Treaties: Behaviour-Controlling Capabilities

Yining Zhao
Department of Computer Science
The University of York
York, UK
hopezhao@cs.york.ac.uk

Alan Wood
Department of Computer Science
The University of York
York, UK
wood@cs.york.ac.uk

***Abstract*—Conventional approaches to access-control, such as ACLs, do not scale well enough for distributed systems. Capabilities on the other hand offer scalability and adaptability advantages in large-scale distributed environments due to their being held and managed by the system's users/agents rather than by the middleware. However the structure of capabilities is only able to provide simple sequence-independent rights-based control, which precludes most forms of composition of actions. If required, such fine-structure behavioural control would be the responsibility of the middleware, thus negating some of a capability system's advantage. Therefore we propose the concept of Treaties: a generalization of capabilities whereby sequence of actions can be specified, and enforced. Treaties can be safely modified and combined by, and passed between, system agents with minimal middleware involvement, enabling rich and complex ensembles of cooperating agents to be formed. This paper outlines the motivation and structure of treaties and their operations, covering the design space focusing on the new implementation issues raised, and ending with preliminary implementation results.**

Keywords-Behaviour Control; Access Control; Capabilities; Distributed Computing

I. INTRODUCTION

Distributed systems and cloud services are important areas of research that are becoming ubiquitous components in everyone's life. It is essential to provide these applications with the necessary supports, such as mechanisms that ensure functionality, confidentiality, integrity, security, etc. Most modern services are provided subject to some required protocols which can be expressed as allowable sequences of actions. For example, to use an on-line banking system, the user would need to first login, and then choose to do a number of actions such as making payments or transferring money between accounts, and then logout. Clearly these services involve confidential data that should be protected and hence access (and other) control mechanisms are essential.

In order that such services can be provided, the system must provide mechanisms to support:

- confidentiality and security — only authorized agents can access certain resources.
- agent-level construction of behavioural specifications.
- flexibility and scalability, so that they are able to operate in open distributed environments.

Most conventional access control mechanisms are centralized, since agents' identities (and other properties) need to be managed centrally. Although this option makes certain useful properties easy to implement, such as strong authentication, these mechanisms are quite static and scale badly.

Capabilities, originally introduced by Dennis and Van Horn [1], to support memory management for multiprogrammed computation, are a well-known dynamic approach to access control. They differ from the widely used Access Control List (ACL) approach, as ACLs are centralized controls with a fixed user population, whereas capabilities are more flexible and can be easily distributed. Capabilities can be visualized as unforgeable tickets, initially created by a system-level kernel, but which can be propagated by agents. When an agent wishes to access (or otherwise operated on) a resource a suitable capability is provided by the agent to the kernel along with the request for the action. So it can be seen that, abstractly, a capability consists of two parts: a reference to an object/resource, and a set of rights indicating the types of operations that the holder of this capability can perform on the referred object. In such an approach, kernels only need to check the validity of capabilities, not the identity of the presenting agent… if you have the capability, it doesn't matter who you are. Hence capabilities are particularly suitable for distributed environments.

However, capabilities have some weaknesses which are hard to overcome. For example, it is widely believed that capabilities cannot enforce confinement [2], which requires that data should only flow from places of a lower security level those of equal or higher security, but not vice versa. Since the propagation of capabilities is an agent-level task, the middleware is unable to exert strong control, and consequently it appears that capabilities might propagate without limit to untrusted agents. As another example, it is difficult in a capability system to revoke permissions that have been previously granted. Since capabilities are bit-strings held within agents' private storage, the middleware cannot delete them. This is called the revocation problem [3, 4] (but see Section II).

In addition to these apparent problems, the structure of capabilities is not rich enough for defining agents' behaviours in distributed systems. For instance, the holder of a capability has unlimited use of any right that the capability provides. Capabilities cannot express cases such as 'only allowed to read

the file three times'. These and other considerations mean that we need to develop new solutions that may better support services which require a finer level of behavioural specification.

This paper introduces the idea of treaties which extends the concept of capabilities, but is more refined in modelling a number of actions for specifications. This is accomplished by replacing the 'set of rights' component of capabilities with behaviour descriptors. These descriptors capture features of behaviour such as sequence, branch and termination. With these it is much easier to build the specifications that service providers require.

In addition to replacing 'rights' by 'behaviours', treaties are first-class objects that can be safely combined using a small set of operations to construct derived specifications. This paper addresses the novel implementation issues that derive from treaties, and discusses the various possible representations of treaties with explanations of different choices. Finally the results of a comparison between time cost of using capabilities and treaties is presented.

II. Related Work

There has been much work aiming at the improvement of the capability approach to overcome the restrictions mentioned above and make it more applicable. A known mechanism that solves the revocation problem is the use of caretaker [5, 6]. When a user passes capabilities to others, he can add 'caretakers' on them. The caretakers refer to some 'gates' set up by the user, and the user can 'open' or 'close' these gates. Only when gates are open, can others use capabilities from the user to access resources. Therefore the user can disable capabilities he previously sent out by shutting down gates, i.e. he revokes these capabilities. Besides, ICAP [3] and Split Cap [4] are claimed to solve confinement problems to some extent.

Capabilities with modifications are also applied in various fields. EROS [7] shows that a capability-based operating system can perform as well as ACL-based systems without any special hardware assists. Establishing private channels in LINDA-like [8] tuple-space systems has also been suggested using mix of ACLs and capabilities [9]. Multicapabilities [10] apply the idea of capabilities into tuple-space systems by modifying the structure of capabilities to refer to multiple unnamed targets of a matching pattern. μKLAIM [11] uses a capability approach to solve the problem of unmatched static control in dynamic environments. Vistas [12] extend the idea of capabilities by introducing combining operations, so that the holder of vistas can construct them to fit different cases. The operation of 'product' in vistas shows some aspects of behavioural specification by constructing a pair of actions that must be performed in sequence. Vistas can also contain references to multiple objects which capabilities do not. The treaty concept is a direct extension of the idea of vistas.

III. The Concept of Treaties

Treaties extend the capabilities and vistas ideas to a new level. Capabilities and vistas only deal with rights. These rights in capabilities are independent, unlimited and unordered. For example, assuming there is a capability containing both read and write rights, the two access rights are not dependent on each other, and the holder of the capability can read or write an unlimited number of times, and can read and write in any order. This is a straight-forward mechanism that is suitable for simple systems. However, as services become more sophisticated, the simple mechanism is no longer satisfactory for complex system requirements. There are many cases that require the access actions to be in certain orders, or a limited number of times. We call these sequences of actions *behaviours*. Clearly capabilities are unable to specify behaviours, except in the most trivial sense. Vistas make one step towards behavioural control, by the product operation, but they are still too simple: there are only pairs of actions.

Hence we introduce the concept of treaties. Treaties are extended from the idea of capabilities. This inheritance defines that treaties adopt many properties from capabilities. They are structures that define the rights to operating on resources. They are not centrally controlled, but are held by user-level agents. In treaty systems a user must present an appropriate treaty to the kernel (middleware) in order to perform actions on the resource that he is accessing. Only kernels can create the initial treaty to a certain resource - users cannot forge any treaties. Treaties shall also inherit the dynamic features of capabilities, and they can be propagated throughout systems. This means that treaties are suitable to be deployed in distributed environments.

The key function that treaties provide is 'behaviour control'. This differs from the 'rights only' mode of capabilities in the sense that a treaty can allow or deny different actions at different times or stages, depending on the current state of the treaty or current stage of the process/behaviour (capabilities are stateless entities, and so cannot control sequences of actions). With treaties sequences of allowed actions are specified, and enforced, in order to form behaviours.

As an example of a behaviour as a sequence of actions from everyday-life, in the morning we sign-in to our email boxes, read new emails, reply to senders and shut-down mail boxes. This is exactly a simple but typical behaviour. At the beginning, one cannot read or compose mails but only log in. After logging in one can read and compose an unlimited number of times, until logging out, after which only logging is allowed. Treaties aim to capture these attributes of behaviours, such as sequencing, branching and terminating.

Abstractly, treaties are access control components that provide specifications of behaviours that can be followed for a number of resources, together with the current state of the evolution of the behaviours. As fundamental requirements, treaties shall:

- refer to resources.
- provide information to a kernel to enable it to allow or deny an action request.
- not be strictly bound to agents, i.e. kernels do not need to check their holders' identity in order to grant or deny actions.

- be able to 'evolve' behaviours according to their specifications, i.e. maintain and update the behavioural state.

Therefore each treaty can be seen as consisting of three components: the reference pointing to the resource (or resources), the behaviour descriptor modelling the behaviours permitted by the treaty, and the current state of the behaviour in this treaty. From this, we can see that that capabilities are special treaties, in which there is only a single behavioural state, and all allowed actions are reflexive transitions on this state.

A. Operations and the Rule of Treaties

Behaviours are more complex than simple rights. However, when a new resource object is created, the kernel provides the creator of this object a 'complete' treaty, which contains all possible action types with unlimited repetitions, i.e. a capability. To transform the complete treaty into a treaty that specifies more refined behaviours, we need a number of operations. However, there is a fundamental rule for these treaty operations. To keep the integrity of treaty systems, we must ensure the action permissions held by a user would never exceed the scope of permissions that have been assigned to him.

Rule. *No increases of behaviours in any treaties are allowed.*

This is the core requirement of treaty operations, all operations and their results must obey this rule, in any representation and implementation of treaties systems.

Corresponding to behaviour attributes of sequencing, branching and terminating, there are three basic operations: concatenate, join and restrict. In addition, two operations of intersection and difference are also addressed below:

1) Restriction is a unary operation that can decrease the number of times a certain type of action can be executed to an assigned number. The actual way of designing this operation can be varied depending on different representations of treaty systems. For a treaty α, restriction will be shown: $[\alpha]_{\mathtt{a}\text{-}n}$, where `a` is the type of action to be restricted, and n is the maximum number of times that `a` can be performed.

2) Difference (–) operates between two treaties, or a treaty and action type. For two treaties α and β, $\alpha - \beta$ contains behaviours in α that are not in β. $\alpha -$ `a` removes all `a` actions from α.

3) Concatenate (•) is a binary operation which connects behaviours in two treaties in order. $\alpha \bullet \beta$ is a treaty that allows all behaviours consisting of a behaviour allowed by α followed by a behaviour allowed by β.

4) Intersect ($\sqcap$) is a binary operation which results in a treaty whose behaviour is the intersection of the behaviours of the two original treaties. Treaty $\alpha \sqcap \beta$ specifies behaviours that are allowed by both α and β.

5) Join ($\sqcup$) is also a binary operation which constructs a treaty from two original treaties, so that the holder can choose to perform a behaviour from either of the original treaties, but not both. Treaty $\alpha \sqcup \beta$ allows all behaviours in α and β.

With these operations, a user-level agent can safely construct treaties containing new behaviours from treaties that it holds, as the operations ensure that the fundamental rule of treaty systems is not broken.

B. Novel Issues Generated by Treaties

Treaties extend vistas, and thus capabilities, in functionality, and so there arise some new practical and theoretical issues that need to be considered. It must also be remembered that treaties are not only inherit positive attributes from capabilities such as their flexibility and scalability, but also some negative issues such as confinement problem and revocation problem, since treaties can also be propagated. Thus the development of treaty systems needs to concern these together with their particular challenges.

A main challenge that results from the extra functionality of treaties is the *duplication problem*. Treaties require state descriptors to keep track of the evolution of behaviours, and according to the rule of treaties, we must ensure that allowed behaviours cannot be *increased*.

Assume that a user is holding a treaty which allows him to perform a `read` action exactly once. Then the challenge is how to prevent him from validly performing more than one `read` by making a duplicate of the treaty, and then using the two copies once each. If treaties are data (bit-strings) completely stored in agents' memory, there is no way of preventing such an abuse.

The duplication problem does not only arise in this way. Taking the same example, assume the holder A sends (a copy of) his treaty to another user B, and then uses his copy of the treaty to perform the action. Later, B sends his treaty back to A, and A now has the original treaty again!

Clearly the duplication problem is a special case of a fundamental issue that must be solved for a treaty system: while there can be many copies of a treaty, they must all be associated with a *single* behavioural state, and this must be maintained by the kernel. Moreover, the situation can be seen to be even more profound when considering treaties which have been formed by the combination operations outlined above. For instance, assume user A has received a treaty from B which allows him to do a `write` then a `read`, and has received another treaty from C that allows him to do a `write` then an `execute`, both referring to the same resource. If A were to do a *join* of the two treaties and send the result to D, and D were to request the kernel to perform a `write` action, whose treaty – B's or C's – is the correct one to be synchronized? Thus the concurrency issue also affects the design and construction of treaty operations.

Generally speaking, the most important new challenges of treaties are related to states, and their management in a scalable fashion in a distributed system. This is unsurprising, as *behavioural states* are exactly the novel feature that treaties provide.

IV. Representations of Treaties

The abstraction only defines treaties generically. Any system or service that meets the requirements and enforces the rule of treaties can be considered as a treaty system. In practice, treaties can be represented in many ways, and there are a number of options for each treaty component. Each option has its strength and weakness, so service providers can choose to combine different kinds of representation depending on their requirements.

A. Referencing Style

Options for the style of referencing concern the number of resources that one treaty can refer to, and the relation and structures of these references. For example, we can choose to let each treaty refer to only one object, as in the capability approach. This is a simple choice that provides straightforward resource references, and no additional facilities are needed to identify mappings between actions and objects. Conversely, this choice means that it is not possible to operate on multiple objects simultaneously. If a user wants to do the same action to many objects of the same type, he would need to pass treaties for each of these objects separately which, in a large-scale distributed environment, would limit efficiency.

Another choice would be to allow treaties to refer to multiple objects, but hold the behaviour descriptor separately (this would be similar to a set of single-reference treaties). It would then be possible to deal with actions on multiple target objects, but since behaviour descriptors are independent, they are not capable of describing sequences of actions among different objects. This choice is particularly suitable for applications that have a number of objects of a same type (e.g. students in the university).

The third choice is to let a treaty to be a 'mix'. These treaties' behaviour descriptors would include actions to multiple objects, making it possible to describe sequences over different objects. Clearly this is the most general, and powerful option, as behaviours specified by the other two options can only refer to a unique object. The bad side of this choice is that each action in treaties' behaviour specification would have to identify the target object, so the complexity of the structure would increase greatly.

B. Behaviour Descriptors

Behaviour descriptors are required to define specifications of behaviours. Behaviours are defined as a group of actions that are combined in certain order and pattern, where an action is a single operation on a resource. Any logic, mechanism or language that is able to represent behaviours of this form can be a candidate in the behaviour descriptor part.

One obvious candidate is the finite-state machine (FSM), which is a widely-understood concept. If FSMs are chosen to represent treaty specifications, then transitions in an FSM will represent actions. The FSM is ideal for representing repetitions, sequences, branches and so on. Regular expressions would be a convenient way of specifying the FSMS treaty representations. As regular expression, actions are represented by characters, boolean '|' illustrates branching and '*' represents repetition.

Another possible candidate would be communicating sequential processes (CSP) [13, 14], a formal language that is used for describing patterns of interaction. It is suitable for treaty representations without too many modifications. In CSP representations, events illustrate actions, and choices represent branches. Repetitions can be captured by recursive definitions.

Choices of behaviour descriptors can be made depending on different applications, as different options result in different effects.

C. Storing of Treaties

The storage location of treaties directly affects the solution to the duplication problem. As has been mentioned, if treaties are completely held in users' (private) memory, it is possible that they could copy a treaty and gain unauthorized behaviours. We can choose to store treaties in a different location to avoid this problem. Otherwise, treaties can be left in users' devices, but extra security support would be required to deal with the issue.

A simple solution is to store treaties centrally. There would then be a central storage for all treaties in the service provided, and users would only hold references to these treaties. When users needed to access resources, they would show references to the kernel, and the kernel would check the corresponding treaties to allow or deny this access. Since only references would be held by users, it would no longer be necessary to manage how users copy and propagate them. However the side effect of this option is obvious: centralization significantly restricts the scalability of systems, and the performance will heavily depend on the storage's processing speed, bandwidth, etc. It also becomes fatal if the central storage fails, where all accesses will be denied.

The option of completely letting treaties to hold treaties has the highest scalability, which is very attractive in distributed systems. To realize this, it might be feasible to contain tags or time stamps in treaties for kernels to check, but this idea is still under investigation. However, it is clear that only the behavioural state of treaties need to be protected from duplication or interference. Consequently a promising compromise approach is to have state managed by the kernel – possibly in a distributed fashion – but the reference and behavioural specification parts made available for (safe) user management.

D. Further Operations

Treaty operations need not only be restricted to the five types given in section III-A. Any useful operations could be included in treaty systems, as long as they do not break the rule.

We may want to combine two treaties into one, so that with the single combined treaty, the holder can do a behaviour granted by either of the two original treaties, *or both*. This is analogous to a Boolean 'or' operation in behaviours, in contrast to the 'exclusive-or' of the *join* operation.

For convenience let us denote this *interleave* and use the symbol '|||' for the operation. For example, assuming a single resource, let treaty α contain only two consecutive actions

`read;write`, and treaty β contain two consecutive actions `execute;rename`. Then $\alpha \,|||\, \beta$ allows the following:

```
read;write;execute;rename
read;execute;write;rename
read;execute;rename;write
execute;read;write;rename
execute;read;rename;write
execute;rename;read;write
```

Moreover, the treaty resulting from this operation also indicates the maximum set of behaviours for any treaty that could result from *any* binary operation between the two original treaties.

Let $\{\alpha\}$ denote the set of behaviours allowed by treaty α, and $\diamond$ denote any binary treaty operation, then we can rewrite the fundamental rule as

$$\{\alpha \,|||\, \beta\} \supseteq \{\alpha \diamond \beta\}.$$

Another possible operation that can be involved is *interrupt*. This is an operation similar to concatenate, since it also connects behaviours from two treaties consecutively. But the interrupt operation can start behaviours in the latter treaty at any point when behaviours in the former treaty is performing, which is different from concatenate where it must wait until the former behaviours are finished. When the behaviours in the latter treaty start performing, no behaviours in the former treaty can subsequently be performed.

Service providers can develop more treaty operations as needed. Also, the way of executing these operations can also be varied. For example, for a treaty that, at its current state of evolution, allows a certain action on many objects, it would be possible to execute that action on all the objects, a subject of the objects, one of the objects (chosen randomly or otherwise), or none of them. All these choices are feasible.

V. USE-CASES

In this section a number of simple use cases are given to illustrate the benefits of treaties. Since treaties can be seen as generalizing capabilities, they inherit their advantages of flexibility and suitability for distributed environments. Moreover, they also have their own advantages of precise control. Thus treaties are particularly suitable for controlling complex and precise processes in dynamic environment.

An example which has been mentioned earlier is the voting system. In anonymous voting, the identities of voters are not needed (in fact, not allowed), but only the overall number or voting population and final result, therefore the decentralized behavioural control of treaties are very applicable. It is necessary to enforce the requirement that each voter only votes once to get the fair result. Fig. 1 shows a possible treaty-model (represented in FSM form), whereby a voter can vote once and then check the result unlimited number of times.

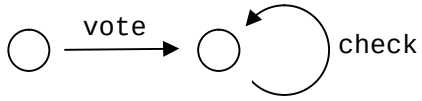

Figure 1. Behaviour model for a voting system

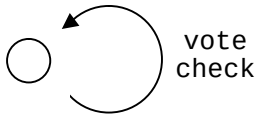

Figure 2. The complete treaty for a voting system

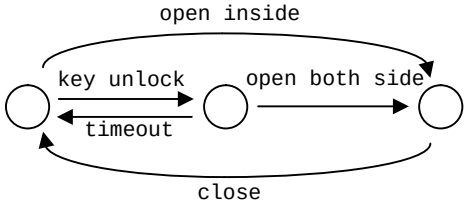

Figure 3. Behaviour model for auto-locked door systems

It is assumed that the kernel will create a 'complete' treaty which gives the creator of the resource the unrestricted permissions (Fig. 2).

Let us call this complete treaty α, then to construct the proper behaviour model, we may use treaty operations

$$([\alpha]_{\text{vote-1}} - \texttt{check}) \bullet (\alpha - \texttt{vote}).$$

A commonly used application that can be modelled by treaties is the auto-locked door system which can be found in most modern hotels. These doors will automatically lock when they are shut. A door is unlocked either by turning the handle from inside, or by using the key (usually a magnetic card) from the outside. The door will automatically re-lock if it is unlocked by a key but is not opened within a period of time. The key/card system behaves just like a treaty. The behaviour model of auto-locked doors can be abstracted as in Fig. 3.

It is notable that actions in this behaviour model will be split into two parts. The key object will have the action of `key unlock` while the lock object contains all rest of the behaviour. Without actions in the key object, the auto-locked door system would have no state transitions (assuming nobody is in the room). This example demonstrates how treaty operations can be used to split the actions in behaviours into partitions and let multiple treaties combine to form the whole model.

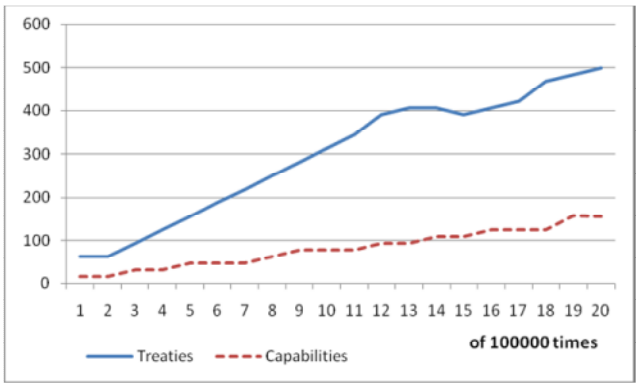

Figure 4. Time consumptions of treaties and capabilities

VI. Preliminary Results

We have built a simply-structured treaty system for preliminary testing, implemented in Java. This prototype uses FSM as the behaviour descriptor implementation for treaties, and each treaty can only refer to a single object. At this stage only local evaluations are involved, so the choice of storing locations of treaties is not critical. It has been shown by simulation that treaty system works properly.

Then capabilities were also implemented in the system in order to make comparisons, in particular the relative speeds of access between treaties and capabilities, since the workload for deciding whether to allow or deny a requested action is expected to be higher in the case of treaties: in the capability approach, the kernel needs to check if the access type is in the set of rights inside the capability, whereas in the treaty approach the kernel must first find the current state, and then find all valid actions at this state, and finally check if the access type is in this set of valid actions.

The simulation was run for the two approaches under the same conditions. The timing results are shown in Fig. 4, in which the horizontal axis gives the number of accesses and the vertical axis is the total time consumed in milliseconds for a test program with no significant computational load.

Although some anomalous behaviour is seen in the graph, possibly due to extraneous system activity, the result has illustrates the general trend of the two approaches: time consumed by checking treaties is about four times of that consumed by checking capabilities.

VII. Conclusion and Future Work

The idea of treaties has been proposed as a dynamic behavioural control mechanism for distributed systems. The abstraction of the concept makes it applicable in various situations and languages. Service providers can choose from a number of options to form different representations of treaties for different purposes, and use treaty operations to construct specifications of behaviours. The result of preliminary evaluations shows that the extra workload of the (simplified) treaty approach compared with the capability approach is in an acceptable range.

Future work will involve developing and implementing treaties operations. At the moment, accesses or actions are dealt separately, and there are no considerations about parameters and returned values, which will be required in general for consecutive actions.

Another feature, which can be called ‘outer conditions’, is an interesting challenge to be investigated. The ‘outer condition’ means states can be changed not by actions of users, but by some external conditions such as deadline times or temperature. How these conditions can be related to distributed treaties is an open question.

References

[1] J. B. Dennis and E. C. Van Horn, “Programming Semantics for Multiprogrammed Computations,” Communications of ACM, vol. 9, pp. 143–155, 1966.

[2] B. W. Lampson, “A Note on the Confinement Problem”, Communications of the ACM, vol.16, pp.613–615, 1973.

[3] L. Gong, “A Secure Identity-Based Capability System”, Proceedings of 1989 IEEE Symposium on Security and Privacy, pp56–63, 1989.

[4] A. H. Karp, G. J. Rozas, A. Banerj, R. Guptai, “Using Split Capabilities for Access Control”, IEEE Software, vol.20, pp42–49, 2003.

[5] M. S. Miller and K.-P. Yee and J. S. Shapiro and Combex Inc., “Capability Myths Demolished”, Systems Research Laboratory, Johns Hopkins University, Tech. Rep., 2003.

[6] M. S. Miller, “Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control”, Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA, 2006.

[7] J. S. Shapiro and J. M. Smith and D. J. Farber, “EROS: a fast capability system”, Symposium on Operating Systems Principles, pp170–185, 1999.

[8] D. Gelernter, “Generative Communication in Linda”, ACM Transactions on Programming Languages and Systems, vol.7, pp80–112, 1985.

[9] A. Wood, “Coordination with Attributes”, LNCS 1594 Coordination Languages and Models, COORDINATION ’99, pp21–36, 1999.

[10] N. I. Udzir, “Capability-Based Coordination For Open Distributed Systems”, Ph.D. thesis, University of York, York, UK, 2007.

[11] D. Gorla and R. Pugliese, “Dynamic management of capabilities in a network aware coordination language”, Journal of Logic and Algebraic Programming, vol.78, pp665–689, 2009.

[12] A. Wood and Y. Zhao, “Vistas: towards Behavioural Cloud Control,” in Virtualization and High-Performance Cloud Computing, 2010, unpublished.

[13] C. Hoare, “Communicating Sequential Processes”, Communications of the ACM, vol.21, pp666–677, 1978.

[14] J. Davis, “Using CSP,” Oxford University Computing Laboratory, Tech. Rep. [Online]. Available: http://www.usingcsp.com