Object Properties in the Raven System

David Finkelstein, Donald Acton, Terry Coatta, Norm Hutchinson, and Gerald Neufeld
{davef, acton, coatta, norm, neufeld}@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, British Columbia, CANADA

Abstract

Raven is an object-oriented programming language and system that supports distributed and multiprocessor computing. This paper describes the motivation and design of Raven's object property scheme. Raven properties are used to provide system services on a per-object basis. Raven is distinguishable from similar systems in several fundamental ways: the behavioral semantics of each system supported property is truly orthogonal to those of the others, allowing properties to be combined without side effects; and all properties can be assigned dynamically, in any combination, even after object creation. Property support is provided automatically by the system. This scheme provides a simple yet powerful and flexible system where every object can have the properties it requires.

1.0 Introduction

A programming environment is more than just the programming language being used, it also embodies the operating system services used by applications. Although the semantics of a programming language generally remain fixed, the system interface changes from environment to environment—e.g., BSD 4.3 Unix[1] has a different system interface from Mach 3.0. The lack of a consistent interface forces the programmer to manage the use of system services. The result is two programming models being presented to the programmer. One model is provided by the programming language, and the other is presented by the operating system services. The work described in this paper reconciles the operating system and programming language views through *properties* which control, on a per data object basis, the use of specific operating system and application level services.

Objects have a set of behaviors corresponding to the methods they understand. These behaviors are the programmer-defined application interface to the object. In many circumstances, an object needs services supplied by the underlying system. An example is saving data to disk, and these services are provided through properties. Object properties are defined as follows:

> **A property is a mechanism by which an object can inform the system about the basic services it needs.**

Once the system knows that the object requires a service, it is provided automatically each time a method is invoked on the object. For example, an object might require concurrency control to prevent multiple threads from modifying its instance data. To inform the system that concurrency control is required, the object is given the Controlled property. The system then ensures the object is protected against concurrent accesses.

Rather than have applications attempt to work with a standardized interface to these specific OS services, the Raven system itself provides to the programmers and users of objects full transparent support for all properties. It should be emphasized that there is no need for the application to invoke any special methods or call any system functions to take advantage of the services available through properties.

Current versions of Raven are running in a threads environment [10] under Unix on Sun and MIPS workstations and under Mach 3.0 on a 20-processor Sequent at the University of Washington in Seattle.

The remainder of this paper is organized as follows. Section two describes the motivation and design of properties. Section three describes properties in detail. Section four describes inter-object relationships in Raven. Section five discusses the effects of combining properties. Section six provides an overview of the implementation status of Raven and is followed in section seven by a brief overview of similar systems, comparing them with Raven. Section eight provides a summary and concluding discussion.

2.0 Raven motivation and design

The motivation for this work comes directly from problems encountered during the programming of distributed and parallel applications; in particular, there was a need to

1. Unix is a trademark of USL, Inc.

1063-6927/94 $03.00 © 1994 IEEE

control concurrent access to particular data objects. Work was also being done on providing atomic transactions and the saving to disk of data objects. The problems associated with conventional implementation approaches were highlighted when existing class libraries were used in these distributed environments. It became apparent that these classes would have to be modified or extended before they could be used in the environments we wanted. Of particular concern was the observation that an implementor of a class had to be aware of which system services might be requested by the user of the class. Additionally, different operating systems supplied some of the basic services in different formats, thereby further compounding the problem and increasing the divergence of the code for the different environments. The first step in addressing this problem was to specify a common set of services that the class library could use. These services essentially translate the particular services of the machine environment to a virtual set of services. This means that only the emulation library needs to be ported from environment to environment. The second step was to attach properties to data objects to create the interface to these common services.

Much of the initial work concentrated on combining a C preprocessing step with a C library to support the features [1] of Raven. It became apparent that these issues could be better explored in an object-oriented system where the strong data encapsulation and binding of code to data provides a highly structured, consistent and uniform environment in which to test these ideas. The initial incarnation of the Raven system [3] was developed for that purpose. The current Raven system consists of an object-oriented programming language, with a syntax similar to C's, and a runtime system that provides support for objects written in the Raven language. Raven is designed to support distributed and parallel applications in both a multiprocessor and distributed processor environment.

As mentioned, much of this work grew out of the desire to support atomic transactions. Rather than develop a transaction manager facility to accomplish this, we decided to investigate the object properties which would be necessary to support such transactions. Our initial attention focused on the properties required by a single object to support an atomic method invocation on that object. Three primary properties were required: first, since Raven was multithreaded, data had to be protected against concurrent accesses; second, object state needed to be recoverable in case of abnormal method termination; and third, instance data needed to be written to disk as whole, distinct objects.

Although it would have been straightforward and convenient to provide these properties for all objects, we also wanted Raven to be efficient enough to use for the development of general applications like mailers and editors. To maximize performance, objects should not suffer any unnecessary performance overhead associated with services they don't require. A programmer should have the ability to pick just those properties required, so that an object incurs the minimum performance overhead possible. Towards this end, we designed each of the Raven properties to be separate, distinct, and independent from each other, so that they could be chosen in the combination required by each application.

It should be emphasized that in our design, each property is independent from the others. The semantics for each of the properties is orthogonal to those of the other properties. This design is crucial to insure that property behavior remains the same regardless of how many other properties an object might have. This scheme allows for the seamless addition of properties to an object, since a programmer or user of an object can always be assured of the behavioral semantics which the object will obey. To our knowledge, this design makes Raven unique among object-oriented systems.

3.0 Raven properties

Properties provide the basic mechanism to inform the system about the services which an object requires. In addition to support for atomic transactions, additional properties were added into Raven to support the development of distributed applications. A sufficient range of properties is needed, so that a wide variety of applications, from mail servers to distributed databases, can be built. At the same time, however, the number of properties should be kept to a minimum.

Seven mutually orthogonal properties were chosen. They are: Immobile, Controlled, Recoverable, Durable, Persistent, Replicated, and Immutable. These properties are each supported independently by the system. It is not necessary that an object have any particular combination of properties. Any combination of properties can be assigned to any object of any class. By default, objects are created "plain", that is, without any properties.

The Raven system handles plain objects in the following way:

- The object exists only in RAM. Therefore, the object does not survive between reboots of the system.
- No system control exists to prevent multiple threads from executing methods within the object simultaneously.
- The object can migrate from machine to machine, either at the discretion of the system or when the object is explicitly asked to move itself.
- Any changes made to the object's instance data are immediately available and are not recoverable.

We believe that for the vast majority of objects created and used in the system, these semantics are all that are needed. Most objects will be created for short-term use and will not require any special system properties. These semantics are similar to those provided by the standard `new` operator used in languages such as C++.

3.1 Property semantics

The system handles objects differently when they are assigned properties. The seven Raven system properties are described below. Since properties are orthogonal, the system will handle any object with any one of these properties in the same way, regardless of how many other properties the object has. Each property affects an individual object, and in general the scope of the effects of a property are limited to that object. (See Section 4 for a discussion on inter-object relationships.)

3.1.1 Immobile: An object given the Immobile property cannot be migrated between machines, and remains fixed on its current host. Immobility is desirable for objects which implement machine-specific tasks, such as support for specialized devices or servers.

3.1.2 Controlled: An object given the Controlled property is protected against concurrent accesses by multiple threads which may modify its instance data [2]. Multiple readers, but only a single writer are allowed access to the instance data. Threads which cannot be granted the access they require (e.g., they wish to write to the instance data while someone else is reading) are blocked until the request can be satisfied. The Raven compiler tags each method as either a Read or a Write method, which it determines by an analysis of the statements within the method.

3.1.3 Recoverable: An object given the Recoverable property has the "all-or-nothing" property. Only when a method terminates normally are any changes made to the object's instance variables kept. If the method terminates abnormally, the state of the instance variables is reset to the state they had just before the method started.

The Raven programming language includes an `abort` statement. If the current object is Recoverable, executing an `abort` will restore the object's instance variables to the state they were in before the method started. If the object is not Recoverable the `abort` statement is ignored.

3.1.4 Durable: An object given the Durable property has a copy placed in non-volatile storage (e.g., on disk). Any changes to the instance data of a Durable object are written to non-volatile storage before the method returns control to the caller. Furthermore, the system ensures that the entire object state is written atomically—only consistent, complete objects are written. This is accomplished by saving the object state atomically using TDBM [6]. Return of control to the caller implies that the information has been updated in non-volatile storage. A Durable object's capability also survives system failure or restart. These semantics are necessary for the correct support of atomic transactions and database applications.

3.1.5 Persistent: The Persistent property is similar to the Durable property, except that no guarantee is made as to when the stored instance data will be updated. Although the in-RAM copy will be marked to be written to storage whenever a method modifies the object's instance data, Persistent objects are only written out periodically by the system to improve performance by reducing I/O traffic and increasing parallelism. It is not guaranteed that the current state of the object will be in storage at the time of a system failure. TDBM is also used to save Persistent objects. Although the in-storage copy may be a previous version of the object, it will be a complete, consistent previous version.

The semantics of Persistent are similar to those offered by the traditional UNIX file systems, however UNIX system semantics make no guarantee that the file will be written in a consistent state.

Although Durable and Persistent are very similar properties, the semantics of the two are different. Orthogonality still holds: if an object is both Durable and Persistent, it properly obeys both semantics, since the semantics of Durable subsumes the semantics of Persistence. We anticipate that for most applications, such as mail agents, simple Persistence will be sufficient.

3.1.6 Replicated: An object given the Replicated property can be replicated on different machines. The decision to replicate is made by the runtime system in an effort to improve efficiency (for example, when an object is heavily accessed by processes on two different machines). Replicated objects have weak consistency: the system does not ensure that all copies of the object always have the same state. Replication is especially useful for improving performance and building highly-available applications, such as name servers.

3.1.7 Immutable: The Immutable property prevents the object's instance data from being changed. If a thread attempts to invoke a write method (i.e., a method which can modify the instance data) on an Immutable object, a runtime error is generated.

Properties take effect only after the object has been created and its instance variables have been initialized. This

permits an object which is created with the Immutable property to be brought into a usable state before the system prevents any accesses which could modify the instance data.

Although it may appear on the surface that some of these properties are not orthogonal, that is not the case. Recoverable, Controlled, etc. provide only the semantics described, and do not attempt to provide more complex behaviors, such as atomicity. An examination of the implementation proves this point: the code which implements a property makes no references to any other properties, nor to data structures used in the implementations of the other properties.

3.2 Assigning properties to instances

Since objects in the Raven system can have any combination of properties, a programmer needs to specify which properties an object will have. Object property specification can be done in two ways:

- The class designer can specify that all instances of the class will have certain properties. The class designer does not need to write any code to support the properties, since they are all supported by the system.
- At object creation time, the programmer can specify a list of properties for the object. Objects in Raven are created using one of two different methods. The `new` method returns an instance of the class; the instance will have the class-specified properties. The second creation method is `pnew`, which takes as an argument a list of additional properties to assign to the instance.

3.3 Property inheritance

A third way in which an object can receive properties is dynamically at runtime through property inheritance. In the Raven language, an instance variable can be marked *Inherits*. When an object is assigned to this reference, the object is assigned all the properties of the object holding the reference. Since the object being assigned can also have instance variables which are marked as Inherits, the runtime system computes the transitive closure of all the references which are marked Inherits and updates the properties of all the objects so referenced. In the current implementation of Raven, an object can inherit properties from at most one other object, and there is no way to "mask out" certain properties, i.e., to specify that an object should not inherit a property dynamically. The complete set of the properties of an object are those assigned at creation time plus those it inherits from another object. When the Inherits reference to the object is dropped, the object's properties revert to those it received at creation.

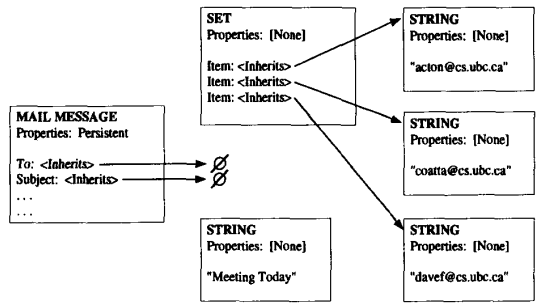

Figure 1: Some of the objects comprising a mail message, just prior to being assigned to the MAIL MESSAGE object.

As an example of the use of property inheritance, consider a mail message. All the objects which make up the mail message (e.g. the `To:` and `Subject:` fields) need to be Persistent. However, it is sufficient to simply create the main mail message object as Persistent, and have each of the sub-objects inherit Persistence from the main object. The advantage of such a scheme is obvious when you consider that objects that comprise the mail message, such as Sets or Strings, may not have been created with the Persistent property. (See Figure 1 and Figure 2.)

The ability to dynamically assign properties to an object is important in client-server systems where servers create objects for use by different clients. With dynamically assignable properties, clients can be assured that the objects they use will have those properties they need. The behavior of the object can be specified by the user of the object, and the creator need not know in advance which properties to assign.

Property inheritance should in no way be confused with the traditional notion of inheritance which describes the inheritance of methods and behaviors by one class from another. Objects can inherit properties from other objects of any class.

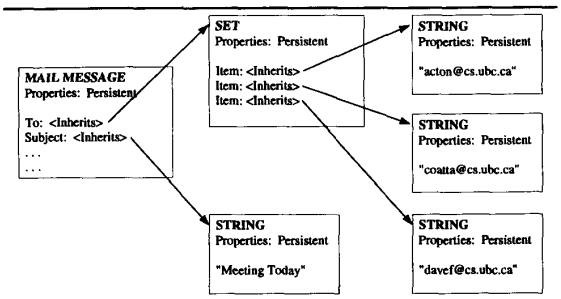

Figure 2: After assignment, objects have new properties and behave accordingly.

Currently, there is no facility in Raven to prevent objects from acquiring properties dynamically, nor to specify that they should not be created with a particular property.

4.0 Inter-object relationships

When Raven properties are assigned, they affect a single object. The scope of the behavior of that property is limited to the object itself. For example, consider a Recoverable object *A* which invokes a method on a Recoverable object *B*. Even though *A* is also Recoverable, once the invocation on *B* completes, the state of *B*'s instance data cannot be restored to the state it was in prior to the method invocation.

Although it is a very convenient notion to limit the scope of property behavior to a single object, it is often necessary to extend the scope to an entire calling chain. To build transaction facilities, for example, it is necessary to retain all recovery information until the entire transaction has completed. To allow such applications to be built using objects constructed with Raven system properties, Raven provides the notion of a *Dependent* reference.

4.1 Dependent references

Dependence describes a relationship between two objects. Whenever an object holds a reference to another object, the reference may be marked as Dependent. Dependency has no meaning for plain objects, but affects objects with the Controlled, Recoverable, or Durable properties. It is possible to develop semantics for Dependency with regard to the other properties, but at this time we feel that they are not required.

Dependency is defined as follows. When an object *P* has a Dependent reference to another object *C* (they can be thought of as Parent and Child objects), *C* is said to be Dependent on *P*. The dependency refers to *property dependency*: the properties of the child are dependent on those of the parent. When a parent invokes a method on the child, state information associated with the properties is passed upwards to the parent when the method returns. Any actions that would normally be taken before the return from the child become dependent upon the parent. As examples of how Dependency works, consider the following situations, where the object *P* has a Dependent reference to *C*, and invokes a method on *C*:

- If *P* and *C* are both Controlled, then the access privileges (read or write privileges) associated with the invocation on *C* are retained by the thread which made the invocation and are passed back to *P*. Other threads are still blocked from accessing *C*. Since the thread now in *P* still holds access privileges, it can make subsequent invocations on *C* with impunity (assuming that the access rights held are sufficient for the invocations—if the initial invocation only required read privileges, the first future invocation that requires write privileges will block if other readers are present). The thread relinquishes control of the object *C* when its invocation on *P* returns, at which time the thread's control of *P* is also relinquished.
- If *P* and *C* are both Recoverable, then any changes made to *C* are not committed until the changes made to *P* are committed (i.e., the invocation on *P* returns normally). If *P* returns abnormally, or an `abort` statement is executed, then both *P* and *C* are restored to the states they were in before the invocation on *P* began.
- If *P* and *C* are both Durable, then any changes made to *C* are not written to disk until the changes made to *P* are written. The invocation on *P* will not return until both *P* and *C* have been written. Since the Durable property assures that objects are written in their entirety, it is guaranteed that both *P* and *C* will be written in an atomic fashion.

If *P* is a child of a third object *G*, then the state information for the properties of *C* and *P* are passed to *G*. State information about a property is kept by the thread, and passed back from a child to a parent as long as some ancestor in the calling chain has either the Controlled, Recoverable, or Durable property. Although an object can inherit properties from only one other object, there is no restriction on the number of Dependent references which can exist to an object.

4.2 Part-of references

Dependency describes a relation between objects which affects how the system handles objects with properties. In object-oriented systems, objects are often bound closely together. Consider a mail message object. The mail message is composed of many other objects: a `From:` field, a `To:` field, etc., each of which are often composed of many other objects—e.g., the `To:` field could be a Set or a List or just a simple String object. In such circumstances, the sub-objects make little sense in an isolated context. They are intrinsically part of the mail message object. This relationship goes beyond simple Dependency; the objects are tightly coupled together by the relationship the programmer has created between them. To describe such a relationship between objects, Raven allows instance variables to be marked *Part-Of*.

By specifying a reference as Part-Of, the reference is considered to be marked as both Inherits and Dependent. For the programmer, there is no semantic difference

between specifying a reference as Part-Of and specifying it as both Inherits and Dependent.

The system interprets Part-Of to imply a tight coupling between the objects. As such, the Raven runtime system can make special use of Part-Of relationships. Consider a chain of objects, $A \rightarrow B \rightarrow C$, where C is Part-Of B, and B is Part-Of A. Together, A, B, and C can be viewed as a single cluster by the Raven system. Since the objects in a cluster are tightly coupled, object clusters can be used by the Raven system in a variety of ways:

- Object clusters can be written to disk as a single unit, even if they are not contiguous in memory. Similarly, when the root object of an object cluster can be loaded in from disk, the entire cluster is loaded, since it is probable that many of the objects in the cluster will soon be needed.
- Object clusters can be migrated together between machines. Simply migrating the root object of a cluster would be inefficient, since any invocations by the cluster root object on other objects in the cluster will either have to be remote (i.e., across machine boundaries), or cause additional object migrations (with their associated overhead).

Clusters are an efficiency mechanism for the runtime system and do not alter the semantics for the programmer. Because objects can inherit properties from at most one other object, an object can be Part-Of at most one other object. The Raven system does not limit the number of references that can exist to an object that is Part-Of another.

5.0 Combining properties

Since properties in the Raven system are orthogonal, they can be easily combined. No side effects result from adding properties to objects; instead, the system simply provides the new functionality of the added property. Properties can be combined to produce the exact behavior required, while invocations on the object do not incur any overhead for properties not used.

Although many objects will have only a single property (e.g., Controlled or Persistent), more complex applications and objects require several properties. One natural pairing is Immutable and Replicated. Since an Immutable object cannot be modified, it can be replicated on many machines (or even in several places on the same machine) without any concerns for maintaining consistency between copies.

Different applications will require different pairings of properties. Consider a name service. The name service must support concurrent lookups by multiple users, provide a facility for adding and removing names, while maintaining the integrity of the service across system reboots. The objects which comprise the name server need to be both Controlled and Persistent, otherwise the name server will not function properly.

More complex systems may require more properties. By combining Raven properties and placing objects in a Dependent relationship, it is possible to create database systems which support atomic transactions. Objects in the database need to be Controlled, Recoverable, and Durable. Since the objects are in a Dependent relationship, the invoking thread has exclusive access to the objects until the transaction returns. This behavior is analogous to a two-phase locking protocol. Furthermore, any changes made to the objects are not committed until the invocation which began the transaction returns. Finally, all changes are written to disk together prior to the return from the initial method invocation which began the transaction.

6.0 Implementation summary

The Raven system consists of a compiler, a class library, a runtime support system and a threads environment. The compiler performs syntax checking and provides the basic definition of the Raven language. Properties form part of the base Raven language and are supported directly by the compiler. The output from the compiler is C code that is then compiled and linked with the class libraries to form the executable. The runtime system is responsible for the basic management of the data objects within the Raven system. In particular this support includes the actual implementation of properties. The runtime system relies upon a threads environment to provide a virtual machine description to program to. The threads package is the only piece of the system that is hardware or operating system specific.

6.1 Implementation of properties

Each object in Raven has stored, along with its instance data, special data used exclusively by the system. This data includes, among other things:

- Pointers to memory and disk management state information (if the object is Persistent or Durable).
- A 32-bit value used to hold the object's property set.

The 32-bit value is divided into two 16-bit words. The first word is used to store the set of properties the object was assigned at creation, while the second stores the currently inherited properties. Since this value is the size of an integer, we assume that it can be read and written atomically.

Property support is accomplished by using pre- and post-invocation functions specific to each property. When a method is invoked on an object, the property set is checked and the appropriate pre-functions are executed, followed by the invocation itself, and then the necessary

post-functions. Unless the pre- and post-functions are executed atomically, the ordering of the pre- and post-functions is important, in order to ensure correctness. For example, it is necessary that locks not be released until after the Durable and Recoverable post-functions have executed, because these functions may need to read or write the instance data. For performance reasons, we use a careful ordering of these functions. Pre- and post-invocation functions isolate each property from the others, and allow a simple method for testing new properties.

6.2 Raven applications

Most of our experience with properties has been with those related to controlling parallelism. For example, a simple distributed mail system consisting of a user agent and message transfer agents made use of properties to control access to message queues. A simulation of an ATM communications network made extensive use of the controlled property to control access to shared reader/writer objects. The Raven Configuration Management System (RCMS) [7] made extensive use of the Recoverable and Controlled properties. RCMS uses a declarative language to specify constraints between the various components of a distributed application and then monitors the components of the application and attempts to restore them to a valid state when constraints are violated. Recovery is used to allow RCMS to roll-back changes which have violated a constraint. Concurrency control is used by RCMS to maintain the consistency of its internal data structures.

7.0 Related work

The problem of providing system services in object-oriented systems is not new, and many solutions have been implemented. One scheme is to map traditional services into object representations. These system objects are crafted to integrate into the object model provided by their runtime systems. Another scheme is to make the distributed environment transparent and to make it appear like a single processor through the use of persistence and atomic transactions. Systems using this scheme typically provide system services through the use of method inheritance or direct typing.

Cronus [5] is an object-oriented distributed computing environment developed by BBN Laboratories Inc., that is capable of connecting groups of heterogenous computers. In Cronus each machine has one object manager per object type and the manager performs pre- and post-method processing and determines the method to execute. Raven provides a similar type of system control only it is on a per object basis.

Individual Cronos objects can be customized through user specified data. The interpretation and use of this data is the responsibility of the application. The information is advisory and only has a meaning if the object user follows usage guidelines associated with the data.

The Argus system [8] [9], from MIT, supports the building of fault tolerant, reliable and highly available distributed systems. Argus provides direct system support for atomic transactions and stable storage through guardians, which collect related data together and control access to this data through handlers (methods). The atomic transaction facilities are provided only on system defined atomic data types.

Like Argus, the Arjuna [12] system also has as its goal the building of robust distributed systems through atomic transactions and persistent objects. Arjuna supplies the system support for atomic transactions and persistent objects through specific C++ classes organized into a class or type hierarchy. Unlike Raven, multiple versions of what are essentially the same class are required if one version is to make use of the system services for atomic transactions and persistence. Additionally the support for these services is not transparent to the programmer like it is in Raven.

The approach used by Arjuna of providing system services through class inheritance is similar to Concurrent-Smalltalk[13][14]. ConcurrentSmalltalk extends Smalltalk-80 for parallel processing partly through atomic objects. Unlike Raven, a separate inheritance tree is required for classes of the same type which differ only in this atomic property.

Guide [4], like Raven, consists of both a programming language and a runtime system. Guide is designed for a multi-threaded environment with persistent synchronized objects which support atomic transactions in a distributed environment. In contrast with Raven, all objects in Guide are persistent. Guide also has support for an atomic property for objects, but requires all updates to atomic objects to be within a transaction. Raven is more flexible since it does not place limitations on the types of invocations that can be made on objects nor to the types of objects which can support transactions.

Raven differs significantly from many of the languages and systems discussed in this section in that it avoids a class explosion when classes differ only in the underlying system support required. For example, an object of an existing class can be made Persistent, Recoverable and Controlled simply by instantiating a new instance of that class with the desired properties instead of programming a new class. Unlike several of the systems described in this section, Raven's support for system services is transparent to the programmer and only requires an initial property assignment.

8.0 Conclusions and future work

This paper has introduced the concept of properties as a way for programs to control the use of operating system services with respect to objects in the Raven system. Raven defines seven mutually orthogonal properties, assignable to objects at the time of creation, that can be mixed and matched to associate the desired level of system support with a given object.

Property assignment is then extended to include property inheritance, which describes a relationship between objects for dynamically associating properties to an object after creation. The use of properties is then expanded to include inter-object relationships that take the form of Dependent relationships. Part-Of relationship is a short-hand form for the Inherits and Dependent relations.

Future work on Raven properties involves exploring the feasibility of allowing properties to be removed from an object, and allowing an object to refuse to inherit properties. These issues are particularly interesting in the context of Part-Of and Dependent relationships. Work is also being done to extend Raven to support user-defined properties. Support in this area is currently minimal, as user-defined properties cannot currently be used to customize the support provided by the system properties. Other work to be done includes allowing objects to inherit properties from multiple sources, and allowing objects to mark an instance variable with a list of properties to be dynamically assigned to an object referenced by that variable. At a different level, we hope to provide a mechanism to permit the programmer to select alternate implementation policies for system properties.

Bibliography

[1] Donald Acton. A C Toolkit for Parallel Programming. In *CASCON'92 Proceedings of the 1992 CAS Conference,* pages 395-407, Toronto, Ontario.

[2] Donald Acton and Gerald Neufeld. Controlling Concurrent Access to Objects in the Raven System. In *1992 International Workshop on Object-Orientation in Operating Systems, IWOOOS '92.* Sept. 24-25 1992.

[3] Donald Acton, Terry Coatta, and Gerald Neufeld. The Raven System. Technical Report TR92-15, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., 1992.

[4] R. Balter et al. Architecture and Implementation of Guide, an Object-Oriented Distributed System. In *Computing Systems,* 4, 1991.

[5] James C. Berets, Natasha Cherniak, and Richard M. Sands. Introduction to Cronus. Technical Report 6986, BBN Systems and Technologies, Cambridge, MA, January 1993.

[6] Barry Brachman and Gerald Neufeld. TDBM: A DBM Library with Atomic Transactions. *Proceedings of the USENIX Summer Technical Conference,* June, 1992, pp. 63-80.

[7] Terry Coatta. Configuration Management in a Distributed Environment. Ph.D. Thesis, Dept. of Computer Science, University of British Columbia, Vancouver, British Columbia. In progress.

[8] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM,* 31(3):300-312, March 1988.

[9] B. Liskov, D. Curtis, P.I Johnson, and R. Scheifer. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles,* pages 111-122. November 1987.

[10] G. Neufeld, M. Goldberg, and B. Brachman. The UBC OSI Distributed Application Programming Environment. Technical Report 90-37, Dept. of Computer Science, University of British Columbia, Vancouver, British Columbia, January 1991.

[11] S. Ritchie. *The Raven Kernel: a Microkernel for Shared Memory Multiprocessors.* M.Sc.' Thesis, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., April 1993.

[12] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An Overview of the Arjuna Distributed Programming System. Technical report, Computing Laboratory, University of Newcastle upon Tyne, UK.

[13] Yasuhiko Yokote and Mario Tokoro. Concurrent Programming in ConcurrentSmalltalk. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming,* pages 129-158. MIT Press, Cambridge, MA, 1987.

[14] Yasuhiko Yokote and Mario Tokoro. Experience and Evolution of ConcurrentSmalltalk. In Norman Meyrowitz, editor, *OOPSLA'87 Object-Oriented Programming Systems, Languages and Applications Conference Proceedings,* pages 406-415, Orlando, Florida, October 1987. Sponsored by ACM Special Interest Group on Programming Languages (SIGPLAN).