Vistas: Towards Behavioural Cloud Control

Alan Wood and Yining Zhao

University of York
{wood,hopezhao}@cs.york.ac.uk

Abstract. Vistas, a generalisation of the *object capability* concept, are presented which provide a scalable, distributed means for specifying and controlling the interaction rights that entities have in open distributed systems such as the Cloud. The operations for combining Vistas and examples of their use are discussed. Finally, a natural development of vistas to full *behavioural* control is outlined.

1 Introduction

The Cloud is perhaps the most genuinely 'open' computing model so far proposed. Active and passive entities — agents, services and resources — can appear within a cloud at any time and start interacting with existing entities. They may also create further entities and thus set up new interactions. All this occurs without any prior knowledge of their behaviours. It is important, therefore, to provide means for constraining this anarchic situation to conform to limited behavioural specifications in order for sensible computation to occur. Since the general Cloud model is decentralised and non-hierarchical, any appropriate behavioural constraint system must scale well over manifold entities and platforms, be dynamically adaptable, and be available to the entities and not merely part of the infrastructure.

The conventional way of limiting behaviour in computational environments is the *access control list* which prescribes, for each group of entities in the environment, some subset of a fixed collection of permissions. However, even in simple monolithic systems, this mechanism has many disadvantages; in open distributed environments the problems with ACLs mean that they become untenable.

The alternative to the ACL approach – *capabilities* — was first described over four decades ago [3] in the context of hardware-mediated protection mechanisms. Over time this concept has been applied to more software-oriented areas, such as operating systems [9], languages [8], and semantics [2]. The simplicity of implementation of the ACL technique, and its choice as the access control mechanism for the most popular operating systems, has meant that it has overshadowed the capability approach. However, with the advent of persistent and open distributed systems the specific advantages of the capability idea are becoming increasingly clear. A recent trend in the use of capabilities has been to apply them in a theoretical context, as an adjunct to type systems [4,5]. This view tends to relegate them to a formal 'trick' which can be applied in early stages and then erased: a kind of catalyst that facilitates the analysis and then disappears.

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 689–696, 2011.

There are, of course, advantages in this approach: if the information is erased before runtime, then there are no runtime resources required to support it. However, in spite of these positive aspects of the *static* view of capabilities, this paper takes the view that there is potential for significant gains in a dynamic approach — capabilities should be first-class, reified runtime objects which can thus be exploited in a variety of Cloud applications. These generalised capabilities are called *vistas*, which introduce a first level of behavioural specification. A richer level of behavioural control is provided by a further extension, *treaties*, which are outlined in §5.

2 Vistas: Constraining Rights

For the purposes of this paper, a simple generic *relational* view of objects will be taken [1,6]: objects consist of a number of *names* which entities can use in ways that are appropriate to the value referred to by the name. Then a *visibility* is defined to consist of an object reference and one of its names.

A *vista* is a collection of visibilities, which define the entitlements that a holder of the vista has to interact with the objects to which the vista applies. Thus, a vista makes visible to its holder a subset of its objects' names.

Vistas constitute a generalisation of conventional object-capabilities: a vista can encapsulate visibilities for *multiple* objects. How this affects the meaning of vista expressions will be clarified in later sections, but the semantics of vistas are identical to conventional object references when they contain visibilities for a unique object.

2.1 Vista Operations

Vistas represent the objects in the system: as far as the entities are concerned, the vistas *are* the objects, in the same way that object references in an OO system can be seen to *be* the objects. Consequently an expression of the form γ`.foo` is defined to be the selection of objects called `foo` that are made visible by the vista γ. This is directly analogous to an expression such as `o.foo` in an OO language which represents the object `foo` in `o`'s scope.

However, the main novelty of the vista model is that γ might contain *several* visibilities that make `foo` available, but on different objects. Consequently, any language that is using vistas must be able to handle *multiple* selections. Some implications of this are dealt with below (§2.1). The crucial point, though, is that a multiple selection does not prescribe what the result of *using* that selection is to be — it simply states that the members of that selection are *available* to be used.

Vista Constructors By analogy to the object-capability model, vistas are obtained by an entity in one of four ways [8]:

- *Initiality.* Some vistas will be available to an entity on its creation.
- *Parenthood.* If an entity creates an object, then it has the full vista for the new object. This vista will be the only way of referring to the new object.
- *Endowment.* When an entity creates an object, it may pass to the object any vistas that it holds. In an OO system this would correspond to passing the vistas as arguments to the object's constructor.
- *Introduction.* An entity can pass vistas to another by sending a message containing them, or by returning them as a result. In a non-distributed OO system this would correspond to calling a method on the other entity with the vista as an argument. Alternatively, this could be a 'physical' communication between distributed entities.

To these means of obtaining capabilities, the vista model adds:

- *Combination.* An entity can use various constructors on the vistas that it holds to produce new vistas. It is these operations that are the focus of this paper.

There are two fundamental principles in vista-enabled systems which are direct analogues of the basic requirements for capabilities:

Requirement 1. *Visibilities cannot be increased.*

Given the set of visibilities for an object implied by the vistas that an entity holds, it must not be possible for the entity to generate a vista that has *more* visibilities for that object. Consequently, any vista operations must conform, directly or indirectly, to this requirement.

The second principle is a consequence of Requirement 1:

Requirement 2. *Vistas cannot be forged.*

That is, there can be no way to create a valid vista other than by parenthood, or the constructors given below. Ensuring this is an implementational issue, that is essentially cryptological: a bit string representing a vista can neither be altered nor created by an entity to form a valid vista. Although it might be objected that how unforgeability is to be achieved is crucial to the viability of the vista concept, it is beyond the scope of this paper. The position taken here is that either unforgeability is solvable, or it isn't. If it were proved to be unsolvable, then the viability of vistas would be the least troublesome of consequences: the ramifications for *all* security would be much more severe. However, this property of vistas is important to bear in mind when considering their use.

The basic four vista operations are:

Sum	$\alpha + \beta$	is a vista representing *all* the visibilities of α and β.
Product	$\alpha \times \beta$	is a vista that represents ordered pairs of visibilities from α and β.
Difference	$\alpha - N$, $\alpha - \beta$	is a vista containing the visibilities of α *without* those, if any, referring to the names contained in the set N. Since a vista can be regarded as a set of visibilities, this operation is naturally overloaded to allow a vista as its second argument.
Intersection	$\alpha \wedge \beta$, $\alpha \wedge N$	is the vista containing visibilities, if any, that are common to α *and* β. This is extended to work with a set of names N as in the case of *difference*.

The safety of these constructors, in the sense of Requirement 1, is guaranteed since an entity can only form a combination if it already holds the vistas involved in the operation.

Interpretations. The definitions of the constructors are given purely in terms of their meanings as vistas. Since the intention is that vistas should be used in the generic open-system Cloud context, it is necessary to define them *independently* of any particular language or computational model. However, there are issues involved in the interpretation of vistas that impact on the properties of the virtual machines that Clouds support. For instance, both *difference* and *intersection* require that entities be able to discover, at run-time, the names associated with other entities in the system. The VMs will have to be the sources of this metadata. Since entities could be written in *any* language, the use of vista values within these languages will vary, but they must be consistent with their definitions given above. This section deals with some of these considerations.

Sum. It is *this* constructor that gives rise to vistas referring to multiple objects. Therefore languages must be able to interpret an expression α`.f` which results in a *set* of values.

The ability to handle multiple selections is common in most OO languages. Although the exact mechanism varies, several procedures can be applied to reduce the selections to at most one value. For instance at compile time most common OO languages use their class system to distinguish between multiple values of an expression such as `obj.doIt(3)`. Occasionally this can't be done statically and the final selection of the object to be applied is left until run-time. There is then the possibility that *no* value can be selected in which case a run-time exception might result.

However, many other mechanisms for dealing with multiple selections exist, and not all require reduction to a unique value. For instance, in a concurrent language it would be possible to handle *all* members of a multiple selection simultaneously. Alternatively, a *non-deterministic choice* of one of the visibilities could be made: such might be the case in a logic programming language.

In order to distinguish between these various ways of dealing with multiple selections languages can provide syntactic features. For instance, a simple language would only provide expressions of the form `vista.name` which expects to evaluate to a unique object, or raise an exception. A multi-cast language might provide a `..` operator, so that `vista..name(params)` would apply each of the values — vistas— of `name` to the `params`, and a concurrent language could provide a `||` operator that would cause `vista||name(params)` to be evaluated in parallel. These 'extra' selection operators could also do some further type-based filtering of the visibility set based on the supplied parameters. However, it should be noted that the `..` and `||` operators are *derived* versions of the basic `.` operator, and don't require a re-interpretation of the meaning of the vistas involved.

Difference. This is the fundamental visibility-reduction operation, and doesn't introduce any new linguistic issues, unlike *sum*, since the set of objects to which it applies isn't increased.

However, it gives rise to an interesting interpretation in terms of a generic OO view: since the effect of a difference expression is to reduce the visible members of an object's interface, it is, in effect, creating the interface for a new super-class of the original object, and 'inserting' this into the object's inheritance hierarchy.

Although this observation is of only marginal interest to the *user* of a vista-enabled language, it suggests two things: firstly, a way of *implementing* vistas in an OO language and, secondly, that the type-system for the VMs for Clouds need to provide facilities for *this* form of interface injection.[1]

Intersection. Intersecting with a set of names is a convenient way to specify a vista with *only* a particular set of visibilities: rather than having to subtract, and thus *know*, the visibilities that a vista *could* hold, it's easier to specify the ones that are required. In this form it is equivalent to a sum of selections.

Intersecting two vistas is a necessary operation since specifying the common visibilities would be difficult using a sum-of-selections expression as the names of the two vistas to be selected would need to accessed first.

Product. This constructor introduces an embryonic *behavioural* specification. Since it represents *ordered* pairs of visibilities, the 'language neutral' statement of its interpretation is that $\alpha \times \beta$ makes visible a name from α *followed by* a name from β. That is, an application of a visibility from a product vista involves evaluating *both* the elements of the pair. The intended semantics is that they are *both* evaluated unless the first fails in which case the second evaluation is abandoned.

Products, therefore, mandate that *both* elements be evaluated in the order specified. This implies that, as far as the evaluating entity is concerned, the two evaluations are atomic, in the sense that the entity cannot do anything until their evaluation has completed. However, there is no implication

[1] The term *interface injection* often refers to *adding* methods to an object — here we require their *removal*. The former usage is, in essence, dynamic sub-classing, whereas vistas require dynamic *super*-classing.

of atomicity in the evaluation of the pair itself — other events in the Cloud can (and probably *will*) occur 'in between' their execution. One consequence of this is that there is no way for the executing entity to process the result of the first evaluation of the product before the second is evaluated. This requires a different behavioural model, *treaties* (§5). However, even the limited restriction on behaviour that products provide gives useful control facilities as shown in the examples in §3.

In order to use product vistas, languages need to provide means for supplying *pairs* of arguments in applications of a product vista. This could be done by means of an API method or, more satisfactorily, syntax such as `vista.<f,g>(<fs, gs>)`, which would select the `<f,g>` visibility from `vista` and then evaluate o_1`.f(fs)` followed by o_2`.g(gs)`, where o_1, o_2 are the sets of objects whose `f` and `g` names are exposed.

3 Use-Case

There are many situations which benefit from the use of vistas— due to space limitations a single example will be given here which illustrates the main operations.

Security Proxy — Introducing Sum and Product Vistas. Alice will be going to University, so her parents set up a college tuition account for her. They wish there to be some control over how it is operated: they can deposit and withdraw funds and view the current balance; Alice is to be able to deposit and see the balance, but not withdraw. So they could request a *bank* object to create an *account* object — both are, of course, represented by vistas. They could then merely give Alice a restricted version of the *account* vista: $(account \wedge \{\texttt{deposit}, \texttt{balance}\})$.

On reflection, however, they want to add a layer of security to the account by requiring Alice to supply login details before using the account's `balance` facilities, while still allowing her unrestricted use of `deposit`. To do this they create a *guard* object which exposes a `setlogin` method. This method takes parameters which set a valid user's login details, and returns a vista for a function which takes details, checks them against the *guard*'s stored user details, returns if they are valid, or raises an exception if not. The vista that is sent to Alice is:

$$alice = (guard\texttt{.setlogin(aliceDetails)} \times account\texttt{.balance}) + account\texttt{.deposit}$$

allowing her to evaluate expressions such as $alice\,.\langle\texttt{aliceDetails}, \texttt{balance}\rangle$ and $alice$`.deposit(5.96)` but both $alice\,.\langle\texttt{aliceDetails}, \texttt{withdraw(100)}\rangle$ and $alice$`.withdraw(100)` would be illegal.

Now assume that Alice enrols at her chosen college. Her parents would be able to supply the college finance office with a vista:

$$fees = guard\texttt{.setlogin(uniDetails)} \times (account \wedge \{\texttt{withdraw}, \texttt{deposit}\})$$

which prevents them from seeing Alice's balance while being able to take the fees, and deposit Alice's earnings as a conference helper in the vacation.

Finally, Alice can send *her* account vista to friends and family so that they can `deposit` useful birthday presents. Since the `deposit` name is *not* protected by the *guard*, there would be no requirement for others to login in order to increase Alice's account.

4 Related Work

As mentioned in the Introduction, capabilities have been studied for the past 35 years, although there has been an increase in interest in their particular benefits relatively recently. This is probably due to the steady increase in the importance of distributed systems research, and internet-hosted systems in particular.

The work most closely related to that reported here has been in the general area of Coordination Languages and systems, particularly those deriving from the tuple-space model. Work by Iain Merrick using the idea of 'scopes' [7] showed the power of combination operations on capability-like objects, and derived in part from earlier work on the use of attributes in coordination.

The μKLAIM language and computational model has always had capabilities at its heart: the most recent work is very close to the proposals in this paper [4], in that the research uses 'pure' capabilities to supply control on process mobility and resource protection. The novel feature in that work is the amalgamation of static, compile-time, analysis of capabilities, with the necessity of dynamic run-time, capability processing. The former is possible since all processes are written in μKLAIM, and so are amenable to a consistent static analysis. However, the non-determinism inherent in both the tuple-space model, and the openness of the system mean that there are 'residual' capabilities that cannot be analysed away statically and thus must be processed by the middleware at run-time. Unlike vistas, their model does not see capabilities as first-class, expressible values, nor does their work address the problems of heterogeneous open systems such as clouds.

5 Future Work

Vistas provide a rich and subtle vocabulary of expressions to control interactions of entities in a Cloud, but in many situations they seem not to be powerful enough to express useful *behavioural* constraints. Take the bank account example in §3: it would be more natural to require Alice to login only once and then access the {`deposit`, `balance`} names arbitrarily often. However, this is not possible with vistas since there is no way of expressing the fact that she is in a 'logged in' state. WE are extending the vista concept to *treaties* which will allow a much wider range of behavioural control, due to the introduction of *state*.

There are many ways in which behaviours may be specified: the simplest, and the one that we are currently working with, is as a finite-state machine, or regular expression, where the state transitions are labelled by the names in

the object's interface. A vista can be seen as a treaty having a single state with reflexive transitions labelled with the names in the vista's visibilities.

The concept of treaties gives significant advantages for behaviour control. For instance, due to their management of state, treaties are able to define actions with a fixed number of applications, and behaviours with 'end-states', and so garbage-collection can be *predictive*, which cannot be achieved using standard techniques in a heterogeneous, open, distributed Cloud. Treaties, due to the complexity of state-handling, give rise to significant challenges in implementation and distribution, but the potential benefits are great. Future work will be carried out to investigate the functionality and scalability of treaties.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
2. Cardelli, L., Gordon, A.D.: Mobile Ambients. Theor. Comput. Sci. 240(1), 177–213 (2000)
3. Dennis, J.B., Van Horn, E.C.: Programming Semantics for Multiprogrammed Computations. Commun. ACM 9(3), 143–155 (1966)
4. Gorla, D., Pugliese, R.: Dynamic Management of Capabilities in a Network-aware Coordination Language. Journal of Logic and Algebraic Programming 78, 665–689 (2009)
5. Haller, P., Odersky, M.: Capabilities for External Uniqueness. Tech. rep., EPFL (2009)
6. Jacobs, B.: Objects and Classes, Co-Algebraically. In: Freitag, B., Jones, C., Lengauer, C., Schek, H.J. (eds.) Object Orientation with Parallelism and Persistence, pp. 83–103. Kluwer Academic, Dordrecht (1996)
7. Merrick, I.: Scope-Based Coordination for Open Systems. Ph.D. thesis, The University of York (2003)
8. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University (2006)
9. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a Fast Capability System. In: Proc. 17th ACM Symposium on Operating Systems Principles, pp. 170–185. ACM, New York (1999)