and *authority*, which is the ability to control the way that resources are used. Although they are important concerns, the present author thinks that these concerns should be treated at the application level, not at the language level.

Some programming languages for mobile computation are described and analyzed in [10] and other articles, and briefly described here:

• **Java**[3] is a strongly-typed object-oriented language. **Java** deals with security[12] and allows transmitting program byte-code to be interpreted by the **Java** Virtual Machine[28], but does not migrate computation. It supports weak mobility with dynamic linking. Security level is increased by the byte-code verifier at loading-time. Some security problems have been found[12]. It was shown that the ability to break **Java** type system leads to an attacker being able to run arbitrary machine code[11]. Static and dynamic type checking.

• **Telescript**[41] is an agent-based and object-oriented language that explicitly deals with locality, strong mobility, and finiteness of resources. There are two kinds of Execution Units: agents and places. Typically, when an agent is running on an interpreter, the *go* instruction causes the agent execution to be suspended, its code and current state are transmitted to a remote virtual machine and, there, the computation is resumed. However, agents do not maintain connections to remote agents. The **Telescript** run-time code is interpreted without security checking since security is ensured at the language level. The replication strategy is dynamic, by move. Static scoping and name resolution. Static and dynamic type checking. In spite of the historical reasons for mentioning **Telescript** here, that technology was replaced by **Odyssey**[9], which is a **Java**-based version of **Telescript**, briefly speaking.

• **Tycoon**[29] provides thread migration like **Telescript**. It is a polymorphic, higher-order functional language with imperative features, which may support other paradigms indirectly, including object orientation. **Tycoon** provides strong mobility and support for persistent programming. All objects in this language have first-class status. Static and dynamic type checking, dynamic replication with strategy by copy, besides static replication strategy.

• **Agent Tcl**[18] provides strong mobility where the whole image of the interpreter can be transferred to a different site by executing a *jump* instruction. **Agent Tcl** also provides weak mobility by executing a *submit* instruction which allows transmission of procedures along with part of their global environment, to a remote interpreter. Typeless language therefore no type checking. Dynamic replication strategy, both by copy and by move. **Agent Tcl** is a Ph.D. Thesis[19].

• **Safe-Tcl**[5] supports active e-mail, where messages may include code to be executed when an interpreter reads the message after receiving it. However, **Safe-Tcl** does not support active e-mail code mobility at the language level but, instead, code mobility is achieved through a dynamic code loading mechanism. Typeless language therefore no type checking.

• **Obliq**[4] is an object-based language that encourages distribution and mobility. While a mobile object is migrating from one place to another, new connections are automatically open between source and destination places in order to guarantee that any change in the variables will update the state in the source place. Therefore, object references are transformed into network references. Although a simple language, there is some loss of efficiency and robustness due to some possibly very large number of connections in an unreliable environment. Dynamic type checking, sharing strategy.

• **Facile**[36] is a functional language, a superset of **ML** with primitives for distribution, concurrency and communication. Mobile code programming was later added to this extension[27]. Static and dynamic type checking. Dynamic replication strategy by copy, besides static replication strategy.

• **TACOMA**[20], [21], the **Tcl** language plus primitives to allow a running **Tcl** script to send another script and initialization data to another host in order to execute the script remotely. Typeless language therefore no type checking. Dynamic replication by copy.

• **M0**[10][37] is a stack-based interpreted language which provides weak mobility and run-time type checking. Dynamic type checking, dynamic scoping rules, dynamic replication by copy.

Aglets Workbench, developed by IBM, is a mobile gent system based on **Java**. Like others, such as ObjectSpace **Voyager**, the system security and other issues depend on the **Java** system[24].

As mentioned before, in the present paper, the present author discusses some of the features of the programming language.

3. *THE PRESENT PROGRAMMING LANGUAGE*

The present sample programming language is a language that supports mobile agents, syntactically somewhat similar to **Java**. It supports strong mobility, as well as some forms of knowledge and belief representation, reasoning, and uncertainty treatment. As an on-going experimental project, security has not been a concern[1]. The virtual machine interprets byte- code and the language provides both replication strategies by programmable handlers. BNF legend: boldface letters are keywords; italic words with initial capital letter are other terminal symbols; words in lower-case letters are non-terminal symbols; meta-symbols: **|** indicates alternative, e is the empty symbol of the grammar. Other terminal symbols: { (, ;) } are used in the grammar. The following BNF definition is of a very simple subset of the programming language in question, and where the first symbol denotes the starting symbol of the syntax:

aprog ? classlist commandlist

classlist ? e | classdef classlist

type ? **int** | **list**

modifier ? **private** | **public** | e

onevardef ? *Id* | assignment

idlist ? onevardef | onevardef ',' idlist

vardef ? modifier type idlist ';'

handler ? evaluator | reactor

evaluator ? **when** *Id* ',' **do** command

reactor ? **when** *Id* ':=' **do** command

classdef ? **class** *Id* '{' defs '}'

defs ? e | vardef defs | function defs | handler defs

command ? assignment | '{' commandlist '}'
| functioncall | ifcommand | **return** |
return} expression

assignment ? *Id* ':=' expression

ifcommand ? **if** expression **then** command |
if expression ',' command |
if expression ',' command **ifnot** command |
if expression ',' command **else** command |
if expression ',' command **otherwise** command |
if expression ',' command
ifnot command
otherwise command

commandlist ? e | command ';' commandlist

where non-terminal symbols, namely *function*, *functioncall* and *expression* are as usual. In **Plain**, they are somewhat syntactically similar to **C++** or **Java**. The main difference is that the symbol $ can be placed where a variable identifier is expected, as it will be explained below. There are other details that will be explained together with the examples. Both [13] and [14] formalize the semantics that will be explained.

4. *UU IN GLOBAL COMPUTERS*

As already stated, *uu* is a constant which stands for ``unknown'' or ``undefined'' and represents unavailable information.

More precisely, *uu* can be used in programming languages in accordance with the following description: For every data

[1] However, the first mobile agent in this language, referred to as **Se-Picou**, had perfectly run over the Internet: briefly, as a prototype, **Se-Picou** started running in the city of Edinburgh; went to a site in Brazil; made some calculation; sent partial results by e-mail; moved back to Edinburgh and finally resumed.

type if any, the language designer can add a special value, namely *uu*, to represent lack of some *domain value*, i.e. some known value in the problem domain. Grammatically, *uu* or *unknown* is a constant. Variables either have *uu* or some domain value. In advance, besides other applications of *uu* in some programming language, *uu* can support fault tolerance over the Internet, and this will be clear while the subject is introduced.

Some languages adopt a default value as initial variable contents. Nonetheless, since there is now *uu*, any variable (at least in the **Plain** programming language) contains this value as the initial one. Programmers should certainly want to initially assign some values to some variables.

Handlers are provided for variables. For any variable, there can be one *evaluator* and/or one *reactor*, independently. As well as other purposes, one handler can protect a variable. The notion of evaluator is for permitting the programmer to write a piece of code for producing and providing some domain value to the corresponding variable, while the idea of reactor is for inspecting and protecting the variable against assignments. Thus, a reactor permits the programmer to write a piece of code to be executed whenever a value is meant to be stored in the corresponding variable. For instance,

```
int x, y;
when x, do { x := 3 * y;}
when x := do { x := $/2; }
```

is an example where two handlers are defined for the variable *x* (Remember, here a handler is a declaration **when**). At the first time that the value of *x* is being requested in an expression, the above evaluator is triggered, which in turn computes the triple of the value of the variable *y* assigning it to *x*. From the second time on, the computed value *3 * y* is already available in *x* and, because of this, the evaluator is not triggered. This idea is not limited to *exception handling*, which in turn is a mechanism supported by some other languages.

An evaluator can contain **return** statement (similar to **C**) as an alternative to assigning a value to the requested variable. In the case of the **return** statement and no prior assignment in the evaluator, the evaluator is always triggered when the variable in question is being used, unless some domain value has been assigned to that variable outside the evaluator.

Whenever a value is meant to be stored in *x*, the control is jumped to the corresponding reactor. Notice that the $ symbol above is used in reactors to represent the value that, in other languages, would be stored unconditionally. In the above example, half the value is accepted.

Predicates can be provided to check whether a variable contains *uu*, for instance, **known** and **unknown**. For each of them, the value is accessed directly from the variable in question and, then, the corresponding binary (either true or false) result from the predicate expression is provided by the interpreter without evaluating the handler of this variable.

If the referring variable contains some value in the problem domain, the semantics is exactly the same as in imperative languages. However, if the variable contains *uu* instead, there are two semantic cases: If there is an evaluator, it is executed. Otherwise, i.e. when no evaluator exists, *uu* is used instead. Moreover, the semantics of an evaluator is not very similar to the semantics of any function call, for the latter is necessarily executed. As an example, in the case of Remote Procedure Call or Remote Method Evaluation, for being both remote, unconditional calls are probably left as the last resort.

In terms of programming language design, *uu* and handlers together replace exception handling used in other languages. This replacement tends to make any language more economic and hence much easier to use. More than this, handlers are very useful during program *testing* and *debugging* phases, because one might inspect what is being stored and, as mobile agents often escape from any of their users, *uu* together with handlers can be used in such a program for sending results to the person who is testing it. For example:

```
class mycl {
  public int x;
  private list queue := [ ];
  when x := do {
   x := $;
   queue := queue +
   [ [ #self + ``.x := `` + $ + `` at `` +
LocalTime() ] ];
        }
      }

mycl c;   c.x := 10;   c.x := 20;   c.x := 30;
```

In the above class, or its subclasses, whenever *x* receives any value, this value is additionally stored in the queue together with the name of the object in question (which is referred to by the expression *#self*), the name of this field with one dot (it is ``.x'' in this case), and the current local time (accessed by *LocalTime()*). The '+' operator the concatenation of lists or strings, besides the arithmetic addition, as more usual. The square brackets are used to construct a list of values of any type. Here the programmer chose list of lists for programming reasons.

Because it is hard to implement debugging system for mobile agents in a relatively satisfactory way, the above piece of code can be written as long as handlers exist. More generally, when a mobile agent stops running, the local runtime system should provide a way of returning that agent to its home, when requested, in order to permit local debugging. Therefore, by using a small query language, the user can inspect the contents of the variables of his or her program. More than this, by writing the *trace* declaration in such a program, a mobile object support system can internally maintain debugging variables.

The difference from other paradigms might become decisive in language design. On the one hand, a variable in an evaluating expression may cause its value to be read from a data base or requested from a remote process, provided that its current value is *uu*. Further, a variable may work like a *cache* because in the subsequent uses, some domain value is available locally and the handler is not triggered. On the other hand, to assign a value to a variable may cause its value to be stored on a data base or sent to a remote host.

The following piece of code exemplifies a persistent field *p* and a remote field *r* that can live together in the same class:

```
class remoteandpersistentcl {
  public int p, r;

  public void ini(int i, int j) {
   inttodb(``p'',i);   p := i;
   inttourl(``www.aaa.bbb.ccc/cgi/server/r.t
xt'',j);
   r := j;
  }

  when p := do { p := $;  inttodb(``p'',p);  }

  when p, do { return intfromdb(``p''); }

  when r := do {
   r := $;
   inttourl(``www.aaa.bbb/cgi/server/r.txt'',r
);
  }

  when r, do {
   r                                    :=
intfromurl(``www.aaa.bbb/cgi/server/r.txt'');
  }
}

remoteandpersistentcl c;
c.p := 20;   // also store 20 locally on data
base.
sendlocally(home,c.p); // send (c.p) home.
c.r := 30;   // also update remotely.
sendlocally(home, c.r); // send (c.r) home.
```

Notice that, according to the above evaluator and program, while the *p* field is being retrieved from a data base (whenever its value is requested), the *r* field behaves as a cache over some global environment. The functions *inttourl*, *intfromurl* and *sendlocally* tell the underlying system to generate internally mobile agents to take part in the protocol. There has been a general criticism concerning mobile agents because they do not maintain connections. The present author agrees that agents should not see connections, but the mobile agent support system should provide remote communication in an appropriate way. This produces positive effects and abstractions in the programming language.

Here we concentrate on features for global computing and present some new aspects of *uu*.

5. *Lazy Evaluation and Timeouts*

Lazy evaluation is one of the most interesting characteristics of programming, in particular in applications where time is regarded as important. In this paper, the present author is not regarding lazy evaluation as being only *call by need* of functional languages. If the language provides functions, lazy evaluation can also be very useful in the same platform, from the same point of view of the present section.

Programming for mobile agents on a global environment tends to be more personal. One of the reasons is that patience and mood vary for different people as well as for the same person at different instants, and one of the purposes of agents is to represent users.

As an example of a situation, a mobile agent *ma* can communicate with a stationary agent *s* which in its turn can send a small agent remotely to *ma*'s home in order to return some piece of information to *s* which in turn can hand it to the mobile agent *ma*. To deal with faults and delays in communication, a timeout can be set, implicitly or explicitly, for every input operation. After that time, the result is *unknown* (*uu*) and the computation continues normally. Similarly, every output operation has a timeout.

In this way, the same statement can be executed at different locations[7], either sequentially or not. This situation happens often. Cache-like variables might produce a similar result as lazy evaluation. Computing with timeouts together with *uu* and handlers is not lazy evaluation, but it can give a somewhat similar impression of *impatience* and, because the resulting value in this case of exception is *uu*, variable values in programs are always sound and finally this scheme improves *agent robustness*.

Another way of dealing with faults and delays is to provide a standard semantics for basic operations such as arithmetic and relational. In particular, if the first operand is *uu*, the expression might result in *uu* without the evaluation of the second operand. This is a form of lazy evaluation.

6. OTHER FEATURES

Because generality is desirable, choices among various strategies for binding resources should normally be programmed. Handlers may be used to implement different strategies for variables that are resources, either local or remote.

During the compilation, in order to support higher-level communication between agents, names of objects in the source program can be written in the object code, which increases the agent size but it is still a good idea. A possibility is to generate only names defined in the dynamic part of the interface. If the language supports artificial intelligence techniques, perhaps it is even interesting to consider the idea of generating all names. Communication between agents can be set from a prefix in function calls containing the name of the destination agent. For example, in *x := prov:func(params)*, the string variable *prov* is a name that indicates the agent which in turn might contain the *func* function definition. The *prov* value is an absolute (global) or relative (to the local host) address. If such a matching name of *func(params)* is undefined in that agent when the call is executed, *x* receives *uu*. In every function call (or method invocation) between two agents, a timeout can be attached. For example, in *x := prov:func(params)* **timeout** 3, if the operation is not completed before 3 seconds, at that time it is interrupted and *x* receives *uu*.

The concepts of **home** and **Id** of agents ought to be key words in the programming language, in a similar way as exemplified above, outside the class *remoteandpersistentcl*.

Surprisingly, although *concurrent programming*[6] is an important technique that can help in certain applications, it is not a specific feature for mobile agent programming languages, as concurrency can be achieved at the operating system level.

However, *uu* permits a large number of parallel operations, not only parallel *and* and parallel *or*[14].

7. CONCLUSION

The presented concepts are harmonious with many approaches in software engineering such as in [22].

Local inefficiency is an issue of the features discussed in this paper. Assuming that, in practice, mobile agent support systems entail code interpretation, the interpreter has to check the presence of *uu* whenever a variable is being requested in an evaluating expression. However, as hardware is getting faster and larger, this is not considered a significant problem. Moreover, this problem can be compensated for the fact that mobility and remote accesses are the bottleneck in applications, and that variables can behave as cache and operations can be lazy. This combination is encouraged by the language.

ACKNOWLEDGMENT

My honest thanks to the reviewers who kindly collaborated with suggestions here.

REFERENCES

[1] Acharya, A., Ranganathan, M., Saltz, J., "Dynamic linking for mobile programs", in Mobile Object Systems: Towards the Programmable Internet, *Springer-Verlag*, Apr. 1997, pp. 245--262, Lecture Notes in Computer Science, 1222.

[2] ------, "Sumatra: A language for resource-aware mobile programs", in Mobile Object Systems: Towards the Programmable Internet, *Springer-Verlag*, Apr. 1997, pp. 111--130, Lecture Notes in Computer Science No. 1222.

[3] Arnold, K., Goslin, J., "The Java Programming Language", *Addison-Wesley Publishing Company*, 1996.

[4] Bharat, K., Cardelli, L., "Migratory applications", in Mobile Object Systems: Towards the Programmable Internet, *Springer-Verlag*, Apr. 1997, pp. 131--149, Lecture Notes in Computer Science No. 1222.

[5] Borenstein, N. S., "Email with a mind of its own: The Safe-TCL language for enabled mail", First *Virtual Holdings, Inc*, Tech. Rep., 1994.

[6] Burns, A., Wellings, A., "Concurrency in Ada", 2nd ed, *Cambridge University Press*, 1998.

[7] Cardelli, L., "Global computation", *ACM*, Computing Surveys, vol. 28A, no. 4, 1996.

[8] Cardelli, L., Davies, R. "Service Combinators for Web Computing.", http://www.luca.demon.co.uk, 1997.

[9] G. M. Corp., "Odyssey White Paper", 1998.

[10] Cugola, G., Ghezzi, C., Picco, G. P., Vigna, G., "Analyzing mobile code languages", in Mobile Object Systems: Towards the Programmable Internet, *Springer*, Apr. 1997, pp. 93--110, Lecture Notes in Computer Science No. 1222.

[11] Dean, D., "The security of static typing with dynamic linking" in Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1997.

[12] Dean, D., Felten, E. W., Wallach, D. S., "Java Security: from Hotjava to Netscape and Beyond", in Proceedings of the Symposium on Security and Privacy, *IEEE*, Oakland, Cal., 1996, pp. 190--200.

[13] Ferreira, U., ``uu for programming languages", *ACM*, SIGPLAN Notices, August 2000, vol. 35, no. 8, pp. 20--30.

[14] ------, "Programming languages features for some global computer", in Proceedings of SSGRR 2003s International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet, *IPSI*, Scuola Superiore G. Reiss Romoli and Telecom Italia Learning Services, From 28 July to 3 August 2003.

[15] Freeman, E., "Supercomputer earth: Massively parallel internet", *Yale University*, Tech. Rep., December 1993, supplement to the Yale Weekly Bulletin.

[16] Gelfond, M., Lifschitz, V., "Logic programs with classical negation", in Proceedings of 7th International Conference on Logic Programming, *The MIT Press*, Cambridge MA, 1990, pp. 579--597.

[17] ------, "Classical negation in logic programs and disjunctive databases", *Ohmsha Ltd and Spring-Verlag*, New Generation Computing, 1991, pp. 365--385.

[18] Gray, R. S., "Agent TCL: a transportable agent system", n Proceedings of the CIKM'95 Workshop on Intelligent Information Agent, 1995.

[19] ------, "Agent TCL: A flexible and secure mobile-agent system", Dartmouth College, Computer Science, Hanover, Tech. Rep. PCS-TR98-327, 1998, Ph.D. Thesis, 1997.

[20] Johansen, D., "Mobile agent applicability", in Mobile Agents: Second International Workshop, MA'98, Lecture Notes in Computer Science, vol. 1477, *Springer*, 1998.

[21] Johansen, D., van Renesse, R., Schneider, F. B., "An introduction to the TACOMA distributed system", Department of Computer Science, *University of Tromsø*, Tromsø, Norway, Tech. Rep. 95-23, June 1995.

[22] Jourdan, G.-V., Zaguia, N., "Incremental software development strategy: A sucessful experience", in Proceedings of The 2nd International Conference on Computer Science and its Applications, Dey, P. P., Amin, M. N., Gatton, T. M., Eds., *National University*, San Diego, CA, USA, June 2004, pp. 100--104.

[23] Kakas, A. C., Kowalski, R. A., Toni, F., "The Role of Abduction in Logic Programming", in Handbook of Logic in Artificial Intelligence and Logic Programming, *Oxford University Press*, 1998, vol. 5, pp. 235--324.

[24] Karjoth, G., Lange, D. B., Oshima, M., "A security model for agents", *IEEE*, Internet Computing, vol. 1, no. 4, July/August 1997.

[25] Kauffman, S, "Molecular autonomous agents", *Philosophical Transactions of The Royal Society*, June 2003, vol. 361, no. 1807, pp. 1089—1099.

[26] Kleene, S. C., "Introduction of Metamathematics", *D. Van Nostrand, Princeton*, 1952.

[27] Knabe, F. C., "Language support for mobile agents", Ph.D. dissertation, *Carnegie Mellon University*, Paittsburgh, Pa., Dec. 1995, also available as Carngie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC--95—36.

[28] Lindholm, T., Yellin, F., "The Java Virtual Machine Specification", *Addison-Wesley Publishing Company*, Reading, Massachussetts, 1997.

[29] Mathiske, B., Matthes, F., Schmidt, J. W., "On migrating threads", *Fachbereich Informatik Universitat Hamburg*, Tech. Rep., 1994.

[30] Da Silva, M. M., "Mobility and persistence", in Mobile Object Systems: Towards the Programmable Internet. *Springer-Verlag*, Apr. 1997, pp. 157--176, Lecture Notes in Computer Science No. 1222.

[31] Da Silva, M. M., Atkinson, M., "Combining mobile agents with persistent systems: Opportunities and challenges", in 2nd ECOOP Workshop on Mobile Object Systems, Linz, Austria, July 1996, pp. 36–40.

[32] Ousterhout, J. K., "Tcl and the Tk Toolkit", *Adison-Wesley*, 1994.

[33] Russel, S., Norvig, P., "Artificial Intelligence", 2nd ed., *Prentice Hall*, 2003, ch. Two.

[34] Schmidt, D. A., "The Structure of Typed Programming Languages", *The MIT Press*, Foundations of Computing Series, 1994.

[35] Tanenbaum, A. S., "Computer Networks", 4th ed, *Prentice Hall PTR*, 2003.

[36] Thomsen, B., Leth, L., Prasad, S., Kuo, T.-M., Kramer, A., Knabe, F. C., Giacalone, A., "Facile antigua release programming guide", *European Computer Industry Research Centre*, Munich, Germany, Tech. Rep. ECRC--93--20, Dec. 1993.

[37] Tschudin, C., "The messenger environment M0 -- a condensed description", in Mobile Object Systems: Towards the Programmable Internet, *Springer*, Apr. 1997, pp. 149--156, Lecture Notes in Computer Science, 1222.

[38] Lukasiewicz, J., "Jan Lukasiewicz Selected Works", *North-Holland Publishing Company and PWN - Polish Scientific Publishers*, Series on Studies in Logic and Foundations of Mathematics, 1970.

[39] Vitek, J., Serrano, M., Thanos, D., “Security and communication in mobile object systems”, in Mobile Object Systems: Towards the Programmable Internet. *Springer-Verlag*, Apr. 1997, pp. 177--200, Lecture Notes in Computer Science No. 1222.

[40] Volpano, D., "Provably-secure programming languages for remote evaluation", ACM Computing Surveys, vol. 28A, Dec. 1996, participation statement for ACM Workshop on Strategic Directions in Computing Research.

[41] White, J., "Telescript Technology: the Foundation for the Electronic Marketplace", *General Magic, Inc*., 1994.