Coordination with Attributes

Alan Wood

Department of Computer Science
University of York
wood@cs.york.ac.uk

Abstract. This paper addresses the opportunities for, and effects of, incorporating object attributes in LINDA-like tuple-space systems. The focus is on exploring how the coordination attributes are affected by this enhancement, and investigating in what ways the classical LINDA model needs to be modified. Particular emphasis is placed on the potential *practical* gains to be expected in both performance and expressibility, with consideration being given to methods for implementing attribute frameworks in open distributed systems.

1 Introduction

The concept of coordination could not be claimed to be 'new' — even in the narrow domain of computing, and its sub-domain of concurrency, the need to coordinate things has been around a long time. However, it is only relatively recently that coordination has been identified as a topic for independent study, and which promises to unify several apparently disparate fields in computing.

In their 1992 paper [6], Carriero and Gelernter crystalised this idea by identifying the *two* functions of a "full" programming language as *computation* and *coordination*, and suggesting that these two aspects of a language are *orthogonal*. Consequently, the study of coordination may be undertaken *independently of* any host computation technology.

Once the idea of coordination being distinct from computation is accepted, the question arises as to what the coordination *structures* are. These would be analogous to the control and data structures in the *computation* half of a language. Some coordination structures have been in common use for many years: semaphores, monitors, signals (in the UNIX sense). Others have been emerging from the increased *use* of parallel and distributed architectures ... examples might be light-weight threads, message-passing, or barrier-synchronisation.

This paper addresses coordination problems in *open distributed systems*, which are to be contrasted with *closed* systems. The distinction lies in the fact that in an open system agents (processes) can 'join' or 'leave' during the lifetime of the system with no prior knowledge of their (potential) existence necessarily being available to the other agents. From the point of view of a programmer, or more importantly a *compiler*, full information about the behaviour of the system while his/her/its process is executing cannot be determined. In a closed system, however, such information is in principle available, and can be used by a

compiler, say, to optimise the process ensemble. The open system model is that of the internet; the closed system is that of a subset of processors in a box, with only a single parallel program running.

1.1 Object Attributes

Objects have a number of properties associated with them — indeed, it is the set of properties that *define* an object. In a computational context, perhaps the most fundamental properties are *value* and *type*. However, there are other properties which are normally associated with objects including their *scope* (or visibility), updatability, or other accessibility features. Although these are some obvious concrete examples, attributes should be seen as a collection of abstract properties associated with objects. This view admits the possibility of having attributes which may be user-defined (and, consequently, would need to be user-maintained), rather than fixed by the system.

1.2 LINDA

Since this paper addresses some *general* aspects of coordination, while suggesting practical ways of implementing them, trying to relate the ideas to the diversity of current coordination languages would almost certainly obscure the issues. Therefore, only one coordination system will be used as a focus for the ideas presented — LINDA. There are two main reasons for choosing LINDA. Firstly, it is (probably) the earliest explicitly 'coordinative' language, and (certainly) one of the best-known of the current coordination languages. Secondly, it is a very flexible model which subsumes most others (although, as shall be seen later, it still has some deficiencies in its coordinative features).

However, many of the issues raised here are not LINDA-specific — indeed, much of the point of the paper is in identifying 'purely' coordinative issues which exist independently of any specific coordination model.

The LINDA model considered here is the 'classical' one [5], with `in`, `out`, `rd`, and `eval` primitives extended to work with multiple tuple spaces.

1.3 Motivation

In LINDA-like systems, agents (processes) coordinate by means of *generative communication* [8]. This term is intended to suggest that the 'carrier particles' of coordination are created by agents, but then have existence independent of their creators.[1] Using this medium, LINDA is able to provide (nearly) all the coordination mechanisms normally used in parallel processing situations. There are other ways of achieving coordination which do not fall neatly into the category of data communications, as will be seen later. However, why should they be needed? There are two reasons:

[1] This is in contrast to conventional synchronous message-passing communication models in which the message only has a fleeting existence at the instant of synchronisation. It therefore cannot exist independently of the sender and receiver.

1. There are coordination tasks which are *not* supported by the model.
2. There are also some coordination tasks arising quite naturally which, although supported in the 'classical' LINDA framework, give rise to significant inefficiencies[2] under the classical model. This in itself is not a sufficient reason to support any proposed modifications to the LINDA model, since any change introduces extra complexity which must be weighed carefully against the potential benefits.

Examples of both of these types of task will be described in Sect. 2.1.

2 The Design Space

The objects that we shall be concentrating on are *tuple-spaces*, and *tuples*, although it is possible also to consider tuple *elements* as being candidates for attribute-based coordination. There is a wide variety of possible object attributes that could be incorporated in an attribute-based coordination scheme, and some of the more unusual candidates might include persistence and fault-tolerance, types, and costs. However, in this paper we shall focus on the widely-understood access attributes, which are analogous to those found in conventional file-systems.

The choice of objects, together with the set of attributes gives rise to a 'design space' formed from the object and attribute 'axes'. This space then enables the issues of semantics, implementability, expressibility etc. to be addressed when assessing which modifications to the LINDA model are required.

2.1 Access-control Attributes in LINDA

The object axis of the design space is clearly defined above — tuples, tuple-spaces, and tuple elements. The attributes will include the right to perform the appropriate LINDA access operations on the objects — `in`, `rd`, and `out` — which approximately correspond to read and write permissions on files in conventional systems.[3] However there are other operations which are performed on objects in a LINDA-like system which must also be considered.

Figure 1 displays the part of the design space determined by these objects and attributes. A tick (✓) implies that a sensible meaning can be assigned to that attribute:object pair; a bullet (•) suggests that that pairing is incompatible; a question mark (?) indicates those pairings for which it is not clear that there is a sensible use, but might prove useful in some situations.

[2] Here, 'inefficiency' is to be taken in a broad sense, which will include run-time performance, but will also encompass *programming* or *software engineering* inefficiencies. An example of the latter would be a perceived need for tuples output by an agent to have additional information 'just in case' some other agent *might* require this extra information.

[3] This correspondence is not exact since there is now a distinction between copying content from an object, and *removing* content, which has no direct analogue for file systems.

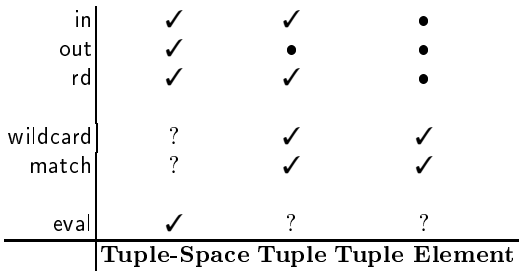


Fig. 1. The design space

The attributes can be placed in the following groups based on their effects:

out, in, rd It is easy to see that out, rd and in attributes applied to tuple-spaces make sense — if a tuple-space has that attribute then an agent can access it accordingly, otherwise that type of access is prevented.
With individual *tuples*, only in and rd can apply — corresponding to permission to remove or copy that tuple, respectively — since outing to a tuple is not defined.[4] Similarly, neither in nor rd nor out can be applied to tuple elements.

wildcard and match
The lack of wildcard permission on a tuple element would prevent an `in` or `rd` operation form retrieving that tuple if the operation's template specified a *formal* for that element. By extension, this would be applicable to whole tuples and tuple-spaces, affecting actual:formal matching on all elements of a tuple, and all tuples, respectively.
The match attribute is the dual of wildcard: its lack would imply that *only* actual:formal matches would succeed, and actual:actual matching would not.

eval
Permission to place an active tuple into a tuple-space is controlled by the eval attribute. This is clearly meaningful when applied to tuple-spaces, although it can also be of use when applied to individual tuples, as will be discussed later.

2.2 Semantic Implications

Extending the LINDA model with an attribute framework will of course change the semantics — there now exist properties that could not be achieved previously. However, these attributes might be purely *extensions*, that is they do not modify the semantics of the underlying model, or they might necessitate a revision of the underlying semantics.

In the first group — those which do *not* change the model — are all of the 'ticked' entries in the first five rows of the design space, except for the out attribute. For instance, due to the non-determinism in LINDA, a process would not

[4] If there were some sort of `update` operator, then this attribute could be applied to a tuple — it would prevent or allow updating.

be able to detect the difference between a `rd` which is blocked due to a tuple being unavailable in tuple-space, and being blocked due to a tuple (or tuple-space) denying rd permission. Similarly for the in, wildcard, and match attributes. However, with out the situation is different. If a tuple-space denies out permission to a process, then one of two outcomes could result depending on the implementation. Either the `out` operation *fails*, in which case the host *computational language* must have some exception mechanism to deal with this, or the operation *blocks*. Both of these represent changes to the user's view of the model's semantics. In the first case, the independence of the computational and coordination languages is broken. In the second, a fundamental alteration in the semantics of `out` has been forced — `out` *never blocks* in the standard LINDA model. Unfortunately, there is no clear choice between the two alternatives — to rely on integrating the LINDA coordination system with a host's exception mechanism would, at least, mean that existing LINDA programs would need modifying. On the other hand, enforcing a 'blocking `out`' semantics would introduce subtle (and not so subtle) problems in using existing techniques for analysing and designing LINDA process ensembles.[5] The problem of choosing between the two alternatives will be side-stepped in this paper by considering that processes are so well-behaved that there will be no attempt to do an operation that would be prevented by an attribute. In this way we shall be able to concentrate on the new coordination structures made available by incorporating an attribute framework into LINDA. There is, therefore, further work to be done in this area.

The following sections will consider only the basic access-control attributes in the examples. However, it is hoped that by discussing some of the other aspects of the basic 'design space', a feel for the potential improvements in expressiveness brought about by the introduction of an attribute framework can be gained.

3 Attribute-Supported Coordination Structures

The purpose of a coordination language is to enable the expression of coordination structures in the same way that a computation language supports the definition of control and data structures — both are means for specifying *algorithmically* the interactions between component objects that are composed to achieve a task. Deciding what constitutes a coordination structure is, however, rather awkward since there is no body of coordination theory to compare with the theory of computation. Consequently, an informal definition will be used here — a coordination structure describes and governs the temporal and spatial interactions between agents, without reference to their internal computational structure.

This formulation leaves some 'grey areas' at the interface between computation and coordination, but will suffice for the purposes of this paper since the examples used will be firmly coordinative.

Given the the difficulty in defining what constitutes a coordination structure, it is perhaps rather unfair to suggest that LINDA is deficient in its ability to ex-

[5] It would, for instance, affect questions of deadlock and termination.

press some coordination structures — it could be argued that such structures are *not* coordination structures, and therefore that LINDA is not deficient. However, at least one facility which appears in other non-LINDA coordination languages (for example IWIM [3]), the provision of private or point-point communication, cannot be implemented using the classical LINDA model. Consequently, there is a *prima facie* case that the classical LINDA model requires some enhancement. The rest of this section investigates such an enhancement.

The following two sections give examples of the types of coordination patterns or structures that become available once an attribute system is incorporated into LINDA. The first section addresses those structures that are generated by the basic access-control attributes discussed in Sect. 2.1. Section 3.2 touches on attributes which do not address access control, but which may affect the ways in which a process ensemble behaves.

As mentioned before, all these coordination patterns can be split into those which could be achieved before but are now more efficient, and those which were previously unavailable.

3.1 Access-control Attributes

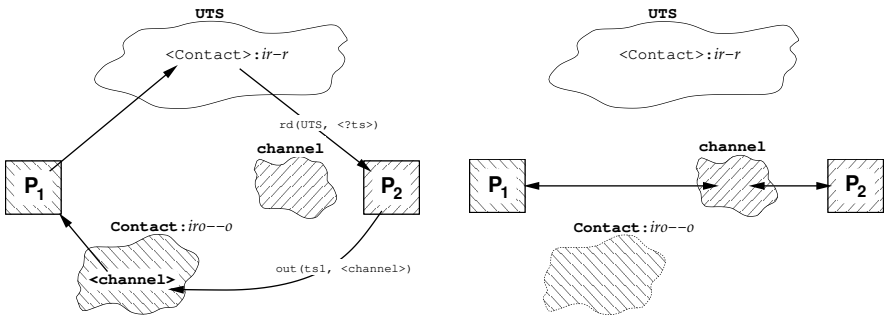
In this section, two attribute-supported coordination structures are described. In order to do so without preempting the discussion in Sect. 4 on implementation choices, the notation used for expressing attributes is simplistic: only the attributes applicable to the structures are shown, and these are in, rd, and out, which are represented by their initial letters. Attributes are grouped into the two *domains* which are applicable — the owner or creator, and all others, respectively. In a full system there could be a large number of domains, defined in a variety of ways (see the discussion in Sect. 4).

In the code examples, tuples are assigned access attributes by decorating the `out` operation's tuple parameter with the required attributes. Tuple-spaces are created with the `tsc` primitive, which is extended to take an attribute set as a parameter.

Private Channels — a new coordination structure in LINDA It is often the case that (subsets of) agents need to establish means by which they can communicate without any interference from other agents in the system. Many coordination languages have explicit structures for setting up such 'private channels' — indeed, private, point-to-point communication is the *only* means of coordination in most common coordination systems. In LINDA-like systems, however, the coordination model is based on *broadcast* communication through tuple spaces which is more general, but less 'secure'. The lack of security can be demonstrated by considering a pair of agents which wish to coordinate by the communication of some tuple. Agent 1 `out`s the tuple into a tuple-space, and Agent 2 `in`s the tuple, *by specifying a matching template*. Unfortunately (for the purposes of a private conversation), there is nothing to prevent a third agent removing the tuple — and this could happen *accidentally* (as well as maliciously).

This problem is not solved by the mere addition of multiple tuple-spaces — some means has to be provided to make such tuple spaces *privately scoped*. The solution is to make use of access control attributes. It will be seen that attributes attached to both tuples and tuple-spaces will be needed.

A sketch of the core of the method is shown in Fig. 2, with the skeleton of the required code (in a `C`-like syntax) given in Fig. 3. One agent (process P_1 in the example) broadcasts that fact that it is willing to engage in a private conversation by placing a tuple containing a reference to a tuple-space (of which it is the creator) in `UTS`. The tuple has `rd`-only access to all other agents. This prevents any accidental or malicious *removal* of the tuple, which is the first part of the guarantee against interference in setting up the private channel.



(a) Establishing contact (b) Channel created

Fig. 2. Creating a private channel using access attributes (shading matches a tuple-space with its creator.)

The tuple-space, `Contact`, is intended to be the means by which the other agent (P_2) informs P_1 of a reference to the private-channel (which is a tuple-space known only to the two agents involved). `Contact` is created with `out`-only access with respect to all other agents, so that its contents cannot be altered by any other agent. This gives the second part of the guarantee against accidental or malicious interference in the process of setting up a private channel — no third-party would be able to gain knowledge of a reference to the 'private-channel' tuple space which is passed in a tuple in `Contact`.

Once `channel` has been created and its reference passed to P_1 via `Contact`, P_1 and P_2 have a secure means for coordination.

Global Software Distribution — an application of access attributes This example of the use of access-control attributes shows how private channels, scoped tuple-spaces and controlled access to tuples can be combined to

```
Contact = tsc( iro--o );
out( UTS, <Contact>:ir-r);
channel = in(Contact, <?ts>);
/* Use channel */
```

(a) Process P_1

```
channel = tsc( iroiro );
contact = rd( UTS, <?ts> );
out( Contact, <channel>:irir );
/* Use channel */
```

(b) Process P_2

Fig. 3. Skeleton code for Fig. 2

provide an architecture for the 'hard media free' distribution of software. In addition, if some further technological problems were to be solved together with the opportunities for *caching*, this architecture could provide a platform for the fully distributed operating systems of the 'Network Computer' type [16], or the 'Millennium Project' [4].

When a user acquires a piece of software from a producer, whether by paying money to a commercial supplier or by copying it from a free source, several things must happen. Firstly, the potential user must know *how* to acquire it — its existence, who can supply it, how to contact the supplier, etc. Secondly, a request for the software must be made to the supplier — this will involve different protocols depending on whether or not money must change hands, whether the software is shareware or freeware, whether or not a formal licence is required, etc. Next, the user must obtain a copy of the software — normally on hard media if commercial, or electronically if free/shareware. Finally the user must install it.

This method of distribution works reasonably well, but it has several problems including:

- Bug fixes — as bugs are found, all previously distributed copies must be replaced. In practice this is so difficult logistically, and commercially, that it is never done.
- Media Waste — each new customer requires a new piece of media which is often discarded after installation.
- Piracy — any distributable medium, and its contents, can be cloned.

Now consider how a tuple-space based system, with access-control attributes, might be used to overcome these problems, while admitting some new possibilities. Figure 4 outlines the skeleton of such a system.

The configuration in the lower part of Fig. 4, involving the customers C_1 and C_2, the Production agent (part of the vendor), and tuple-space `V1`, represents the goal — a number of users of version 1 of the software, which is stored as a set of tuples in `V1`. `V1` can only be modified by Production, and is only known to the vendor and the authorized customers.

The upper part of the figure shows how a new customer would acquire access to the software. The situation shown is that C_3 wants version 2 of the software, in

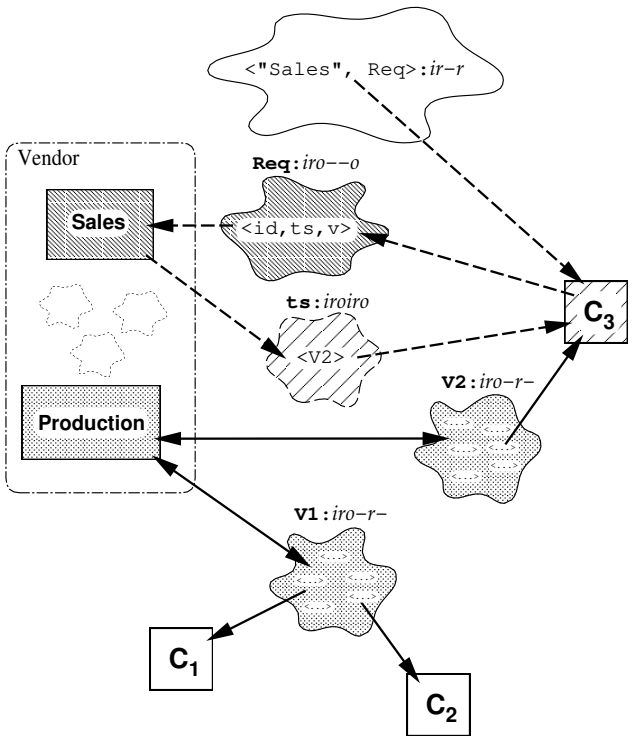


Fig. 4. The use of access attributes in a global software distribution system. (Shading is used to associate tuple spaces with their creator.)

which case it would communicate with the Sales agent of the vendor via a private channel established using the protocol described in Sect. 3.1. The information passed to Sales would be the customer's `id`, the version of the software wanted, and the private-channel tuple-space reference. Once Sales is satisfiedthat C_3 is entitled to the software, it would obtain a reference to the appropriate tuple-space from Production (using private channels local to the Vendor), and pass this to the customer. Then C_3 would be in a situation similar to customers C_1 and C_2, but with access to version 2.

This method for distribution does not depend on the actual representation of the software as tuples in the tuple-space. In the simplest case it would be possible to have the entire piece of software as an element of a tuple. Then authorized customers (those with a reference to the tuple-space) could either copy the software to local storage, or run the software from the tuple-space (in effect 'copying' it by means of their system's program loader). This would require no fundamental alteration to the way in which the software would be installed and used ... only the method of delivery would have changed.

However, by representing the software as a *set* of tuples, a number of novel possibilities for software architectures can be considered. The tuples might then correspond to library routines (as in pre-'object-oriented' systems), objects, or services (as in operating systems). Some of the potential advantages of this arrangement are:

Bug fixes. When modifications to the software are required that don't imply a new full release, only those components that require correcting need be replaced. An *update* of a tuple (routine or object) would correspond to the Production agent removing the original version (using an `in`), and `out`ing the replacement. Therefore, a completely up to date version of the software would always be available to both existing and new customers — management of bug-fixes by the vendor would become relatively trivial.

Copy protection. A degree of copy protection — more accurately, copy *discouragement* — becomes available. It is likely that most of the tuples accessed by customers would contain 'interface' routines, or 'wrappers' which call other services/objects. Within a tuple-space based system, these lower-level routines would be accessed via tuple-space operations as would the routines that they require. Consequently, it would not be necessary to have the *complete* code for the software existing in the customer-accessible tuple-space — much of the system could exist in tuple-spaces local to the vendor which is only accessed by the *code* of the routines contained in the customer tuple-space. This has two effects. Firstly, since only the 'interface' tuples formats need be made known to the user, it would be practically difficult to take a copy of the *entirety* of the tuple-space, since this would require being able to specify templates to match *every* tuple, and this would not in general be possible. Secondly, even if a complete copy of the tuple-space's contents were able to be made, which is doubtful, there would be no simple way of obtaining copies of the lower-level routines.[6]

The execution model resulting from this system is similar to demand-paging — the routine (code) for the required service is loaded by `rd`ing the appropriate tuple. When the execution of this routine caused the retrieval of the contents of further tuples this would be equivalent to 'demand' loading the required code — the newly loaded code could then retrieve code-tuples itself, and so on. This could form the basis of a Network Computer style of operating system in which the operating system's services were represented as code contained in tuples.

However, currently this execution model would not be feasible since it implies a *single* source of the code, which would be frequently accessed by a very large number of agents. A single tuple-space could not cope. For this to work a very efficient caching and replication scheme would need to be implemented.

[6] This would at least require disassembling the routines' code to determine all the references to vendor's local tuple-spaces. Even then this would not necessarily be the end of the unethical copier's task, as the routines thus copied could call the contents of tuples in yet other unknown tuple spaces.

If the caching and replication problems were satisfactorily solved, then this execution model could be comparable in efficiency to the disk-based demand-paging schemes in common use.

3.2 Other Attributes

The previous sections have shown, in detail, how *access-control* attributes can be used to significantly enhance a system's coordinative vocabulary. It has been argued that, despite focusing on LINDA-like systems, incorporating such an attribute framework is *generally* useful in any coordination language. However, once the notion of attribute-based coordination is admitted, there are many attributes other than access-control that might be considered — often these will be more or less specific to the actual coordination platform being used. Two examples, which are of particular interest in LINDA systems are:

Ordering It is sometimes the case that the order of insertion of tuples into a tuple-space is important to an agent retrieving those tuples. For instance, if the tuple-space were to represent a keyboard buffer, then the order in which the tuples are inserted would reflect the order in which the keys were pressed. Consequently, the tuple-space (or its controlling LINDA kernel) would need to maintain sequence information. This could be specified as an attribute of the tuple space.

In the absence of attribute-supported ordering, this would have to be emulated when needed by the inserting agent producing a sequence number as an extra element in the tuples that it `out`s.

Other coordination systems will have similar examples.

Non-determinism LINDA's associative access mechanism is *non-deterministic*. However, this is not the same thing as *random* access, which is related to the *fairness* attributes of the system. Consider a process which repeatedly executes a `rd` of a tuple-space containing several tuples which match the `rd`'s template. LINDA's non-determinism ensures that the process cannot predict which tuple it will receive, but LINDA's semantics does not prevent the kernel supplying the *same* tuple every time.[7]

A random access scheme, on the other hand, would identify *all* candidate tuples and randomly choose one to be returned. This would guarantee fairness over a sufficiently large number of repeated accesses. In some circumstances an improved *performance* can be obtained with a random system compared with a purely non-deterministic implementation [7], Therefore, 'random choice' is an example of a useful behaviour-modifying attribute.

[7] This is not only semantically valid, but highly likely in practice — the 'natural' way for a LINDA kernel to represent a tuple-space is as a list. Then a retrieval operation would return the first tuple in the list that matches. Repeating the operation would merely find the same tuple again.

4 Implementation Considerations

So far attributes have been represented abstractly as a set of flags attached to representations of the objects. This has enabled discussion of the issues without being concerned with how the abstractions are actually to be implemented. In fact, if the only attributes were access-control, and it were only necessary to distinguish an object's creator from other agents, then the implementation would follow this representation. However, in the context of open distributed systems, it will be necessary to provide a level of attribute control that is both more general, and finer, than can be achieved with this simple model. This section discusses the implementation issues and alternatives in the open system environment.

4.1 Conventional Approaches

There are two approaches to the implementation of access (and other) attributes described in the standard operating systems literature (see for example [12, 17, 18]).

Access Control Lists The first approach is essentially that assumed in the earlier sections of this paperin which an object, generally a file or device, has attached to it a list of sets of attributes. Each set in the list represents the attributes that are appropriate for a given *domain* of agents when accessing the object. In UNIX there are three domains — the owner, a 'group', and all others. This method of implementing attributes is called the *Access Control List* (ACL). The exact means by which the domains are defined and maintained will vary from system to system.

Capabilities In the second method, each domain holds rights to the object's attributes that it requires, rather than the object maintaining a list of domains' attribute rights. If a domain needs to perform some action on an object, it requests the right to do so from the object. If the request is granted, the object returns an unforgeable 'passport' — a *capability*[8] — which identifies the domain, the object, and the rights granted. When an element of the domain (a process) needs to perform some action on the object, it presents the object with a copy of the capability along with the access request so the object can determine whether the process possesses the right to do the requested action.

Although the technique still encompasses the idea of a domain consisting of several elements, in practice it is normal to have a capability *per process* and to identify domains with processes.

These two techniques have several complementary attributes. Summarising the pros and cons (+ and − respectively) of ACLs and capabilities:

[8] As far as the domain's elements (processes) are concerned this would be some 'opaque' bit-string. For full details of how this is implemented, see the references [18].

ACLs

+ **Applicable to un-named objects.** Therefore they can be used with tuples, tuple elements, and tuple-spaces.
– **Require distributed domain management.** This limits the flexibility and increases complexity.
– **Require arbitrarily large lists attached to each object.** This is not practical, *especially* in an open system.
– **The set of applicable attributes is fixed by the system,** since each ACL element must account for the attribute *set* for that domain.

Capabilities

+ **Easily implemented in an open distributed system.** Agents store their own set of capabilities, putting no storage overhead on the objects to which they refer.
– **Cannot easily *withdraw* permission,** or otherwise modify a capability once granted.
– **Cannot be used with anonymous objects.** This limits their use to tuple-spaces alone.

Therefore there is no clear choice between the two methods for an open distributed tuple-space system, and so a compromise solution must be found. Since there is no choice but to use ACLs for tuples and tuple elements, it is proposed that the simple, two domain ACL system implicitly used in Sect. 3 be implemented — the domains being *creator*, and *others*.

For tuple-spaces, however, a capability system should be used for several reasons:

Firstly, the lack of 'domain flexibility' inherent in the use of ACLs for tuple-space contents can be overcome by designing process ensembles to use tuple-spaces (with capabilities) as the prime means of attribute-control thereby regaining access to the richness of domains that is a feature of capability-based systems. Secondly, user-defined attributes can easily be handled — a kernel needs only to know that an attribute *exists* in order for it to be able to grant, and verify, a capability for that attribute. Finally, a capability-based scheme is not restricted to a fixed set of attributes — this set may change over time, which is essential for an open system since its life-span might be very long. The protocols described in the literature for managing capabilities are quite straight-forward, and are easily transferred to a tuple-space based system.

Secondly, there are four important features that this capability scheme has which positively affect the implementation:

1. The kernel is only involved in generating and checking capabilities — initially deciding whether or not an agent is to be granted some attribute rights to an object is the responsibility of the object's creator.
2. All the information required to generate capabilities (object identity, agent identity, etc.) would naturally exist in any 'conventional' LINDA kernels.

3. Since all LINDA operations must be serviced by the kernel, the only overhead involved in incorporating a capability-based scheme is the extra communication costs due to the need to provide the capability in addition to the other parameters in an operation.
4. No information regarding capabilities need be stored in the kernel — the information contained in the capability itself is sufficient for the kernel to asses its validity for the specified object.

The remaining problem with capabilities is more awkward to solve — withdrawing an agent's capability for an object is difficult. Several methods have been reported which all result in revoking the capability of *all* agents for a particular object. Any finer degree of control seems very difficult to achieve. However, it is most likely that, in a LINDA-like system, the only need to revoke a capability would be when the creator (or one of its proxies — agents which have been given creator capability) needs to block access temporarily — for example, if it needs to 'freeze' a tuple-space when doing a series of operations. In this case, a request to the kernel to temporarily withdraw non-creator access permission would be sufficient — all processes would merely block until the permission were reinstated.

5 Related Work

The deficiencies in LINDA, in particular the inability to express directed, or point to point, communication have been noted by several authors. In general the response has been to propose entirely new systems (or existing systems extended to work in a distributed environment) which overcome the perceived deficiencies, but at the expense of losing the advantages that LINDA-like systems have over many of the proposals in the area of open systems.

An example of an attempt to solve the LINDA problems by improving LINDA, is that of Pinakis's Joyce-Linda [13]. The focus was solely on the directed communication (private channels) problem, and developed a solution which was based on a type of public-key encryption technique inspired by the capability-like 'port' concept in an early version of the Amoeba operating system. There are, therefore, strong links with the ideas presented in Sect. 4 regarding implementation of the attribute framework. However, Pinakis's solution had difficulties in communicating "tickets" (his proto-capabilities) through tuple-space securely, since a basic attribute of ticket values was that they could not be matched by formals in templates. Consequently he had to resort to the rather unsatisfactory solution of introducing a second special data type (tickets being the first) to which ticket values could be coerced, and hence matched by formals. Hence it can be seen that his implementation incorporated some of the attributes of the capability-based implementation of the attribute framework discussed here. Finally, the Joyce-Linda system "relies on the compiler and runtime system", and therefore would be difficult to adapt to the *open* systems problems addressed in this paper.

Two proposals taking the more radical approach to the LINDA deficiencies are those of Law Governed LINDA [11], and ActorSpace [2]. Minsky and Leichter's

motivation in developing Law Governed LINDA was to address LINDA's "inherent lack of safety", specifically citing its inability to support "pair-wise communication" (private channels). The reason why they do not abandon LINDA completely is that they regard it as "particularly suitable as a coordination model for open systems", and that it is by implication merely the lack of safety that prevents it from being more widely used. They identify the principal safety problem as the need for component processes in an ensemble to rely on "voluntary conventions" regarding tuple/template formats and hence tuple-space accessing. This mode of operation is just acceptable for closed systems, but cannot be expected to work in open systems. Therefore they propose to replace these voluntary rules with globally enforceable 'laws'. This is to be achieved by the system assigning a 'controller' to *each process* which mediates between the process and the system, thereby ensuring adherence to the built-in law.

Again there are parallels with the attribute framework described in earlier sections — in this case there is a rough duality between LGL and the attribute framework. Whereas LGL's integrity is maintained by the *agents* (or their controllers in conjunction with the system), in the attribute framework the control resides with the *objects* (in conjunction with the kernel). Which system has the performance/resource advantage would depend on the relative complexities of the controlling components and the relative numbers of objects and agents.

The LGL system is very powerful and the laws that can be expressed are extremely general. The authors do comment, however, that the efficiency of the system is still to be determined.

The ActorSpace proposal of Agha and Callsen [2] also criticises LINDA for its insecurity, but makes no attempt to fix LINDA, preferring to develop Agha's Actors [1] to work in the open distributed environment. The result is a new coordination language, rather more complex than LINDA, in which the focus is on a 'generalized' direct (point to point) communication based on a pattern matching mechanism. The patterns which are involved in the match in essence define attributes in the sense used in this paper. In their proposal, the authors make the case for attribute-based control of system activity, and choose to provide security by means of capabilities. Their work is therefore in several ways directly comparable to that described in this paper, only using a different coordination basis.

6 Conclusion

It has been argued that an attribute framework is necessary for complete coordination in open distributed systems. In particular, access-control attributes have been shown to be *essential* for LINDA-like tuple-space systems to be able to provide at least one coordination structure — private channels — that could not be achieved previously. Other desirable coordination attributes also emerge from such an attribute framework.

The implementation proposed is a mixed system combining both Access Control Lists when necessary, with Capabilities when possible. It has been argued

that a capability-based implementation is particularly natural in the context of LINDA.

The ideas discussed in this paper are being implemented in the latest in the series of LINDA kernels [14, 15] developed at York. This will also incorporate recent advances in distributed resource management (in particular, garbage collection [9]) and the confluence of Input/Output with the LINDA model [10].

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha and C.J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. *SIGPLAN Notices*, 28(7):23–32, 1993.

[3] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume LNCS-1061, pages 34–56. Springer-Verlag, 1996.

[4] W.J. Bolosky, R.P. Draves, R.P. Fitzgerald, C.W. Fraser, M.B. Jones, T.B. Knoblock, and R. Rashid. Operating System Directions for the Next Millennium. `http://www.research.microsoft.com/research/os/Millennium/mgoals.html`, 1996.

[5] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[6] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–106, 1992.

[7] D. Edwards. Genetic Algorithms in LINDA. Technical report, University of York, Department of Computer Science, 1997.

[8] D. Gelernter. Generative Communications in LINDA. *ACM Trans. Programming Languages and Systems*, 7:80, 1985.

[9] R. Menezes and A. Wood. Garbage Collection in Open Distributed Tuple-Space Systems. In *Proc. 15th Brazilian Computer Networks Symposium — SBRC'97*, 1997.

[10] R. Menezes and A. Wood. Incorporating Input/Output Operations in Linda. In *submitted to 30th Hawaii International Conference on System Science*, 1998.

[11] N.H. Minsky and J. Leichter. Law-Governed Linda as a Coordination Model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume LNCS-924, pages 125–146, 1995.

[12] J.L. Petersen and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[13] J. Pinakis. Providing Directed Communication in LINDA. In *Proc. 15th Australian Computer Science Conference*, pages 731–743, 1992.

[14] A. Rowstron, A. Douglas, and A.Wood. A distributed LINDA-like kernel for PVM. In *Proc. Euro PVM*, pages 107–112. Hermes, 1995.

[15] A. Rowstron and A.M. Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96*, volume LNCS-1123, pages 510–513. Springer-Verlag, 1996.

[16] D. Simpson. Who Needs a Network Computer. *Datamation*, 42(16):96–99, 1996.

[17] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[18] A.S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.