Law-Governed Linda Communication Model

Naftaly H. Minsky* minsky@cs.rutgers.edu
Jerrold Leichter leichter@cs.rutgers.edu

Department of Computer Science
Rutgers University
New Brunswick, NJ, 08903 USA

March 14, 1994

Abstract

Linda is a very high level communication model which allows processes to communicate without knowing each others's identities and without having to arrange for a definite rendezvous. Along with the considerable advantages of this high level of abstraction, Linda carries some serious liabilities: it is very unsafe, and is difficult to optimize. We propose to remove the first of these difficulties, and alleviate the second — while preserving the advantages of Linda — by means of the concept of law-governed architecture previously applied to centralized and message passing systems. We define a model for Law-Governed Linda (LGL) communication, and we demonstrate its efficacy by means of several illustrative examples.

1 Introduction

There is much which is attractive about the Linda communication model [Gel85], under which processes communicate by placing *tuples* in a shared and wide open *tuple space*, and by retrieving tuples from this space. The model *uncouples* communicating processes in both time and space [BA90] by allowing them to communicate without knowing each others's identities and without having to arrange for a definite rendezvous; it uses a single mechanism for both communication and synchronization; and it lends itself to various styles of communications, including message passing and broadcasting.

But the very generality of this model, and its high level of abstraction, carry with it some serious liabilities. First, Linda communication is unsafe, as unsafe as a market place in which all goods are dumped into the public square, free for anybody to grab as he sees fit. Second, Linda communication is very hard to implement efficiently, because of the lack of any *a priori* knowledge about who might store which tuples in the shared tuple space, and who might retrieve these tuples.

*Work supported in part by NSF grants No. CCR-8807803 and No. CCR-9308773

We propose to remove the first of these difficulties, and alleviate the second — while preserving the advantages of Linda — by means of the concept of law-governed architecture (LGA) [Min91]. Under this architecture the interaction of each process with the tuple space would be governed by the same set of rules, called the "law" of the distributed system. Such a law may secure[1] communication via the shared tuple space, as financial transactions in the market place are secured by societal laws; and it may facilitate some optimizations by providing *a priori* knowledge about the interaction of processes with the tuple space.

We start with a brief introduction to the Linda communication model, and with a discussion of its liabilities. In Section 3 we introduce our model for Law-Governed Linda (LGL), which is an adaptation of the model for distributed law-governed architecture introduced in [Min91], illustrating it by showing how secure and efficient message passing can be established by means of a law. In Section 4 we introduce additional examples of very simple laws that establish the following regimes: capability-based control over message passing, multiple tuple spaces, and a regime that allows one to lock collections of tuples by unique keys. A much longer example is presented in Section 5, where we introduce a law that ensures the confidentiality of servers with respect to information they get from their clients.

2 Linda and Its Weaknesses

The Linda model [Gel85] provides communication and synchronization between parallel processes by controlled access to a shared data structure called *tuple space*. A tuple space is a collection (bag) of *tuples*, each of which is a linear list of typed values called *fields*.

We present here a slightly simplified description of this model, with some syntactic modifications. We represent a field of a tuple by the term `t(v)`, where `t` is a symbol that specifies the *type* of the field, and the possibly empty `v` is a literal that represents its *value*. We do not confine ourselves to the type system of any particular language, but allow arbitrary "types," represented simply by type-symbols. Thus, a tuple `[person,name(jones),age(23)]` may be used to represent a person with the specified name and age. Note that our Linda model does not define the semantics of any of these "types," but such semantics may be established by the law of the system, as we shall see.

Certain Linda operations use tuple-like structures called *templates*. The fields of a template may be either *actual* or *formal*. An actual field has a specific literal as its value, just as in a regular tuple. A formal field, on the other hand, is a place-holder: It has a type but no value. Instead, a formal field has associated with it an unbound variable, which may either be named by a capitalized symbol or "anonymous," denoted by the symbol `_`. We say that a tem-

[1] We will be talking here about securing communication against *programming errors* and not against malicious attacks. The treatment of malicious attacks requires control over the creation and update of the law, as well as the complete control over the platform on which a system like ours runs; both of these issues are beyond the scope of this paper.

plate *matches* a tuple if all corresponding fields of the two match, in the following sense: An *actual* field in the template matches only an identical field in the tuple, and a *formal* field in the template matches *any* field with the same type. For example, the template `[person,name(jones),age(A)])` would match the tuple `[person,name(jones),age(34)])`. The matching of a formal field binds the value matched with the variable associated with the field. Thus, the match above the value 34 binds to variable `A`. It is through such binding that processes gets information from the tuple space.

Every process in the system can operate on the shared tuple space by means of the following three operations[2]:

`out(T)` inserts the given tuple `T` into the tuple space.

`rd(T)` Locates in the tuple space a tuple `T'` that matches the template `T` and binds values to any formal fields in `T`. If no matching tuple can be found the process waits for one to appear. If multiple possible matches exist, one is chosen arbitrarily.

`in(T)` Locates and binds a matching tuple just like `rd`, but also removes the tuple from tuple space. The match and removal are an atomic operation on the tuple space.

For a detailed discussion of the advantages of this model and its use for communication and synchronization the reader is referred to [Gel85, CG86, WL88, CG88]. Here we we will discuss the major weakness of this communication model.

2.1 Weaknesses of Conventional Linda Communication

To illustrate the weaknesses of conventional Linda we will demonstrate that this model cannot support even a simple form of pairwise communication (i.e., message passing) which is secure from eavesdropping and from interference, and which is reasonably efficient.

It might seem that message passing can be easily accomplished[3] by adopting the convention that a tuple

```
[msg,from(s),to(t),m]
```

represents a message `m` in transit from process `s` to process `t`. In other words, the convention is that only process `s` **out**'s such a "message-tuple," and that only process `t` **in**'s it.

Unfortunately, however, such realization of message passing would be very unsafe, and potentially quite inefficient. It would be *unsafe* because it relies on a *voluntary convention*, which can be easily violated by every process. Indeed, under Linda every process *can* read, and even remove, message-tuples that, by our

[2]Some dialects of Linda have additional operations, which are not central to the purpose of this paper but can be easily included in our model if desired.

[3]At least, if we do not worry about preserving the order of messages exchanged between pairs of processes.

convention, are intended for a given process `t`; and every process *can* **out** message-tuples that, by our convention, appear to have been sent by some process `s`, thus effectively forging a message from `s`. Message passing programs are written under the assumption that messages come from their apparent senders and are delivered to their intended recipients. Without this assumption, we can say little about what the code will do. A Linda program simulating message passing insecurely could easily fail because of bugs in unrelated modules that cause them to interfere with message-tuple's.

Such message passing would also be *inefficient* because it relies on an *implicit* convention, unknown to the Linda implementation. While tuple space is *logically* global, its *physical* implementation may be distributed. It is the implementation that is responsible for the placement of tuples in the (physical) tuple space and later for their location and retrieval. The implementation has no way of knowing that the best place in the tuple space for a message-tuple above is as close as possible to the process `t` for which it is intended.

It may be possible to optimize the placement of such message-tuples by analyzing the code that drive all processes of the system and concluding that only process `t` actually reads this tuple. An implementation can also combine global compile-time analysis with run-time monitoring [Lei89, Bjo92]. But for large, evolving systems such *global optimization* is quite impractical, as has been pointed out by Kahn and Miller [KM89].

Similar criticism has been leveled against Linda by other writers, who proposed various partial solutions. Kahn and Miller [KM89] proposed to enhance the safety of communication in Linda by means of the concept of *multiple tuple spaces.* And Pinakis [Pin92], who was worried specifically about safe message passing, proposed to use encryption technique to support such communication in Linda. Our solution by means of law-governed architecture, is more radical, and far more general then these. In the particular case of message passing we would fortify the above *convention* by an explicit and enforced *law*, which does not let message-tuples be mistreated or forged, and which thus provides some *a priori* knowledge about the system, that may be useful for optimization. And, of course, we are not limited to laws of message passing, as we shall see after introducing our concept of Law-Governed Linda, next.

3 Law-Governed Linda

In this section we define a model for law-governed architecture for Linda communication, or simply Law-Governed Linda (LGL), which is an adaptation of the previously published model of law-governed architectures for message-passing systems. This paper is self contained, and does not assume any familiarity with the previous model.

A distributed law-governed system is a 5-tuple

$$\langle \mathcal{C}, \mathcal{P}, \mathcal{CS}, \mathcal{L}, \mathcal{E} \rangle$$

where $\mathcal{C}$ is a *communication medium* — in this paper, a Linda tuple space; $\mathcal{P}$ is a set of sequential *processes* that interact with each other via $\mathcal{C}$; $\mathcal{CS}$ is a set of what we call *control states*, one associated with each process; $\mathcal{L}$ is the *law* of the system, which governs the interactions of the various processes in $\mathcal{P}$ with the communication medium $\mathcal{C}$, and thus, with each other; and $\mathcal{E}$ is a mechanism that enforces the law.

The *law* $\mathcal{L}$ deals with certain events, called the *controlled events*, that occur at the boundary between a process and the communication medium $\mathcal{C}$. An example of a controlled event is an attempt of a process to **out** a tuple. The law has no control over the *occurrence* of controlled events,[4] but it prescribes the *effects* that any of them should have. This prescription, which is called the *ruling* of the law, is a sequence of *primitive operations* which are to be carried out in response to the event in question. For example, the ruling of the law for an `out(t)` operation by a process `p` might be to complete this operation by actually storing tuple `t` in the tuple space. Alternatively, the ruling might be to **out** some different tuple instead of `t`, or to block the operation altogether; moreover, the ruling might update the control state of `p`.

Note that although the law itself is global, in the sense that all processes are made to obey the same law, the ruling of the law for a given event may depend on the control state of the process in which the event happened. In other words, the global law of a system may be written to distinguish among processes on the basis of their control states.

The law is *enforced* by means of the following *distributed enforcement mechanism* $\mathcal{E}$: There is a *controller* associated with every process in the system. It acts as a mediator between the process and the communication medium. All the controllers in the system have identical copies of the law $\mathcal{L}$; this is what makes the law global. Each controller also maintains the local control state of its own process, which generally differs from the control state of other processes. We refer to the combination of a process and its controller as an *object*.

A controller operates essentially as follows: It intercepts all controlled events that occur at its own process; it computes the ruling of the law for each such event, using its own local copy of the law and the local control state; and it carries out the ruling. Thus, the global law is interpreted and enforced locally at each process by its own controller.

It is important to understand that the above is only an abstract description of the architecture. The actual implementation of LGL may be more efficient, and more complex, than the above it two ways. First, in some cases it may be possible to enforce the law more efficiently by compile-time analysis and manipulation of the programs that drive various processes. Second, although in principle the controller of all processes must have the same law, in practice each controller may have to maintain only part of the law, by excluding rules that can be proved to be irrelevant to this process. But these aspects of LGL are beyond the scope of this paper.

A final observation is in order before we turn to the detailed description of this architecture: We employ the Prolog language for the formulation of the law.

[4] This is not quite the case for laws that incorporate obligations, as discussed in [ML85, LM].

Consequently, the reader is expected to have a passing familiarity with Prolog and with its Edinborough syntax [CM81]. However, this model is in no way limited to systems whose processes are programmed in Prolog, and even the choice of Prolog for the formulation of laws is quite incidental.

3.1 The Controlled Events

Recall that the events that are subject to the law of a system are called *controlled events*. Each of these events occures at the boundary between one of the processes in $\mathcal{P}$, which is called the *home-process* of this event, and the communication medium $\mathcal{C}$.

We distinguish between several classes of controlled events:

Invocation events: These events occur when the home-process invokes one of the Linda-operations: **out**, **rd** or **in**. Such an event should be viewed only as an *attempt* by the home-process to carry out the respective operation. The effect of such an invocation on the tuple space and on the control state of the home-process is determined by the law.

Selection events: These events occur when the template of a Linda operation **in** or **rd** invoked by the home-process is matched with some tuple in the tuple space. A copy of this tuple is presented to the home-controller, but it is not necessarily returned to the home-process itself, nor is the matched tuple necessarily removed from the tuple space (in the case of a preceding **in** operation); such effects are subject to the law.

Asynchronous events: The two previous types of events occur as a consequence of some action by the home-process, and in synchrony with it. But it is sometimes necessary to control events that occur asynchronously with respect to the home-process. These include various kinds of interrupts that may happen around a process, and the event of an *obligation coming due*, which is used to support the concept of *enforceable obligations* presented in [ML85, LM]. Asynchronous events are beyond the scope of this paper and will not be discussed further.

3.2 The Control State of a Process

As has already been pointed out, the control state associated with each process is what enables the global law of the system to make distinctions among different processes. Structurally, the control state is a bag of Prolog-like *terms*, also called the *attributes* of the process. A term is recursive data-structure which has the form `f(arguments)`, where `f` is a literal symbol, and its zero or more arguments are terms. (A singleton term, i.e., a term without any arguments, must be a literal constant — no Prolog-like variables are allowed in the control state.) For example, the term `level(3)` might be used to signify that the process in question belongs to the third level of some kind of layered system. Generally these terms have no built-in semantics in our model; such semantics are established by the law. However,

several distinguished attributes do have predefined meanings and behavior in our model. These include the following attributes:

`self(i)`, where `i` is a unique identifier of the process.

`clock(t)`, where `t` is the current local time at this process. (We make no assumptions about the relationship between clocks at different objects).

`op(t)`, where `t` is the argument of the latest Linda operation invoked by the process in question. `t` is a tuple if the last operation was an **out**, a template if the last operation was an **in** or **rd**.

In the particular but very common case of a unary term, such as `clock(t)`, we refer to the argument `t` as the *value* of the *attribute* `clock`.

The control state for an object is maintained by the object's controller, and is not directly accessible to the process.

3.3 The Primitive Operations

The primitive operations are the operations that can be included in a ruling of the law, to be invoked by the controller in response to the occurance of a given event. Each such operation is defined in the context of the home-object of this event. We distinguish among three classes of such operations, described in the next three sections.

3.3.1 Operations Effecting Interactions Between Home-Object and Tuple Space

`complete` This operation can be included only in the ruling for an invocation event. The effect of `complete` is to actually carry out the operation being invoked. If the event in question was `out(t)`, then `complete` will actually **out** the tuple `t`. If it was `in(t)` or `rd(t)`, a request to match template `t` is forwarded to the tuple space.

`complete(arg')` This operation is like `complete`, except that the original argument of the operation in question is replaced with `arg'`.

`return` This operation can be included only in the ruling for a selection event, i.e., when a tuple from the tuple space has been matched with the template as a result of a preceding **in** or **rd** event which was completed. The tuple is *delivered* to the home-process: If this underlying event specified template `t`, variables in the matched tuple are bound to actuals from `t'`; in the case of **in** the matched tuple is removed from tuple space; and the process continues its execution.

`return(t')` This operation is like `return`, except that the tuple `t'` is delivered to the home-process in place of the matched tuple `t`. (For **in**, it is the matched tuple is still removed from the tuple space.)

`out(T)` This is the conventional **out** operation of Linda, which, when included in the ruling of the law, is carried out without any further interpretation by the law.

`remove` This operation removes the home-process from the system.

In the case of a selection event, note that the process cannot proceed unless the law's ruling includes a `return` primitive operation. Further, if the ruling `return`'s a tuple other than the one matched in tuple space, it must provide one that matches the template of the underlying event. Returning a non-matching tuple constitutes a serious error in the law itself.

We do not allow the inclusion of **rd** and **in** operations in the ruling of the law because we do not want the controller to stall.

3.3.2 Operations on the Control State

`+t` This operation adds the term `t` to the control state of the home-object.

`-t` This operation removes from the control state a single term `t'` which unifies (in the Prolog sense) with `t`. If `t` unifies with no term in the control state, then this operation has no effect; if it can unify with several terms, then one of them is arbitrarily selected for removal.

`(t1<-t2)` This operation performs `-t1` followed by `+t2`. It is a shorthand used only for convenience.

3.3.3 An Operation that Signals an Error

`error(d)` This operation signals an error condition, described in the *diagnostics* `d`. We do not specify here what happens to the object in which such an error is signaled.

3.4 The Law

As has been explained, the law prescribes the effect that any possible controlled-event should have on the system. It does so in the form of a ruling — a (possibly empty) sequence of primitive operations which are to be carried out by the controller of the object where the event occurred.

The law is defined by means of a somewhat restricted (and usually very simple) Prolog program[5] $\mathcal{L}$ which, when presented with a goal E, representing a controlled event, produces the ruling as a list of primitive operations.

The implementation of the controller can be pictured as follows. The controller first initizializes an auxiliary variable `R`, to contain the empty list. Then it presents E

[5]In particular, certain types of Prolog goals, such as `assert`, `retract` and `call`, are not allowed in a law.

to $\mathcal{L}$ for evaluation.[6] During the evaluation, the law may evaluate one or more *do-terms*, which have the form **do**(p), where p is a sequence of one or more terms each of which represents a primitive operations. The evaluation of do-term **do**(p) simply appends p to the ruling list R. After the evaluation of the ruling is complete, the controller proceeds to carry it out, by executing the sequence of primitive operations in R.

Just before starting to evaluate the ruling for a given event the controller instantiates several Prolog variables, which are accessible to the law:

CS: The control state itself, represented as a list of terms (in unspecified order).

Clock:
The value of the attribute clock in the control state.

Self:
The value of the attribute self in the control state.

The latter two of these are really redundant, as they can be retrieved from CS, but they are provided directly for convenience.

The variable CS is usually examined by means of the infix operator @. When invoked by t@l, where t is any Prolog term and l is any Prolog list, this operator attempts to unify t with each element of l in turn. For example, if the object in question has the attribute level(3) and L is unbound then the goal level(L)@CS would bind the value 3 to L, thus extracting the "level" of the object from its control state. (Note that although the most common use of this operator is to search the control state CS, it can be used to search other lists as well.)

3.4.1 An Example: Secure Message Passing

Our first example is the law in Figure 1, which enforces the message-passing convention introduced in Section 2.1, thus making message passing secure and optimizeable.

Specifically, this law makes any message-tuple of the form

[msg,from(s),to(t),m]

into a carrier of a message m from process s to process t by allowing such a tuple to be **out**'ed *only* by s, and to be **in**'ed *only* by t. Besides such message-tuples this law also provides for conventional (unrestricted) Linda-like treatment of all tuples whose first field is not "msg."

To understand this law, first consider a process x which performs the operation

out([msg,from(x),to(y),hello]),

[6] We assume this evaluation always succeeds. This can be ensured by appending appropriate "catch-all" rules to the law, and in fact for the purposes of this paper, we assume such rules produce an error ruling, so that events not specifically dealt with by the law are treated as errors.

giving rise to the corresponding invocation event. When this event is presented as a goal to this law, it matches the head of Rule $R1$, causing the execution of its body, which succeeds, producing a ruling with the single operation: `complete`. The execution of this ruling completes the originating **out**.

While the above **out** operation by `x` will be completed, `x` will not be able to **out** the following tuple:

`[msg,from(z),to(y),hello]`.

This is because the corresponding event will not match the head of Rule $R1$: The variable `Self` is pre-bound to the identification of the home-object — `x` in this case — and will not unify with `z`. Since no other rule in this law would satisfy this event, the operation will be rejected.[7]

```
R1. out([msg,from(Self),to(_)| _ ]) :- do(complete).

R2. in([msg, from(_), to(Self)| _ ])
       :- do(complete) :: do(return).

R3. out([X| _ ]) :- not(X=msg), do(complete).

R4. in/rd([X| _ ])
       :- not(X=msg), do(complete) :: do(return).
```

Figure 1: Message Passing

It follows then that Rule $R1$ allows each process to **out** message-tuples, but only with its own identification in the `from` term. Thus, it is not possible under this law to forge a message; that is, to send a message (**out** a message-tuple) that appears to have come from somebody else.

We turn now to Rule $R2$. This rule deals with the two *coupled events* caused by an **in** operation attempted by a process: (a) the invocation events, and (b) the resulting selection event, in case the invocation event is completed and a matching tuple is found. These two coupled events are handled by the two parts of the body of Rule $R2$ separated by the `::` symbol, as explained in detail below.

Suppose that a process `y` perform the operation

`in([msg,from(x),to(y),Text])`

giving rise to the corresponding invocation event. When this event is presented as a goal to our law it will unify with the head of Rule $R2$ (because the variable `Self` is bound to `y` for an event that happens at object `y`, and, of course, the anonymous variable "`_`" will unify with anything). As a result, the first part of the body of this rule, up to the "`::`" symbol, is invoked. The resulting ruling in this case is, the single operation `complete`, which is carried out by the controller initiating a search

[7] As noted in the footnote on page 9, we generally assume that there will actually be a catch-all rule that will add an `error` primitive to the ruling in a case such as this. If not, the result is an empty ruling, the tuple is discarded, and the process continues.

of the tuple in question. During this search, further evaluation of the selected rule is suspended.

Suppose the search eventually locates the matching tuple

```
[msg,from(x),to(y),hello] .
```

At this point, a selection event occurs. It is resolved by resuming execution of the rule that was selected for the original invocation event (Rule $R2$), in its second part (the part on the right hand side of the "`::`" symbol) The resulting ruling for the selection event would be the primitive operation `return`, causing the tuple to be removed from the tuple space and to be bound to the template of the **in** operation, and thus delivered to the home-process.

Any attempt by a process to **in** a message-tuple which is not intended for it could not select any rule of this law.[8] Similarly, any attempt to **rd** any message-tuple cannot select any rule of this law.

Next, Rules $R3$ and $R4$ provide for standard Linda treatment for all tuples that do not start with the symbol "`msg`." By Rule $R3$, the ruling for any **out** operation with a tuple whose first component in not "`msg`" is `complete`, which means that such tuples can be freely **out**'ed by anybody.

Rule $R4$ employs a syntactic convention that needs to be explained. The symbol `in/rd` in the head of this rule simply means that there are two rules here, one with `in` and one with `rd`, but which are otherwise identical. The effect of these particular rules is that the **in** and **rd** of tuples other than message-tuples will be completed, and the tuples retrieved as a result of these operations will be returned to the process.

It is easy to see that this law establishes secure message passing, without losing any of the flexibility of Linda, which is still available under this law for most kinds of tuples. Moreover, this kind of message passing can be made quite efficient by means of a general purpose optimizer which can draw upon the *law*, which is global and rarely changes, rather than the text of many different programs, which might change often.

Broadly speaking, this optimizer would work as follows: When given an `out(t)` operation to perform, the optimizer would use the law in an attempt to find the identifications of the processes that are *able to* retrieve this tuple. If there is just one such process `z`, then the optimizer could cause tuple `t` to be transferred directly to `z` as a regular message. In our example, it is easy to prove that only `y` can retrieve the tuple produced by

```
out([msg,from(x),to(y),hello])
```

Hence, the potentially expensive Linda communication could, under this law, be transformed into simple message passing automatically, *even without any knowledge about the code of any process in the system.*

[8] Again, we usually assume the law produces an `error` primitive in this case. If the ruling is simply empty, the process will be blocked forever unless we extend the definition of Linda to allow for "suppressed" operations.

4 Examples

4.1 Capability-Based Control Over Message Passing

Elaborating on our previous example, we now consider the law defined in Figure 2, which establishes a *capability-based control* [DG72] over the exchange of messages between processes. As mentioned earlier, Pinakis [Pin92] proposed a modification of Linda that uses encryption to support capabilities. Here we show how, under Law-Governed Linda, a similar discipline can be established simply by a law.

We represent the possession by process x of a capability for y by means of a term cap(y) in the control state of x, and we will say in this case that y is an *acquaintance* of x. We assume some initial distribution of such capabilities among the processes of a system.

Initially:

Some initial distribution of cap*-terms is assumed.*

```
R1. out([msg,from(Self),to(T)| _ ]) :- cap(T)@CS, do(complete).

R2. in([msg, from(_), to(Self)| _ ])
        :- do(complete) :: do(return).

R3. out([cap(Self),for(T)]) :- cap(T)@CS, do(complete).

R4. out([cap(Z),for(T)]) :- cap(Z)@CS, cap(T)@CS, do(complete).

R5. in([cap(Z), for(Self)])
        :- do(complete) :: do(+cap(Z), return).
```

Figure 2: Capability-Based Message Passing

The semantics of cap(y) terms as capabilities for y is established by Rule $R1$, which governs the **out**'ing of message-tuples. This rule is similar to the corresponding rule in the previous law, except that it conditions the **out**'ing of a message-tuple addressed to a given process on the possession, by the home-process, of a capability for the target process. This condition is checked by the expression cap(T)@CS, which succeeds if the term cap(T) can be unified with any term in the control state of the home-object. By Rule $R2$, which is identical to the corresponding rule of the previous law, every process can pick-up message-tuples addressed to it.

The rest of this law establishes the following principle concerning the *distribution of capabilities*: A process can give to its acquaintances a copy of any capability it has, as well as a capability for itself. Specifically, under this law, the "giving" to y of a capability for z is carried out by **out**'ing the tuple [cap(z),for(y)] into the tuple space. The **out**'ing of such *capability-tuples* is governed by Rule $R3$, which allows a process to give capabilities for itself, and by Rule $R4$, which allows it to gives capabilities for its acquaintances. In both cases the capability thus given must be addressed to an acquaintance of the donor.

Finally, by Rule $R5$, once a capability-tuple is in the tuple space, it can be **in**'ed by the process y which is specified in the tuple's for field. Also by this rule, a process that **in**'s a capability tuple is granted the capability specified in it.

4.2 Establishing Multiple Tuple Spaces

The concept of *multiple tuple spaces* has been frequently proposed [KM89] as a a means for enhancing the usefulness of Linda. Here we show how such a facility can be established under LGL by means of a very simple law. Specifically, under the law of Figure 3, the tuple space is effectively partitioned into disjoint named subspaces, where a subspace s, consisting of all tuples of the form [subspace(s),...], is accessible to every process that has in its control-space the term hasAccess(s), which thus represents permission to access the subspace. This is done simply as

$\mathcal{I}$nitially:

Every object can have any number of hasAccess(s) *terms in its control state, identifying the subspaces it can access.*

```
R1. out([subspace(S)| _ ] :- hasAccess(S)@CS, do(complete).

R2. in/rd([subspace(S)| _ ]
        :- actual(S), hasAccess(S)@CS, do(complete)
        :: do(return).
```

Figure 3: Multiple Tuple Spaces

follows: Rule $R1$ of this law allows every process to **out** into any subspace s if there is the term hasAccess(s) in its control state. Rule $R2$ provides the counterpart ability to **rd** and **in** from any subspace which it has the permission to access.

Rule $R2$ uses the predicate actual supplied by the controller, which tests a field in a template to determine if, as specified in the code within the process, was an actual, not formal, field. Its use here requires the name of the subspace be explicitly specified; a process cannot attempt to query multiple subspaces at once. This restriction can be removed by replacing Rule $R2$ is replaced with Rule $R2'$: Rule $R2'$ imposes no restrictions on the ability of a process to invoke **rd** or **in**

```
R2'. in/rd([subspace(S)| _ ]
        :- do(complete) :: hasAccess(S)@CS, do(return).
```

Figure 4: Multiple-Tuple Spaces

operations, but when a tuple arrives as a result of such an operation, it will be returned *only* if it is in a subspace that this process is allowed to access. This rule enables a process to ask for a tuple without knowing in which subspace it resides, provided the tuple resides in one of the subspaces it is allowed to access. But there may be a price to be paid for this flexibility — by some other processes — because the very invocation of an **in** or a **rd** operation on a tuple, even if it is

ultimately blocked by the law, may affect the time that it takes some other process to retrieve the same tuple. Consequently, one may want to prevent processes from even invoking retrieval operations on subspaces they are not allowed to see, as it has been done by Rule $R2$.

Note also that law the law of Figure 3 can be easily combined with the law in Figure 1, producing a law that allows capability controlled message passing, along with multiple tuple spaces. This does not imply that laws are always additive; they are not. But it is often possible to design law fragments that can be combined with little or no change.

Before we leave this particular example we point out the there are many useful variation of the notion of multiple tuple spaces which can be supported under LGL. One particularly interesting variation will be discussed in Section 5.

4.3 Locks and Exclusive Keys

We now consider a law, defined in Figure 5, that provides for the locking of arbitrary collections of tuples by means of keys. Each key will be unique and unforgeable, and it will be possible to transfer keys, but not copy them. In other words, under this law only a process that holds a key `k` can read and write tuples locked by `k`; and only one process can have a given key at a given moment in time (which guarantees mutual exclusion of access to the tuples locked with a given key). This law also provides for the management of keys, and it allows free storage and retrieval of unlocked tuples.

Unlocked tuples under this law are those whose first term is "`unlocked`." The unrestricted storage and retrieval of such tuples is provided for by Rules $R1$ and $R2$. Tuples that are locked with a key `k` (whose internal structure is discussed below) are those whose first term is `locked(k)`. By Rules $R3$ and $R4$, such tuples can be stored and retrieved only by a process that has the term `key(k)` in its control state.

4.3.1 Key Management

We are assuming that in the initial state of the system no process has any keys, and that no keys exist in the tuple space. The law provides for the means for the creation of new unique keys and for their subsequent exchange.

Specifically, Rule $R5$ provides for the creation of new keys. The internal representation of a key is a pair containing the identification of the process at which the key was created, and the time when the creation occurred. This pair provides a value that is unique throughout the system. The key is added to the local control state, and is also returned to the process so that it can begin using it. Note that the ruling resulting from this rule never contains a `complete` primitive, so tuple space is never searched: The **in** operation receives a value computed directly by the law. As a result, the rule has no "`::`" symbol or second part. (The process does not care, and in fact cannot determine, whether the value it receives came from tuple space or directly from the law.)

Finally, once a key is created it can be exchanged between processes, as follows:

$\mathcal{I}$nitially:

No process has a term of the form `key(K)` *in its state, and there is no tuple of the form* `[key(K)]` *in the tuple space.*

$\mathcal{R}$1. `out([unlocked| _ ]) :-` **do**`(complete).`

out'*ing unlocked tuples.*

$\mathcal{R}$2. `in/rd([unlocked| _ ]) :-` **do**`(complete) ::` **do**`(return).`

Retrieving unlocked tuples.

$\mathcal{R}$3. `out([locked(K)| _ ]) :- key(K)@CS,` **do**`(complete).`

out'*ing locked tuples.*

$\mathcal{R}$4. `in/rd([locked(K)| _ ])`
`:- key(K)@CS,` **do**`(complete) ::` **do**`(return).`

Retrieving locked tuples

$\mathcal{R}$5. `in([newkey(K)])`
`:- K=[Self,Clock],` **do**`(+key(K),return(newkey(K))).`

Generating a brand new key.

$\mathcal{R}$6. `out([key(K)]) :- key(K)@CS,` **do**`(-key(K),complete).`

Returning a key to the tuple space.

$\mathcal{R}$7. `in([key(K)])`
`:- actual(K),` **do**`(complete) ::` **do**`(+key(K),return).`

Getting an exclusive key K.

Figure 5: Keys

By Rule $R6$ a process can give up any key it possesses, releasing it to the tuple space, simply by **out**'ing it. The key will be removed from the process's control state. On the other hand, by Rule $R7$, any key stored in the tuple space can be acquired by any process by means of an **in** operation. Because Rule $R7$ includes the test `actual(K)`, a process cannot undertake "fishing expeditions": It can only request specific keys.

There are many useful variation of this lock-and-key discipline which are very easy to establish by small variations of this law. In particular:

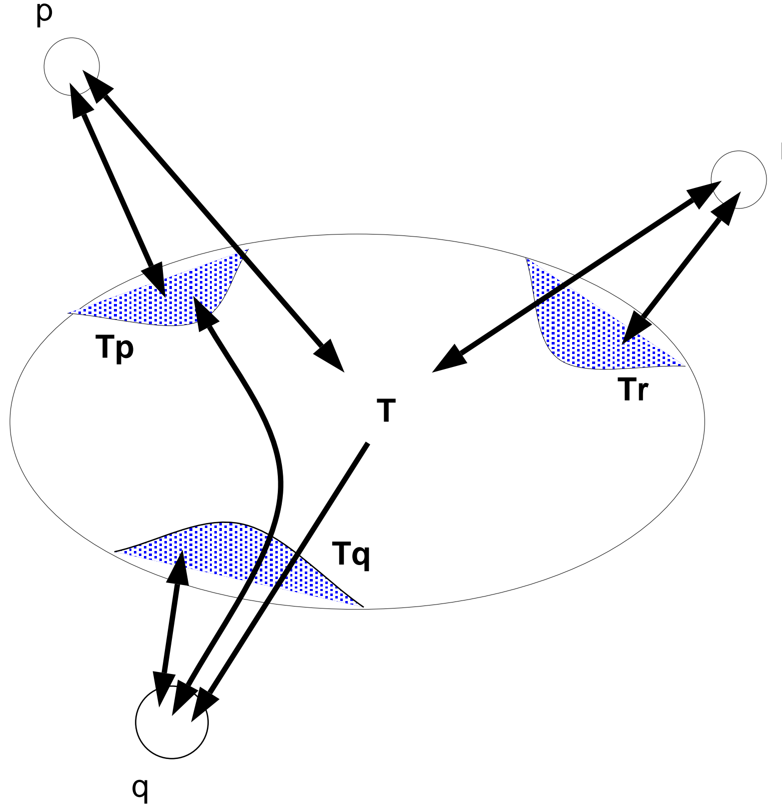
- Only certain processes may be allowed to get certain keys.
- One may make a distinction between the ability to read (**rd**) and the ability to modify (**out**, **in**) locked sets.
- One may require a process to have only a single key at a time.
- To enforce a common method of deadlock prevention, one may organize the various keys into some kind of partial order, and force processes to obey this order in some way.

5 Case Study: The Confidential-Server Problem

In this section we address the following problem: Consider a client C who employs a server S. C provides S with some private information which S needs to perform the desired service. C wishes to be certain that S cannot reveal to anybody else any of the private information it received in confidence from its client.

This is an ancient problem faced by the proverbial king who employed an architect to design his underground treasury. To protect the secrets of his new treasury, the king confined the architect to a cell during the design process, and killed him when he finished the design. An analogous solution for computer systems, where a server might be a program employed to compute the income tax of some client, for example, has been studied as the "confinement problem" [Lam76]. The inhumanity of the ancient king is not a problem here, but the confinement of a programmed server may be technically difficult. It is, of course, always possible to write a specific server so that it reveals no secrets, but how does make sure that *no* server can reveal any secrets? That is, how does one *impose* a confinement regime on *any* process that plays the role of a server?

The law to be introduced below imposes just such a regime, and in a very *open* context, where the client may not know the nature or even the identity of its server. It should be pointed out, however, that our solution is not quite complete, in the following sense: We deal here only with the interaction of processes with the tuple space, ignoring possible communication between individual processes and the outside world, either by means of simple output, or through the very subtle, even if narrow, channels discussed by Lampson [Lam76]. The former kind of channels can, in principle, be blocked by extending the class of controlled events; the later ones generally cannot be.



Process q here is serving p, and process r in unengaged.

Figure 6: Confidential Servers

Informally speaking, we intend to establish the following regime, partially illustrated by Figure 6. In detail, this regime is defined as follows:

1. A process may be in one of the following states: It can be a *server*, a *client* or it may be *unengaged* in any service transaction. Moreover, a client may be served by only one server at a time, and a server may serve only one client at a time.

2. The tuple space is partitioned into a *public subspace* T, and, for every process p, a *private subspace* T_p.

3. The private subspace T_p is fully accessible to its owner p, as well as to any process that serves it. This subspace can thus be used for communication between a client and its server.

4. The public subspace T is fully accessible to all non-servers. However, once a process takes on the role of a server, for any client, it loses the ability to

update T, lest it store in T, and thus make public, information it got from the private space of its client. But even servers can read from T.

5. Upon the termination of a service transaction the server is removed from the system, very much like the architect in the old legend.

The tuple space may contain some *control tuples* that are used as part of the implementation of the regime, and do not logically belong to either T or any T_p. Their structure and function will be discussed later.

This general regime, and suitable protocols for starting and terminating a service transaction, are established by a law described in Figures 7, 8 and 9, which are discussed in the following three sections.

5.1 Private and Public Subspaces

We represent the *public* part of the tuple space by means of the set of tuples of the form `[public,...]`. The private subspace T_p of process `p` is represented as the set of tuples of the form `[private(p),...]`.

When a process `s` (the server) is serving process `c` (the client), it will have the term `servantOf(c)` in its control state; conversely, `c` will have the term `servedBy(s)` in its control state.

$\mathcal{R}1$. `rd([public| _ ]) :- ` **do**`(complete) :: ` **do**`(return).`

Anybody can **rd** *public tuples.*

$\mathcal{R}2$. `out([public| _ ]) :- not(servantOf(_)@CS), ` **do**`(complete).`

Non-servers can **out** *public tuples.*

$\mathcal{R}3$. `in([public| _ ]) :- not(servantOf(_)@CS), ` **do**`(complete)`
`    :: ` **do**`(return).`

Non-servers can **in** *public tuples.*

$\mathcal{R}4$. `out([private(P)| _ ])`
`    :- (P=Self|servantOf(P)@CS), ` **do**`(complete).`

Only a process `p` *and whoever is serving it can* **out** *into* `p`*'s private space.*

$\mathcal{R}5$. `in/rd([private(P)| _ ])`
`    :- (P=Self|servantOf(P)@CS), ` **do**`(complete)`
`    :: ` **do**`(return).`

Only a process `p` *and whoever is serving it can extract tuples from* `p`*'s private space.*

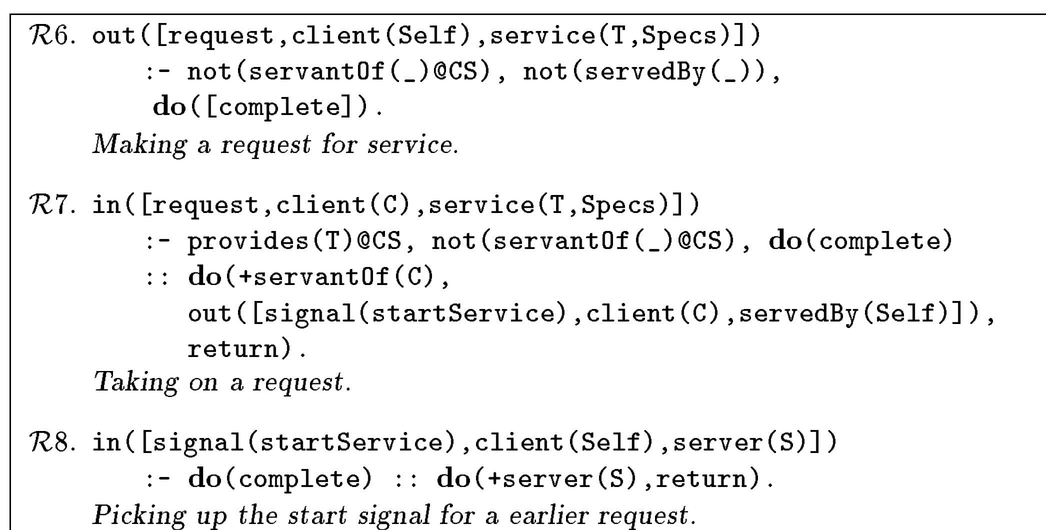
Figure 7: The Law of Confidential Servers — Part I

Now, by Rule $R1$ of Figure 7, all processes can read from the public subspace, and by Rules $R2$ and $R3$ this subspace can be updated only by non-servers. On the other hand, due to Rules $R4$ and $R5$, the private subspace of a process is accessible (for retrieval and for update) only to this process and to a process that serves it.

Note the distinction, in Rules $R1$ and $R3$, between **rd** and **in** operations. A **rd** operation does not modify the public space, so a server can be allowed to invoke it. On the other hand, an **in** operation has a side-effect, and hence could be used to transmit information, so cannot be allowed.

5.2 Establishing a Client-Server Relationship

We assume here that a process is *qualified* to provides services of type `t` when it has an attribute `provides(t)` in its control state. We also assume that the identity of such potential servers is not known *a priori* to their prospective clients. Under these conditions, the rules displayed in Figure 8 establish the following protocol to



$\mathcal{R}6$.
```
out([request,client(Self),service(T,Specs)])
    :- not(servantOf(_)@CS), not(servedBy(_)),
    do([complete]).
```
Making a request for service.

$\mathcal{R}7$.
```
in([request,client(C),service(T,Specs)])
    :- provides(T)@CS, not(servantOf(_)@CS), do(complete)
    :: do(+servantOf(C),
       out([signal(startService),client(C),servedBy(Self)]),
       return).
```
Taking on a request.

$\mathcal{R}8$.
```
in([signal(startService),client(Self),server(S)])
    :- do(complete) :: do(+server(S),return).
```
Picking up the start signal for a earlier request.

Figure 8: The Law of Confidential Servers — Part II

institute a service relationship between two processes:

1. When a process `c` (the prospective client) wants a certain service to be performed it asks for it by **out**'ing a tuple

   `[request,client(c),service(t,specs)]`,

   where in the term `service(t,specs)` `t` specifies the type of service being requested, and `specs` provides additional details about it. Rule $R6$ allows an *unengaged* process to **out** such a tuple, and ensures that the request identifies the client correctly.

2. By Rule $R7$, a service request can be retrieved (via **in**) by any unengaged qualified server `s`, with the following consequences:

   (a) Attribute `servantOf(c)` is added to the state of `s`, making `s` the server of `c`. (As we shall see, only `c` will be able to release `s` from this servitude.)

   (b) A tuple `out([signal(startService),client(c),server(s)])` is **out**'ed into the tuple space, to be picked up as a start signal by the client.

3. By Rule $R8$, the client `c` can pick up a start signal generated in response to its request. Doing so causes the term `servedBy(s)` to be added to the control state of `c`.

At this point the client and its server know of each other, and due to the rules displayed in Figure 7, they can now communicate and share information via the private space of the client.

5.3 Terminating the Client-Server Relationship

The rules displayed in Figure 9 establish a protocol for terminating the connection between a client and its server, as follows:

1. The termination of a service starts by a client `c` **out**'ing a tuple

   `[signal(satisfied),server(s),client(c)],`

   which is to be eventually picked up the server `s` and interpreted by it as a termination signal. The sending of this signal is governed by Rule $R9$ which ensures that the server and the client are identified correctly. We note that *sending* this signal cannot, by itself, terminate the relationship between client and server, as the server may not be aware that the client is satisfied and may be still manipulating the client's private subspace.

2. By Rule $R10$ a satisfied-signal can be **in**'ed only be a process identified by the signal as the server, with the following consequences: (a) an informative "record" tuple is **out**'ed, registering the conclusion of a service. This tuple might be used for billing or, more generally, to provide evidence of the successfully completed client-server relationship; (b) a "done" signal of the form

   `[signal(done),client(c), server(s)]`

   is **out**'ed, to be later picked up by the client `c`; and (c) the server is removed from the system, very much like the architect in the old legend.

3. Finally, by Rule $R11$ a "done" signal can be retrieved only by a process identified in the signal as the client, causing the term `servedBy(s)` to be removed from the client's control state. At this point the former client can be sure that its former server has already been removed, and thus can no longer manipulate the client's private subspace. The receipt of this signal marks the real termination of the service, freeing the former client to ask for a new service, or to become a server himself.

```
R9. out([signal(satisfied),server(S),client(Self)])
        :- servedBy(S)@CS, do(complete).
    Sending a satisfied-signal.

R10.
    in([signal(satisfied),server(Self),client(C)])
        :- servantOf(C)@CS, do(complete)
        :: do(out([record, server(Self),client(C),at(Clock)]),
              out([signal(done),client(C),server(Self)]),
              remove).
    Ending servitude by picking up a "satisfied" signal and dying.

R11.
    in([signal(done),client(Self),server(S)])
        :- servedBy(S)@CS, do(complete)
        :: do(-servedBy(S),return).
    A client picking up a "done" signal from his server.
```

Figure 9: The Law of Confidential Servers — Part III

6 Conclusion

Our objective in this paper has been to remedy two serious deficiencies of the original Linda model: its inherent lack of safety, and its inefficiency. We have amply demonstrated that a law-governed architecture rectifies the safety problem in Linda, but its real effect on the efficiency of Linda communication is less clear. In principle, the law of a system under Law-Governed Linda does provide some *a priori* knowledge about that system which can be helpful for optimization. We have also demonstrated that such an optimization can actually be carried out in some cases, such as in our law of message passing. But the degree to which this potential can be realized in a broad range of applications is still an open question.

References

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed programming.* Prentice Hall, 1990.

[Bjo92] Robert Bjornson. Linda on distributed memory multiprocessors. Technical Report RR-931, Yale University Department of Computer Science, November 1992. Also a 1993 Yale University PhD thesis.

[CG86] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with Linda. *ACM SIGPLAN Notices*, 23(9), September 1988. Proceedings of ACM/SIGPLAN PPEALS 1988.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[DG72] P.J. Denning and G. S. Graham. Protection - principles and practice. In *AFIPS 1972 Spring Joint Computer Conf.*, pages 417–429. AFIPS, 1972.

[Gel85] David H. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.

[KM89] K.M. Kahn and M.S. Miller. A letter about the article 'Linda in Context' by Carrierro and Gelernter. *Communications of the ACM*, 32(10):1253–1255, Oct. 1989.

[Lam76] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct. 1976.

[Lei89] Jerrold S. Leichter. Shared tuple memories, shared memories, buses and LAN's — Linda implementations across the spectrum of connectivity. Technical Report TR-714, Yale University Department of Computer Science, July 1989. Also a 1989 Yale University PhD thesis.

[LM] Jerrold Leichter and Naftaly H. Minsky. Obligations in law governed distributed systems. In preparation.

[Min91] N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.

[ML85] N.H. Minsky and A. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th International Conference on Software Engineering*, pages 92–102, August 1985.

[Pin92] J. Pinakis. Providing directed communication in linda. In *Proceedings of the 15th Australian Computer Science Conf.*, pages 731–743, 1992.

[WL88] Robert A. Whiteside and Jerrold S. Leichter. Using Linda for supercomputing on a local area network. In *Proceedings, Supercomputing '88*, November 1988. Also Yale University Department of Computer Science Technical Report TR-638 and Sandia National Laboratories Report SAND88-8818, both June 1988.