Towards Customisable Tuple Field Matching In VLOS

Voon-Li Chung **Chris McDonald**

Department of Computer Science & Software Engineering
University of Western Australia,
35 Stirling Highway,
Crawley, Western Australia 6007,
Email: {vlchung,chris}@csse.uwa.edu.au

Abstract

VLOS is a research project investigating the feasibility of a tuple space-based distributed operating system for use on small to medium sized clusters of Intel™PC based computers.

Arguments are made that providing a mechanism by which tuple field matching expressions can be redefined from the usual "bitwise-binary" matching schemes to more complex (user-defined) matching expressions, would allow tuple space-based communications to be involved in the provision of distributed computational resources. These matching schemes also help simplify the complexity of distributed applications, by moving some of the computation from the application to the coordination medium.

A test implementation of a tuple field matching system is described; the MiniMe matching expression language is an in-kernel compiler whose language has been designed specifically to disallow dangerous operations. Several examples of MiniMe matching expressions are shown.

The protocol used by nodes to propagate expression matching changes is described here. The task that has redefined a tuple field matching expression blocks until the changes have been propagated across the cluster. These semantics ensure that the distributed application does not attempt to perform any operations using the tuple space prior to the field matching rule redefinition being propagated globally.

Keywords: Distributed Operating Systems, Compilers, Programming Language Implementation

1 Introduction

To date there has been very little research examining the use of a tuple space-based communication model as the basis for an operating system (Binning 1999). Instead, focus has been generally on providing applications' programmer support, typically through the use of libraries and programming language extensions (Picco, Murphy & Roman 1999, Davies, Wade, Friday & Blair 1997, Sun Microsystems 2000).

VLOS is a research project that is currently focused on exploring the feasibility of a distributed operating system using only tuple space-based communications. Unlike previous efforts, backwards compatibility with existing operating systems' APIs is not a design requirement for VLOS. Future programmer-level APIs will be designed, bearing in the mind the heavy focus that the operating system will have on manipulating tuple spaces, field types and type signatures.

Copyright ©2002, Australian Computer Society, Inc. This paper appeared at Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

1.1 VLOS' Current Status

The VLOS operating system currently consists of a capability-based security system for protecting tuple spaces, field types and type signatures from unauthorised use (Chung & McDonald 2002*b*), a distributed tuple space operations protocol, allowing tuple space-based communications to occur across all nodes in a cluster, and a disk-based non-volatile backing store (Chung & McDonald 2002*a*).

All inter-node communications for the security and tuple space operations protocol use the *Transaction Manager*. The Transaction manager provides reliable and secure communications for VLOS subsystems (Chung & McDonald 2002*a*). The Transaction Manager uses dedicated and encrypted one-to-one TCP/IP connections between all nodes in the VLOS cluster.

All systems described has been implemented using the Flux OSKit. The Flux OSKit is a series of libraries and utilities developed to help lower the barrier to operating system researchers. Originally developed by a team lead by Ford at the University of Utah, the OSKit provides a significant amount of the low-level infrastructure needed, such as device drivers and IP stacks, in a manner that is well defined and easily overridden (Ford, Back, Benson, Lepreau, Lin & Shivers 1997).

VLOS development is currently focused on the development of a process model which will eventually provide to the applications programmer a tuple space-centric view of processes. Processes will be treated as a specific instance of a tuple signature, allowing processes to be suspended, cloned or terminated using standard tuple space operations.

1.2 Overview

For readers unfamilar with the tuple space communications model, known by its implementation such as Linda (Carriero 1987), Sun's Javaspaces (Sun Microsystems 2000) or IBM's TSpaces (Wyckoff, McLaughry, Lehman & Ford 1998), a brief introduction to the paradigm follows in section 1.3.

Section 2 explains our motivation for redefining the matching expression for a data type's matching expression, arguing that movements in this direction have already been made, and that simplification of a distributed applications implementation can be achieved.

We argue that the movement of some of the computation from the distributed application to the coordination would allow coordination to be viewed in terms of the execution of a particular algorithm,

rather than purely in terms of the allocation of distributed resources. We illustrate this point by discussing two examples – defining floating-point number equality and the implementation of a genetic algorithm.

Section 4 describes an initial implementation of a system through which a field's matching expression can be changed – the MiniMe (MINI-Matching Expression) language, which was designed to be incorporated into the VLOS kernel itself, providing its services to user-level programs as a system call; in doing so, the VLOS operating system can offer customisable tuple field matching without the possibility of infinite loops, exceptions, or the violation of memory protection regimes. Some simple examples are also provided.

In order to propagate changes in the matching expression to other nodes in the cluster, a network protocol, using VLOS' Transaction Manager, is used. An originating task is blocked until all nodes in the cluster have acknowledged that they have redefined their tuple field matching expressions. The protocol was designed so that distributed application execution only continues when VLOS can guarantee that the tuple matching rule redefinition has been globally propagated. This is described in Section 6.

Section 7 concludes with some suggested future changes to the MiniMe system. Current suggestions include the ability to perform tuple space operations (`in`, `rd`, `out`, `collect` and `collect-cp`) while evaluating fields and the introduction of state information (variables) that can be used to store information across repeated field matching operations.

1.3 The Tuple Space Communications Model

Tuples are finite collections of strictly typed data objects divided into searchable fields. Tuples have an *arity* which refers to the number of their fields and a *type signature* which reflects the ordered types of these searchable fields, as described by its host language. For example, the tuple

```
('c', 12.30, 95551111)
```

has an arity of 3 and a type signature of

```
(char, float, int)
```

Originally, 4 tuple space manipulators existed: `out`, `in`, `rd` and `eval`. The `out` primitive deposits a tuple into a tuple space: for example, the request

```
out('p', 4.14)
```

will deposit the tuple `('p', 4.14)` into the tuple space, while the `in` primitive performs an extraction of a tuple from the tuple space. For example, the *tuple template*:

```
in('p', ?value)
```

will remove from the tuple space exactly one tuple whose first field is `'p'`, copying the contents of the second field into the variable `value`, as indicated by the `?` placeholder. The `rd` primitive performs exactly the same function as `in`, except that the `rd` primitive copies the fields of a tuple without removing the tuple from the tuple space. The `eval` operator is Linda's process creation mechanism.

Wood *et al* later proposed the introduction of 2 additional operators, `collect` (Butcher, Wood & Atkins 1994) and `collect-cp` (Rowstron & Wood 1996) to provide global synchronisation primitives to the tuple spaces. They accept 2 tuple spaces and a tuple template, and either move or copy (respectively) all matching tuples from the first tuple space to the second and return the number of tuples that were transfered (in the case of `collect`) or copied (in the case of `collect-cp`).

2 Motives For Customising Tuple Space Field Matching

All tuple space-based communication involves at least one pattern matching search on a tuple of a particular type signature during the `in`, `rd`, `collect` or `collect-cp` operation (see section 1.3). Currently, tuple field matching is based on pure binary matching – a bit-wise comparison of each field is performed to determine whether or not two fields match. Whilst binary matching is acceptable for primitive data types such as integers, this causes well known problems for simple float-based data types and more complex data structures.

2.1 Simplifying Distributed Programming

Tuple Space applications programmers have traditionally been encouraged to avoid attempting tuple matching operations on floating point numbers due to the issues involved in their precision (Carriero & Gelernter 1989). However if (in the case of floating point numbers) the matching expression for a float data type could be redefined from an expression resembling:

$$x_1 - x_2 = 0$$

to the usually recommended definition of equals for floating point numbers:

$$|x_1 - x_2| < \epsilon$$

(where ϵ is some tolerance), then tuple space programmers would have increased flexibility in the design of their parallel applications.

By allowing the redefinition of a field matching expression, some of the computation is moved from the application into the coordination language, which may help reduce the level of complexity of the distributed application. Additionally, the ability to redefine the field matching expression allows the act of coordination itself to reflect a necessary step in the algorithm, rather than merely as a necessary step in the provision of distributed infrastructure. This has the effect of simplifying the understanding of a distributed application.

Consider another example of a genetic algorithm – most implementations of a tuple space-based genetic alorithm create "task tuples" which are then accepted by worker tasks as a directive on what is the next computation to be performed (Zorman, Kapfhammer & Roos 2002), (Stracuzzi 1998). Using a customised matching expression for a "chromosome" data type, it becomes possible to `collect` the entire population that exceeds some fitness level; thus a single act of coordination can be considered in terms of the operation of a genetic algorithm ("Collect all those individuals with fitness above the currently highest individual") rather than in terms of the direct provision of distributed infrastructure ("You will calculate the fitness function for this individual").

2.2 The Next Step

Some researchers have noted that opportunities exist once tuple spaces divest themselves of strict pattern matching (Broadbery & Playford 1991). Some steps towards this concept of a more "programmable" tuple space have already been taken – for example, the work performed on $\mathcal{ACLT}$ by (Denti, Natali & Omicini 1997) and the ReSpecT environment by (Ricci, Omicini & Viroli 2002). These environments enable a tuple space to generate tuples when a particular action on a particular tuple occurs via *reactions*.

In an alternative view of tuple space semantics, the T-Cham environment (developed by Ma *et al*(Ma, Johnson & Brent 1996)), all tuples are considered to be molecules in a chemical reaction. Tuples that are inserted into a tuple space can cause other tuples to be created and destroyed.

If reactions (both as a response to a tuple space operation, as well as a reaction to the presence of particular tuples) to tuple space operations are occuring, then we also believe it is reasonable to argue in favour of the next step forward – mechanisms by which the reactions themselves can be controlled – by the customisation of field matching expressions.

Continuing the example in section 2.1 of a genetic algorithm, combining the ability to redefine field matching with the ability to produce additional tuples as a reaction (i.e., create mutated "chromosome" tuples as they enter the tuple space), a genetic algorithm implementation has the potential to become considerably simpler to understand and implement.

3 Issues and Approaches

Given our arguments and the fact that VLOS is tuple space-based, it is reasonable to expect that VLOS should be responsible for allowing programmers to redefine field matching expressions. However, allowing the user control over aspects of an operating system function creates new challenges, particularly with regards to security – it is now necessary to ensure that any code presented to the operating system by the user does not contain any unbounded loops, or attempts to violate the bounds of the tuple field being compared. Two possible solutions were considered – interpreted or compiled code.

Interpretation involves the code for the tuple field matching remaining as some form of intermediary code between human-readable and machine code, with an additional VLOS subsystem being responsible for executing the code on the matching expression's behalf in a Java-style virtual machine environment (Lindholm & Yellin 1999). Interpretation was not adopted due to its likely performance penalty and more importantly, the complexity (and effort) of implementation, for a language whose ultimate purpose is to return a `TRUE / FALSE` (non-zero or zero) value.

The use of a simple compiler with a dedicated language has the advantage of allowing user-based code to be used by the kernel at the full speed of native machine code with minimal modifications to the existing VLOS kernel; evaluation functions are executed directly, as opposed to the possibly complex initialisation for an interpreted language environment.

The decision to use a kernel-based language compiler is not without precedent – for example, the BSD Packet Filter (McCanne & Jacobson 1993) is effectively a system by which an assembly-like language is provided to the (BSD) kernel to control user-level packet capture.

An issue that must be addressed when allowing user-level code to be executed inside the kernel is of authentication and the key question of establishing trust between the kernel and an outside source of compiled code. Since any scheme to authenticate a user-level program acting as compiler would be complex and most likely troublesome (Thompson 1984), it was decided that the simplest way to ensure a trustworthy compiler was to have an instance of a compiler built into the kernel and accessed via a system call. This solves the problems associated with trusting the compiler used to produce the code, as well as ensuring that only a suitably secure language is used.

When selecting a language to use, most existing languages were discounted due to the size that such a compiler was likely to take when compared with the small subset of the language that would actually be used; nearly all forms of flow control would need to be removed to eliminate the possibility of infinite loops. When taking into consideration the likely size of such a language (which would only need a few key instructions), it was decided that the construction of a small purpose-built language was the best time-vs-reuse tradeoff.

4 The Solution: MiniMe

The MiniMe (MINIature Matching Expression) language provides only a single data type, the 32-bit integer. Many aspects of MiniMe have a strong resemblance to the C programming language, due in part due to the need for MiniMe compiled expressions to interface with the VLOS kernel, which is being developed in that language.

4.1 General Syntax

The syntax of MiniMe language statements is very simple, consisting of consecutive diadic statements,

`identifier = operand` *`operation`* `operand.`

where operands are identifiers (variables), or numeric constants.

The valid assignment operations available to MiniMe programs are shown in table 1. Note that the divide and modulus operations are the only operations that are capable of performing actions that are capable of causing an exception by providing a second operand of zero (and inducing a divide by zero exception). As such, these operations are checked beforehand; if the second operand being used is zero, then the evaluation of the matching expression is immediately terminated with a FALSE result.

Operator	Description
$+$	Addition
$-$	Subtraction
$*$	Multiplication
$/$	Division
$\%$	Modulus / Remainder
$\gg$	Shift right
$\ll$	Shift left
$\&$	Bitwise-AND
$\vert$	Bitwise-OR
$\hat{}$	Bitwise-XOR

Table 1: A list of all available basic operations in MiniMe. The modulus and divide operators compile into code that checks the operands to ensure that they are legal values before the assignment operation is performed – ensuring no exceptions are generated by an attempt to divide or modulus by zero.

Standard relational operators (eg. equality comparison) are also available under VLOS, and are listed for completeness in table 2. Any given relation evaluates to either 0 or 1, depending on whether the relation holds.

Three unary operators exist: the bitwise-complement (NOT) $\sim$, as well as the `template(index)` and `candidate(index)` operators – these return a 32-bit number corresponding to the contents of the field starting at an offset of `index` bytes into the field, of the tuple template's and the tuple being evaluated, respectively. These operators also check the boundaries of the field data types, to ensure that no attempt is made to exceed the boundaries of the field.

Relation	Description
$>$	Greater Than
$>=$	Greater Than Or Equal
$<$	Less Than
$<=$	Less Than Or Equal
$==$	Equality
$!=$	Inequality

Table 2: A list of the available relational operators available under VLOS. These operators evaluate to either zero (0) or one (1), depending on whether the relation holds.

4.2 Flow Control

The MiniMe language currently supports two main methods of controlling the flow of execution, the standard conditional block statements as well as a fixed-loop construct.

The syntax of the conditional block statments is of the form:

```
if ( condition )
   condition TRUE block
else
   condition FALSE block
end
```

Where *condition* is an expression comprising of one of the relation operators described in table 2. Each conditional block can consists of any number of valid MiniMe language statements, including nested conditional if-else-end blocks.

One of the important issues when allowing user-code to be executed in kernel space is the prevention of code attempting to access areas of memory outside the two fields under consideration. In order to address this concern, the only loop construct available under MiniMe is a fixed iteration instruction `foreach`. The usage of the `foreach` instruction is:

```
foreach Identifier
   Code Block
next
```

The number of iterations executed by the `foreach` construct is fixed to the size (in bytes) of the field type being examined. Two additional reserved words are provided: `TEMPLATE` and `CANDIDATE`, which refer to a single byte indexed via the loop counter variable. The identifier used to initiate the loop is available for (read-only) use within the code block specified, however indexing into the two arguments under consideration is fixed by the compiler, ensuring that no attempt can be made to exceed the bounds of the field type.

Control over `foreach` loops is also possible via the `break` and `continue` statements, which operate in identical fashion to their C counterparts (terminating either the entire loop or just the current iteration, respectively).

4.3 The Compilation Process

The compilation process is quite simple; the MiniMe subsystem consists of a single system-call with the following API:

```
int miniMe (Capability type, CString
code)
```

where `type` is a capability that refers to the field type whose matching expression is being changed, and `code` is a string containing the entire MiniMe expression. Note that due to the fact that the miniMe compiler is not a user-executable program and as such, all field matching rule redefinitions *must* be specified to the VLOS kernel in this manner, irrespective of the user application's host language.

The `miniMe()` system call is first subject to the scrutiny of VLOS' capability security system (Chung & McDonald 2002*b*). A task attempting to redefine the matching rule must be in possession of a valid capability allowing that task to do so.

Once the capability holder has demonstrated that it has the authority to use MiniMe on a field type, the matching expression is compiled into machine code, and used whenever matching on that field type must be performed.

5 Some MiniMe Examples

The following examples illustrate how the miniMe language can be used to improve the flexibility of the tuple space-based distributed application. The first example is an integer version of the example problem of redefining equality to incorporate a tolerance value, while the second example illustrates how the matching rule could be redesigned to match chromosomes in a genetic algorithm whose fitness is above that of the tuple template's.

5.1 Integer Equality With Tolerance Value

The following example performs an integer version of the example problem of redefining "equals" to mean that the difference between two numbers is less than some pre-specified tolerance value (in this case 1):

$$|int_1 - int_2| \leq 1$$

```
numA = template(0);
numB = candidate(0);
if (numA > numB)
  ReturnVal = numA - numB;
else
  ReturnVal = numB - numA;
end
return ReturnVal <= 1;
```

5.2 Genome Fitness Value

Consider a simple genetic algorithm whose chromosome is declared (in C-syntax) as follows:

```
struct Genome
{
  UInt32 g;
  UInt32 h;
  UInt32 i;
};
```

and whose fitness function is

$$f(x) = g^2 - hi$$

the following snippet of miniMe code redefines the matching rule of the chromosome field type to:

$$f(A) \geq f(B)$$

where $f(x)$ is the fitness function as evaluated over a chromosome x, A is the field from the tuple template and B is the field of the candidate tuple being evaluated.

```
numGA = template(0);
numHA = template(1);
numIA = template(2);
valA = numGA * numGA;
valAHI = numHA * numIA;
valA = valA - valAHI;

numGB = candidate(0);
numHB = candidate(1);
numIB = candidate(2);
valB = numGB * numGB;
valBHI = numHA * numIA;
valB = valB - valBHI;

return valB <= valA;
```

5.3 N Dimensional Manhattan Distance

The Manhatten Distance metric is a commonly used metric in many areas of computer science, including signal processing and path planning. Consider a tuple field of the form:

```
Int8 coordiantes[N];
```

where N is some fixed integer. In this example, the field matching rule is designed to only select fields whose N dimensional Manhattan Distance is less than or equal to some constant value (in this case 5).

```
result = 0;
foreach I
   xc = CANDIDATE;
   xt = TEMPLATE;
   if (xt > xc)
      difference = xt - xc;
   else
      difference = xc - xt;
   end
   result = result + difference;
next
return result <= 5;
```

6 Distributed MiniMe Propagation Protocol

One of the key concerns with the use of MiniMe is the ability to propagate changes to the matching expression across all nodes in the VLOS cluster. In order to achieve this the Transaction Manager (Chung & McDonald 2002*a*) is used to develop a MiniMe-specific protocol.

When a task attempts to redefine a field type's matching expression, the local compilation described in section 4.3 is followed prior to distribution. If local attempts at compilation are successful, then the originating task is blocked. Figure 1 describes the interaction between the different message types that are described below:

MiniMe-define – this message contains details of the actual field data type that is being redefined, along with the MiniMe code (as a string) for the new matching expression.

MiniMe-compiled – acknowledges not only the receipt of the MiniMe code, but also informs the originating node that the field type in question has had its matching expression redefined on that node.

MiniMe-error – used when a node is unable to successfully redefine a field type's matching expression.

MiniMe-cancel – used to abort an attempt to redefine a field type. Typically this is done when a node responds to a MiniMe-define with a MiniMe-error.

MiniMe-commit – used by the originating node when all nodes have sent back MiniMe-compiled messages, ensuring that if a single node is unable to redefine a field matching expression the process can be aborted.

MiniMe-ACK – used by responding nodes to confirm that the matching rules for those field data types have been redefined.

Once the originating node has received a **MiniMe-ACK** message from all nodes, the originating task is unblocked and allowed to continue executing. The blocking-while-redefining semantics have been intentionally designed to ensure that a blocked task does not prematurely start performing `in` and `rd` operations before the matching rule has been globally propagated. All attempted to redefine matching rules are done on a first come first served basis – if two processes attempt to redefine a field matching rule, then the rule definition from the process that made the request last is used.

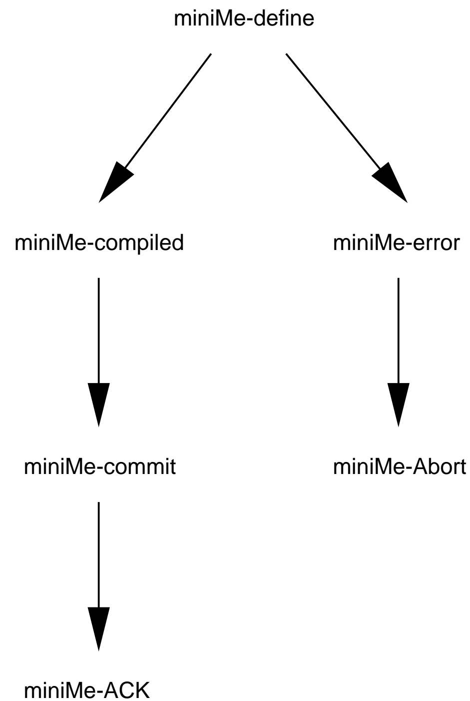

Figure 1: An illustration between the message types when propagating MiniMe redefinitions to other nodes in the cluster. If a node reports back with **MiniMe-error**, then a **MiniMe-abort** command is used to prevent the other nodes from redefining their matching expressions. When the originating node receives a **MiniMe-ACK** from all nodes in the cluster, execution of the application task is resumed, ensuring that the application only proceeds when the matching rule has been redefined globally.

7 Conclusions

We have presented strong motives in favour of allowing tuple field matching expressions to be customisable by the distributed applications programmer – by moving some of the computation from the application to the coordination, programs can be simplified, reducing their complexity.

The MiniMe matching expression language was presented as a potential solution to the need for customisable tuple field matching expressions. It uses a small custom-designed language compiler that is integrated into the VLOS operating system, to allow programmers to specify new field matching expressions.

To ensure that MiniMe – overridden matching expressions are propagated throughout the VLOS cluster, VLOS' Transaction Manager is used to firstly transfer the redefined expressions to all other nodes, and secondly to receive confirmation from all nodes. Tasks redefining a field's matching expression are blocked until confirmation of redefinition is received from all nodes.

7.1 Future Work

MiniMe is an initial implementation of a matching expression language for tuple space-based field matching; there are still several key areas that require continued development to increase the convenience of the language.

7.1.1 Global State Preservation

The example in section 5.2 of miniMe code for selecting chromosomes that are better than an existing member of the population highlights the current lack of any global state information, which would assist in the reduction of any computation involved in the selection of matches. In the example given, the existence of global state information would allow the fitness of the template field to be calculated once only.

Other possibilities for global state preservation include the ability to apply metrics across an entire tuple population, and use the `in` or `rd` operators to select the tuple with the optimal metric reading, rather that specifically a direct match.

7.1.2 Reactive Tuple Space Support

The work on reactive coordination media, such as $\mathcal{ACLT}$ allows additional tuples to be generated in reaction to a tuple being deposited (`out`'d) or extracted (`in` or `rd`) (Denti et al. 1997). If the MiniMe language was extended to allow the full range of tuple space operators to be executed whilst a field matching was being evaluated, this would combine the flexibility of the current MiniMe-based matching system with the reactivity of $\mathcal{ACLT}$.

References

Binning, B. (1999), Is Linda the foundation for the operating system of the future?, *in* 'Southwest Business Symposium, University of Central Oklahoma'. Available from `http://backofficesystems.com/tips/paper/linda`.

Broadbery, P. & Playford, K. (1991), Using object oriented mechanisms to describe Linda, *in* 'Linda-Like Systems and Their Implementation'. Technical Report TR-9113, Edinburugh Parallel Computing Centre.

Butcher, P., Wood, A. & Atkins, M. (1994), 'Global synchronisation in Linda', *Concurrency: Practice and Experience* **6**(6), 505–516.

Carriero, N. (1987), Implementation of Tuple Space Machines, PhD thesis, Department of Computer Science, Yale University. Also available as Technical Report RR-567.

Carriero, N. & Gelernter, D. (1989), 'How to write parallel programs: A guide to the perplexed', *ACM Computing Surveys* **21**(3), 323–357.

Chung, V. L. & McDonald, C. (2002*a*), The continuing evolution of VLOS, *in* 'International Conference on Parallel and Distributed Processing Techniques and Applications', CSREA, pp. 1385–1392.

Chung, V. L. & McDonald, C. (2002*b*), The development of a distributed capability system for VLOS, *in* 'Asia-Pacific Computer Systems Architecture Conference', ACS, pp. 57–64.

Davies, N., Wade, S., Friday, A. & Blair, G. (1997), Limbo: A tuple space based platform for adaptive mobile applications, *in* 'Proceedings of The International Conference on Open Distributed processing / Distributed Platforms', IEEE.

Denti, E., Natali, A. & Omicini, A. (1997), Programmable coordination media, *in* 'Coordination Languages and Models (Coordination '97)', Springer, pp. 274–288. ISSN 0302-9743.

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A. & Shivers, O. (1997), The Flux OSKit: A substrate for kernel and language research, *in* 'Proceedings of the 16th ACM Symposium on Operating Systems Principles'.

Lindholm, T. & Yellin, F. (1999), *Java Virtual Machine Specification*, 2nd edn, Addison-Wesley. ISBN 0201432943.

Ma, W., Johnson, C. W. & Brent, R. P. (1996), Concurrent programming in T-Cham, *in* 'Proceedings of the 19th Australasian Computer Science Conference', pp. 291–300.

McCanne, S. & Jacobson, V. (1993), The BSD packet filter: A new architechure for user-level packet capture, *in* '1993 USENIX Conference'.

Picco, G., Murphy, A. & Roman, G.-C. (1999), LIME: Linda meets mobility, *in* 'Proceedings of the International Conference of Software Engineering'.

Ricci, A., Omicini, A. & Viroli, M. (2002), Extending ReSpecT for multiple coordination flows, *in* 'International Conference on Parallel and Distributed Processing Techniques and Applications', CSREA, pp. 1407–1413.

Rowstron, A. & Wood, A. (1996), Solving the Linda multiple `rd` problem, *in* 'Proceedings of COORDINATION '96. First International Conference on Coordination Models and Languages', Springer, pp. 337–367.

Stracuzzi, D. (1998), 'Some methods for the parallelization of a genetic algorithm'. `http://www.cs.umas.edu/~stracudj/genetic/dga.html`.

Sun Microsystems (2000), 'JavaSpaces(TM) technology'. Web page located at `http://java.sun.com/products/javaspaces/`.

Thompson, K. (1984), 'Reflections on trusting trust', *Communications of the ACM* **27**(8). Turing Award Lecture.

Wilson, G. (1991), Linda-like systems and their implementation, Technical Report 91-13, Edinburgh Parallel Computing Centre.

Wyckoff, P., McLaughry, S. W., Lehman, T. J. & Ford, D. A. (1998), 'TSpaces', *IBM Systems Journal* **37**(3).

Zorman, B., Kapfhammer, G. M. & Roos, R. S. (2002), Creation and analysis of a JavaSpace-based distributed genetic algorithm, *in* 'International Conference on Parallel and Distributed Processing Techniques and Applications', CSREA.