massachusetts institute of technology — computer science and artificial intelligence laboratory

New Architectural Models for Visibly Controllable Computing: The Relevance of Dynamic Object Oriented Architectures and Plan Based Computing Models

Howard Shrobe and Robert Laddaga

AI Memo 2004-005 February 2004

New Architectural Models for Visibly Controllable Computing:

The Relevance of Dynamic Object Oriented Architectures and Plan Based Computing Models *

Howard Shrobe and Robert Laddaga
NE43-839
CSAIL
Massachusetts Institute of Technology
Cambridge, MA 02139
hes@ai.mit.edu

February 9, 2004

*This report describes research conducted at MIT CSAIL (formerly the Artificial Intelligence Laboratory) of the Massachusetts Institute of Technology. Support for this research was provided by the Information Technology Office of the Defense Advanced Research Projects Agency (DARPA) under Space and Naval Warfare Systems Center - San Diego Contract Number N66001-00-C-8078. The views presented are those of the author alone and do not represent the view of DARPA or SPAWAR.

Contents

1 Background

DARPA's Software Engineering research has for some time paid attention to lifecycle costs incurred after the first system build. Indeed, the argument made for the EDCS (Evolutionary Design of Complex Software) program at DARPA was that DoD software systems in particular are deployed for very long periods of time during which the technology base and the geopolitical situation changes dramatically. Consider, for example, the B-52 aircraft, whose design began in 1948 (the preliminary design was done by a team of 6 over 1 weekend) and whose first flight was on April 15, 1952, piloted by Tex Johnson and Lt. Col. Guy M. Towsend. The B-52 entered service in 1955 and it is still very much in active service today (the B-52H model, which was first flown in 1961). On the 50th anniversary of that first flight, Townsend said in an interview:

> "None of us ever dreamed the airplane would stay in service this long," Townsend, 81, said recently in an Associated Press interview. "Three generations have flown the B-52. By the time it's retired we ought to have two more generations.
>
> "If you would have told me that then, I would have said you were out of your tree."
>
> April 11, 2002 Boeing News[1]

While the B-52 might be the extreme case, the general pattern is constant. A PBS Nova show called "Carrier," produced some years ago to illustrate life on an aircraft carrier, quoted an admiral observing that the ship was considerably older than was the average crew member serving onboard. Military systems are deployed for a long time, the world changes drastically both politically and technologically during that time while the system is constantly upgraded and modified to meet new demands.

Over the last two decades, it has become increasingly useful to move functionality from hardware into software, precisely because software is considerably more maleable. The EDCS program took the position that the maleability of software was its prime asset and "one optimizes best when one optimzes for change", even when this flexibility negatively impacts performance. This position has become increasingly obvious as Moore's law has led to 3 orders of magnitude improvement in raw cycle rates over the last two decades.

However, the steady process of packing increasing functionality into software has led to systems of enormous size and complexity whose operating characteristics

[1] http://www.boeing.com/defense-space/military/b52-strat/b52_50th/ff.htm

are hard to understand or predict. When security concerns enter the equation this assumes even greaeter importance. Unauthorized users who break into a system can cause significant damage and their presence is difficult to detect. This has led major vendors to release a constant stream of security oriented "service packs" or "software updates" to address one newly discovered vulnerability after another. It is certainly expensive for the vendors to keep producing the patches needed to fix these problems while also providing a constant stream of evolutionary improvements to functionality. But cost of installing these patches on the millions of computers requiring them is considerably larger, involving the time of thousands of system operators and millions of users.

It is thus becoming clear, that the total lifecycle cost is dominated not by the cost of writing the initial software, nor even by the cost of constantly evolving the software, but rather by the operational costs of monitoring, upgrading, and patching the system. Worse yet, there is an enormous economy of scale in producing software that runs on millions of computers but a nearly total diseconomy of scale in the operational costs.

This study was initially motivated by the observation of a rather frightening set of trends occuring in mainstream operating systems. Circa 2003, both the Linux and Windows 2000 releases correspond to approximately 50 million source lines of code (SLOC). During the preceding decade, Windows grew at a rate of roughtly 35% per year; Internet Explorer grew at a rate of 220% per year during its first several years. A well known rule of thumb in the software engineering community states that the complexity of a software system is roughly proportional to the square of the number of lines of code in the system; although this rule is clearly is bit bizarre, it has been sustained by the test of time, at least as a qualitative rule of thumb. The worry is that, at the current scale, we are fielding systems that cannot be easily monitored and that are so complex that even if they could be easily inspected the results would be nearly impossible to interpret. Thus, we have failed to guarantee two properties: visibility and controlability, that are basic for any operator (and particularly so from the standpoint of a security operator).

In contrast, Genera, one of the commercial versions of the operating system of the MIT Lisp Machine, contained only about 1 Million SLOC. The MIT Lisp Machine was a hardware and software research project conducted at the MIT Artificial Intelligence Laboratory during the mid-1970's and early 1980's. The system was commercialized by Symbolics Inc, Lisp Machines Inc, and by Texas Instruments. The Symbolics version of the system lasted in the commercial domain longest (it is still a product); Genera was the version of the Operating System developed by Symbolics after the commercialization.

One might question the relevance of the comparison, under the assumption that since Genera is a rather old system it would lack many features of modern operating systems. However, this assumption is largely unwarranted. Genera contained a highly capable programming environment, still regarded by many as the best that has existed, an advanced object oriented operating system, a web-server, text editor, a graphical user interface, support for the LISP, Fortran, Ada, C, and Prolog programming languages, a network file system supporting multiple protocols (including NFILE, NFS, FTP), an advanced hypertex document system, an object oriented database, a set of AI oriented knowledge representation tools, and an graphics system integrating both 2-D and 3-D graphics technologies. All this is contained with the 1 Million SLOC. It is certainly possible to argue that modern systems still present greater functionality than Genera, but it is very hard to argue that there is 55 times the functionality. There is something much more fundamental involved.

It should also be noted what Genera did not provide: it provided no access control beyond standard log-in ID's, passwords and access control lists on the file system (all of which were almost always not used because Genera is a single user operating system). However, Genera always operates in a networked environment, typically alongside mainstream systems. From the point of view of the network, it is just another node, running similar protocols. However, Genera has proven to be virtually impervious to network based attacks (except for denial of service attacks that kill the network). This is largely because Genera is written totally in Lisp and is, therefore, incapable of buffer overflows which account for about 80% of all known vulnerabilities. Even experienced users of the system, armed with the current source code have been unable to penetrate the system in a red-team experiment.

The purpose of this study is to try to understand what accounts for this radical difference and to examine whether the factors that make Genera so much more compact also lead to greater visibility, controllability, and reduced lifecycle operational cost, particularly when security concerns are added.

2 Thesis

It is our thesis that there needs to be a unity of design between the supporting hardware, the programming language and the run-time environment (including the operating system). Systems that do not have such a coherence fall apart. However, there are many different perspectives around which this unity of design might be achieved. We believe that the fundamental difference between the Lisp Machine and mainstream systems is in their choice of how to achieve this unity of design. Such unity of design promotes productivity at the programming level, but also at the level

of system administration and system usage, the places where we experience the bulk of system ownership costs.

The Lisp Machine presents a uniform Object-Oriented perspective while mainstream systems are unified around a perspective that we term (somewhat tongue in cheak) "Raw seething bits". There are 3 key elements that constitute an object-oriented perspective: Objects have **Identity**, **Extent**, and **Type**. By identity we mean that it is possible to tell whether two references refer to the same object (and not just to two copies which momentarily have equal values in all fields). By extent we mean that it is always possible to tell the bounds of the object within memory. By type we mean that each object is located within some type hierarchy, that its type is manifest at runtime, and that the set of operations that the object may legitimately participate in is determined by its type. The Lisp Machine builds support for this object abstraction into the hardware (e.g. there are type tags and extent markers on each word of memory, see section 8), into the programming language (Lisp is the oldest programming language built around such an object abstraction) and into the runtime environment (e.g. the memory system is a single level garbage collected store hosting all applications as well as the "kernal").

The maintenance of these three key properties of the object abstraction imposes invariants on how memory is organized and these invariants in turn lead to constraints on how the computational elements may behave. For example, this implies that memory is referenced not by raw addresses, but rather by offsets within an object and that these offsets are always bounds checked. A consequence of this perspective is that buffer overflows are impossible. Another consequence is that memory is managed by creating objects, not by allocating arbitary blocks of memory, and that there is no need for a dangerous"free" operation to reclaim (often incorrectly) allocated memory. Rather, memory is garbage collected to reclaim exactly those objects that are already inaccessible thereby presenting an illusion of an infinite pool of objects. This leads to a strikingly simple model: There are objects and there are function calls and there is need for little else.

In contrast, mainstream systems such as Unix and Windows are unified around a much more primitive notion. The hardware maintains only the illusion that memory is grouped into bytes (or words), the principal programming language (`C`) also regards memory as a collection of bytes (e.g. one can always cast any integer into an address of an arbitrary byte), while the operating system imposes few other constraints. Because there are so few constraints in this model (which is why we use the term "Raw seething bits") there are few guarantees. This is why it is necessary to erect a special barrier between the kernel and the applications, the kernel cannot trust other code. This barrier then leads to a need for variety of special purpose mechanisms to

Fundamental Unit	object	byte
Properties	identity, extent, type	none
Memory Model	single object store contains everything	kernal and applications
constaints	bounds checking	none
fundamental services	object allocation, garbage collection	malloc, free
procedure invocation mechanisms	function call	system call, procedure call, interprocess communications, and more

Figure 1: Comparison of Object Abstraction and Conventional Model

get around the barrier (e.g. copying buffers between kernel space and user space). In effect, because of the lack of system-wide constraints, each component of the system tries to find a safe haven in which it can hide from the other components, making the sharing of common infrastructure very difficult and causing the need for yet more special purpose mechanisms (e.g. interprocess communication mechanisms).

Further complicating the issue for mainstream systems is the fact that this unity of design has long since been abandoned as Object-oriented ideas have crept into these systems. The introduction of GUI's in particular has created a need for some form of object orientation, but the resulting designs lack coherence (this is particularly true in the Windows world where there have been a succession of COM-like models).

Systems with a unity of design often achieve enormous power while remaining elegant and simple. An illustration of this is presented by looking at the interaction between 3 fundamental components of a system: The user interface, persistent storage and computational modules. In the original UNIX system all three of these elements were unified around the concept of a byte-stream (connected by pipes and filters). Because all three of these elements manipulated a common representation they were interchangeable: Anything a user could type could instead be piped from a file or sourced by a computational module; anything that could be presented to the user could instead be piped into a file or into a computational module. Given a good collection of basic computational modules, it is trivial to assemble more complex behaviors. Indeed, particularly for system administrators, this is the basic tool of end-user programming to this day; it has led to the development of special purpose scripting languages (e.g. PERL).

As mentioned earlier, later day UNIX and Windows based systems have lost this

coherence. The OSX generation of the MacIntosh system illustrates this quite clearly (precisely because it is one of the best of the modern generation of systems): It is a UNIX system at its core, veneered over by a MacIntosh style GUI, and although interaction between the Unix based tools and the GUI based tools is possible, it isn't nearly as fluid or natural as one might desire.

The Lisp Machine software system itself never quite reached this same level of synergy, although it was understood eventually how to do so and the principal components of the substrate were built. The basic Object-Oriented computing model was provided by Common Lisp and its object system CLOS (The Common Lisp Object System). Persistent storage was provided by an Object-Oriented database called Statice (which was the first commercial OODB). An Object-oriented GUI was provided by CLIM (the Common Lisp Interface Manager). However, each of these had predecessor systems (e.g. MacLisp, ZetaLisp, an earlier object system called Flavors, the predecessor of CLIM was Dynamic Windows; there was an earlier version of Statice) which were firmly embedded in the environment and never completely replaced (thus, going back to our comparison of the size of Genera to that of UNIX or Windows it is worth noting that Genera was also a bloated system with more than 1 version of everything). To highlight the key ideas in the rest of our discussion, we will describe a somewhat idealized version of a Lisp Machine that employs a single idealized version of each of these sub-systems.

In the rest of this report we will explain how each of these components work, how they serve to create a simple, elegant and transparent environment. We will in turn survey the following idealized Lisp machine facilities:

- The Object Model of Computing
- Aspect Oriented Computing
- The Object Oriented Data Base
- Object-oriented, Presentation-based User Interfaces
- Hardware and OS support for the model

We will then turn to the question of how a pervasive security mechanism can be integrated into this framework and to make the overall system both visible to and controllable by its operators. As we do so, we will keep in mind a set of questions that a visible and controllable system should be able to answer for its operators:

- What are you doing?

- How did you get here?
- Why are you doing it?
- Are things behaving reasonably?

We will show that the object abstraction provides very good answers to the first two of these questions but that it is not completely adequate for the last two. We will suggest another layer of structuring that builds on the object abstraction and that can provide answers to these more difficult questions. We call this layer "the plan abstraction".

3 The Object Model of Computing

The illusion that Genera tries to create is that of an eternal, evolving environment. All data and code lives within this environment; computations act on these data by invoking functions. There is really no equivalent notion to that of a "job" in a conventional system; "jobs" begin by creating a brand new, isolated memory image which is destroyed when the job terminates. Instead, in Genera there is only the object abstraction.

In this section we explore how the Object Abstraction was manifested and elaborated within the Lisp Machine environment. As we said earlier, the key element of the object abstraction is the viewing of memory as a pool of objects, rather than as unstructured bits and bytes. This means that there are a set of basic conventions about how storage is structured that must be maintained as a system wide invariant. The most basic of these is that memory is regarded as a single level pool of objects, that all storage is referenced only through **Object References**, and that object references are typed and must refer to objects of the indicated type. This, in turn, leads to the observation that any object has three key attributes:

- **Identity**: The conventional notion of a pointer is replaced by that of an *Object Reference*. Modifications of an object's structure are coherently visible through all references to it. There is no operation for object freeing; storage is reclaimed only when there are no object references to the object.
- **Extent**: Object may only be accesses through Object References and only by accessors that provide bounds checking. There is no pointer arithmetic. The bounds of an object are manifest at runtime.
- **Type**: Determines the set of operations that the object may legitimately participate in. The type of an object is manifest at runtime.

Although these basic features of the object abstraction are simple, they lead quickly to a several consequences and elaborations that will constitute the body of this section:

- Polymorphic operators (section 3.1)
- Type hierarchy (section 3.1)
- Garbage Collection (section 3.2)
- Reflective Class System (section 3.3)

3.1 Polymorphism and Type Hierarchies

Consider the simple act of adding two numbers. In Lisp this is written as follows:

```
(+ 5 10)
(+ 5.0 10)
```

Notice that the same operation + is used to add integers to integers as is used to add a floating point number to an integer. Although every programming language (whether using prefix or infix notation) allows such notation, few follow it to its logical conclusion. The notation implies that + is an operation that operates on 2 operands, that these operands are of any numeric type, and that the result is calculated by a process that depends on the types of the operands. In the first of the two examples, integer arithmetic is used, producing an integer result. In the second example, floating point arithmetic is used, after the second operand is converted to a floating point number and the result is floating point. Integer arithmetic that overflows the machine's precision produces a result of type **bignum** (i.e. arbitrarily extended precision integer) rather than either trapping or losing precision.

Thus, even as basic an operation as addition, is polymorphic. In effect, one can think of the behavior of plus as actually being specified by a set of **methods**, one for each combination of legitimate operands. Each method specifies how to conduct the operation on operands of that type. Since Lisp regards all operations as **functions**, the term **Generic Function** was coined to describe a function whose behavior is actually specified by a set of methods. If the generic function is presented with a set of operands for which there is no legitimate operation, then the operation is illegal and an error is signaled. The Lisp Machine hardware (section 8) actually implements this capability for arithmetic operations and provides support for the general method dispatch mechanism.

A second implication of this simple example is that types form hierarchies. Plus, as an polymorphic operation, is obviously defined to operate on a pair of numbers, but there are a variety of numeric types: Floats, Integers, Rationals. These types also have subtypes: the hardware types single-floats and double-floats, Single precision integers and bignums, etc (see Figure 2).

Also notice that, in contrast to certain Object Oriented languages (e.g. Java), the specific method selected to implement the operation is dependent on the types of all the operands, not just the first. Also this selection is made in terms of the runtime types of the operands rather than based on what can be determined by type inference at compile time (as is done in Java for all but the "receiver"). A compiler is free to optimize this dispatch if it can prove what the types will be at runtime;

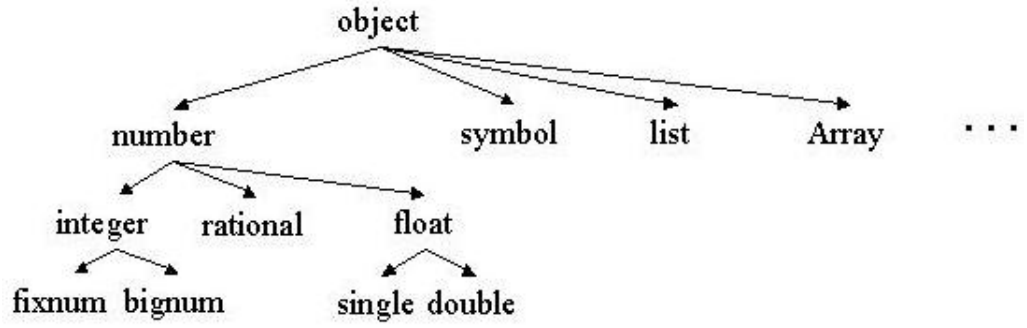

Figure 2: Type Hierarchy

but the semantics is dynamic (or late binding). The need for this is can be seen from the following simple example:

```
(defun my-operation (x y z)
  (declare (fixnum x y z))
  (+ (+ x y) z)
  )
```

Here it is known that `X` and `Y` are single precision integers (i.e. `fixnums`), but, it is still not know whether the result of adding the two will overflow the machine's limited precision. Thus, the second addition, which involves the result of the first addition and `Z`, may need to be an extended precision add and there is no way to tell this until runtime. Accepting this dynamic semantics as the base and then optimizing where it is provably permissible to do so, leads to a much clearer, simpler semantics, and much safer execution model.

It is necessary that newly created objects have contents that respect all of the storage conventions required by the object abstraction. In particular, their internal contents must have types that make sense from the moment the storage allocated to the object is made visible. Object initialization may be a rather complex process (and, in fact, the object system described in section 3.3 has such a complex initialization process); it is therefore, necessary that the allocated storage be initialized will a special value, an object reference which participates in no legitimate operations. This is accomplished by initializing storage with the **NULL** type; all operations will will signal an error when passed such a reference as an operand, because there is no way to define a method that operates on NULL objects (NULL should not be confused with **NIL**, which in CommonLisp denotes both the empty list and false.

Many functions accept NIL as an legitimate operand). The various slots of an object are filled in with legitimate object references as part of the extended initialization process. However, the fact that all storage starts off initialized to a known and safe value it critical to maintaining storage conventions.

3.2 Garbage Collection

The maintenance of storage conventions is a key system wide concern. We have just seen that this affects the way objects are initialized and places a requirement on the type system. The fact that memory is structured as a single address space containing a pool of objects and that object references must refer to objects of appropriate type implies that storage should not released by program action since this might release a chunk of memory to which there are still active references. These existing references might be inconsistent with the types of the reallocated storage. I.e. explicit **free** operations can lead to dangling pointers that violate storage conventions.

The alternative is to make the reclaiming of storage a system wide service that is guaranteed to respect storage conventions. Such a single mechanism replaces dozens of special purpose storage management systems spread throughout conventional system. In a system with a very large address space (and even more so in one with real-time requirements) it is untenable for storage reclamation to suspend regular processing. Instead, incremental copying collector algorithms were developed based on the fundamental algorithm developed by Henry Baker [1]. We will refer to these algorithms and the service they render collectively as the **GC**.

In this section we will explain how these collectors work and outline their requirements. In a later section 8 we will explain what hardware is required to efficiently support the algorithms. We will also try to illustrate how the clean structure of these algorithms make its possible for the system to be aware of its own progress through the garbage collection process.

One basic goal of the GC is to create compact storage. This means first of all that the reclaimed space is a single, contiguous large block of free memory, allowing allocation of arbitrarily sized objects (an alternative strategy might result in a linked set of blocks of varying sizes). Secondly, it means that the preserved objects are contiguous and furthermore that objects tend to be laid out in memory near to the objects that they reference (and are referenced by).

To achieve these goals, the GC breaks memory into separate regions:

1. **Old Space**: Contains the objects that are being reclaimed as well as objects that are inaccessible.

2. **Copy Space**: Contains copies of those objects in old space that are still accessible.

3. **New Space**: Contains objects newly allocated during the GC process.

By accessible we mean that the object is either one of a set of **root objects** (such as the contents of the stack and global registers). or that it is referenced by an object reference in an accessible object. Thus, the accessible objects are a rooted tree containing all objects in the transitive closure of the object-referencing relationship. During the GC process, no new objects are allocated in Old Space. Instead they are allocated in New Space; after the GC completes, Old Space is reclaimed, while Copy Space and New Space are treated as a single space in which allocation continues until there is not enough free storage remaining. One can think of Copy space and New Space as growing from opposite ends of a contiguously allocated region (although in practice they may each be drawn a page at a time from a pool of free pages).

Old Space	Copy Space – >	< – New Space

The GC algorithm maintains three pointers during the process:

1. **The Allocation Pointer**: Marks the place in New Space where the next object may be allocated.

2. **The Transport Pointer**: Marks the place where the next item will be relocated to in Copy Space.

3. **The Scavenge Pointer**: Marks the first object in Copy space that has not yet been scanned for pointers to Old Space ("scavenged").

The GC is governed by the following invariants:

1. The processor contains no references to Old Space.

2. Objects between the beginning of Copy Space and the Scavenge Pointer contain no references to Old Space.

3. Old Space may contain no references to objects in either New Space or Copy Space (except for a special type of reference called **GC Forwarding** pointer.

4. Objects in New Space contain no reference to Old Space.

The GC is initialized by placing copies of all the root objects into Copy Space, advancing the Transport Pointer to point just past the last such object. There are two principal operations that constitute the work of the GC:

1. **Transport**: This copies verbatim an object from Old Space to Copy Space. The object is copied to the location referenced by the Transport Pointer, which is then advanced past the copied object. A **GC Forwarding** pointer is placed at the site of the original object in Old Space; this points at the new copy of the object in Copy Space. The initial step of the GC amounts to transporting the root objects.

2. **Scavenge**: This examines a single object reference in Copy Space at the location pointed to by the Scavenge Pointer. If the reference points to Old Space, it is replaced by a new reference to an object in Copy Space. There are two cases that govern how this reference is created. If the referenced object in Old Space is a **GC Forwarding** pointer, then the original object has already been transported and the GC Forwarding pointer points to the new object. The reference in Copy Space is changed to refer to the copied object. If the referenced object in Old Space is not a GC Forwarding pointer, then the object has not yet been transported. In that case, the object is transported and the reference is changed to refer to the version of the object in Copy Space. After this operation is completed, the Scavenge Pointer is advanced past the examined object reference.

Both Transporting and Scavenging are conducted as atomic operations. This has implications for the ability of such a system to behave in a strict real-time regime. As currently stated, the transport operation could require unbounded time, since it needs to copy objects of arbitrary size. We will return to this issue below.

Given the invariants stated above, it follows that when the Scavenge Pointer is advanced to the same place as the Transport Pointer, then there are no references in Copy Space to Old Space. Similarly neither the processor nor New Space can contain references to Old Space. Finally all the root objects are in Copy Space. In other words, there are no references to Old Space within the tree rooted at the root objects, and therefore Old Space consists solely of inaccessible objects. At that point, old space may be reclaimed.

To maintain the invariants it is necessary to guarantee that the processor never contains a pointer to Old Space. At the beginning of the process this is guaranteed, because the first step is to transport all of the root objects (which includes all of the processor state). Thus, all that is required is a **Read Barrier** that causes the

processor to trap when attempting to load a reference to Old Space. This trap is called a **Transport Trap**; it is serviced by transporting the referenced object from Old Space to Copy Space and loading the reference to Copy Space instead. New Space initially contains no references to Old Space (since it is initially empty). As Objects are allocated in New Space, they are initially filled with NULL pointers (since these contain no address, they certainly don't contain references to Old Space). After that, normal code initializes the contents of the newly allocated object; but since the processor only contains references to Copy Space or New Space, the contents of a newly initialized object can not include a reference to Old Space). Finally, since the processor can't contain a reference to Old Space, there is no way that it can store anything into old space; therefore there is no way that it can store into old space a reference to New or Copy Space.

Maintaining a Read Barrier implies that every read must be checked. This would be prohibitively expensive if implemented in software. However, the Lisp Machine hardware provided such a trap with a minimal amount of hardware (see Section 8). However, given the existence of a Read Barrier, user processes can conduct their operations as normal; transport traps will occur and be serviced behind the scenes.

To guarantee that the GC eventually completes, a special **Scavenge Process** is spawned. This simply loops, scavenging one item per iteration. As long as the scavenger process has a large enough duty cycle, the Scavenge Pointer will eventually catch up to the Transport Pointer and Old Space will be reclaimed.

We now return to the question of establishing upper bounds on the transport operation so that real-time demands can be met. There are two approaches to this problem. The first is to erect a programming methodology in which there is an upper bound on object size. This directly leads to an upper bound on the length of time taken by the transport trap; however, this is a relatively inelegant solution that shifts the burden to the programmer. A second approach, which was prototyped in a real-time variant of the Genera, called Minima, instead worked by modifying how transport traps were serviced. In this version, the transport trap would allocate enough storage for the whole object, but it would not actually copy all of the object at that time. If the object was larger than a single page of memory (256 words in the Ivory chip version of the processor), the part of the object that would reside on the first partial page in Copy Space would be copied, as would the part that starts its last partial page in Copy Space. All the intervening pages of Copy space would be marked in the page table as being in the "Needing Transport" state. Any attempt to reference such a page would initiate a trap that would transport one page of data from Old Space to Copy Space. This strategy puts an upper bound of one page of workload on each transport trap.

The GC described above allows interlacing of normal user "Mutator Processes" with the Scavenger process and can be made to meet real-time demands. However, the total length of time and amount of work required to complete a full GC is still proportional to the amount of accessible objects in the system and this number may be very large indeed.

To deal with this, a variation of the basic GC, called **Generational** garbage collection [5] was developed. The key observation driving this strategy is the realization that some objects are far more volatile than others. Some objects become inaccessible very soon after their creation, while those that do not, tend to remain accessible for a long time.

Generational GC breaks the memory up into an ordered set of generations of decreasing volatility (i.e. Generation 0 is most volatile followed by generation 1, etc.). The GC is then modified to only reclaim a single generation at a time, breaking that generation up into its personal Old, New and Copy spaces. The invariant that the processor must maintain in order to carry out this strategy is that it must know the location of all references from higher level generations to lower level generations. To GC generation `X`, for example, one finds references to generation `X` in all higher generations and treats the referenced objects as the roots of the generation GC. The copying collector described above is then used to reclaim the old space of generation 0. Objects that survive a number of GCs at level `X` may then be promoted into level `X + 1` by forming the Copy and New Spaces of level `X` inside generation `X + 1`.

The key piece of information required to enable this strategy is the knowledge of references from higher to lower generations. Let us assume that the system initially contains no such references (because the lower level generations are empty). Alternatively, we can assume that it knows all such reference because it scans the entire memory at startup. In either case the invariant is initially met. Thus what is required is to notice when a reference to a lower generation is stored in an object that resides in a higher generation. This is called the **Write Barrier**. In the Lisp Machine system, the write barrier is serviced by noting the page on which the reference was stored. A generational GC (which the Lisp Machine calls **Ephemeral GC**) is initiated by scanning all such marked pages for pointers into the generation desired. Since memory writes are considerably less frequent than memory reads, the Write Barrier is often implemented completely in software in conventional hardware systems. The Lisp Machine hardware however provided a Write Barrier at very low hardware cost (see Section 8).

We mention briefly a few other features that were part of the Lisp Machine GC suite. First, there was provision made for **Static Space**, this is an area of memory that is never transported itself, but which is scanned for references to other spaces.

Conceptually, this is merely an generation that is less volatile than any other; the write barrier can be used to help improve the performance of scanning Static Space. Finally, the Lisp Machine provide for **Temporary Space**, a region that nothing else should reference and that is reclaimed as a whole. The Lisp Machine system did not use Temporary Areas for its own purposes, but allowed programmers to create them. This was an unsafe operation since nothing guaranteed the absence of outside references to the temporary area. The use of a write barrier would have guaranteed this; if there were outside references, then a Generational GC would be used to preserve the accessible objects. However, with rare exception the need for temporary areas was supplanted by the use of Generational GC.

In closing, it is worth noting that the entire state of the GC is held in a few registers (e.g. the Transport and Scavenge pointers, the sizes of the various areas and the percentage of each that is still free). These were all packaged up as a single object representing the state of the GC. Interfaces to this allow one to see the state of the GC system. Here is an example:

```
Status of the ephemeral garbage collector:   On
First level of DW::*HANDLER-EPHEMERAL-AREA*: capacity 10K, 64K allocated, 1K used.
Second level of DW::*HANDLER-EPHEMERAL-AREA*: capacity 20K, 0K allocated, 0K used.

First level of SYS:WORKING-STORAGE-AREA: capacity 196K, 512K allocated, 11K used.
Second level of SYS:WORKING-STORAGE-AREA: capacity 98K, 4096K allocated, 16K used.

Status of the dynamic garbage collector:   Off
Dynamic (new+copy) space 9,928,084.  Old space 0.  Static space 23,235,780.
Free space 106,103,808.  Committed guess 75,686,549, leaving 30,155,115 to use before
your last chance to do Start GC :Dynamic without risking running out of space.
There are 47,825,718 words available before Start GC :Immediately might run out of space.
Doing Start GC :Immediately now would take roughly 56 minutes.
There are 105,765,174 words remaining before Start GC :Immediately In-Place
might run out of space.  (The current estimate of its
memory requirements is 338,634 words.)
There are 106,103,808 words available if you elect not to garbage collect.

Garbage collector process state: Await ephemeral full
Scavenging during cons: On, Scavenging when machine idle: On
The GC generation count is 414 (1 full GC, 4 dynamic GC's, and 409 ephemeral GC's).
Since cold boot 20,280,950 words have been consed, 21,799,639 words of garbage have
been reclaimed, and 29,093,782 words of non-garbage have been transported.
The total "scavenger work" required to accomplish this was 179,631,816 units.
Use Set GC Options to examine or modify the GC parameters.
```

The GC is critical to *establishing* the object abstraction but its implementation operates *beneath* that abstraction. In particular, the Scavenge Pointer, Transport Pointer and Allocation Pointer are all *sub-primitive* elements in that they are raw pointers upon which pointer arithmetic is performed. However, the Lisp Machine's GC is implemented entirely in Lisp; this required extensions to Common Lisp to create and manipulate pointers, thereby exposing capabilities that operate below the object abstraction. Pointers in the Lisp Machine are called **Locatives**. Locatives are typed references, not pure raw pointers, however, facilities exist at the sub-primitive level to create and modify any field of a locative, thereby allowing unbridled access

to memory, including access that could violate the storage conventions. All such facilities are kept in a special namespace, to indicate to the average programmer that they should not use such facilities; however, they are not in any sense protected operations. A knowledgeable user with access to the system can use these facilities to do anything. Thus, when we later discuss access control, one must remember that special steps must be taken to control access to these sub-primitive layers of capability.

3.3 The Class System

So far, our description of the Object Abstraction has focussed on the most basic layers: those which establish identity, extent and type. In general, however, one things of Object Oriented Programming in terms of class hierarchies and methods. In this section, we will describe the Object System of the Lisp Machine, which is called CLOS (The Common Lisp Object System). The Lisp Machine actually was originally built around two predecessor object systems: Flavors, a message passing oriented system with multiple inheritance but single argument dispatch, which introduced the notion of method combination that we will soon describe and New Flavors which introduced the Generic Function notation.

In CLOS the notion of type is extended to include Classes. Unlike many other Object Oriented systems (e.g. Java), classes can have multiple superclasses. In conventional Object Oriented systems the driving metaphor is *inheritance and extension*: subclasses inherit structure (i.e. slots) and behavior (i.e. methods) from their parent class and then extend that with new structure and behavior. Subclasses can also override inherited behavior by providing alternative methods.

However, given the ability to have several direct superclasses, the appropriate metaphor for CLOS, is *combination* of structure and behavior. One normally proceeds by building *Mixin* classes that provide a modular chunk of structure and behavior dealing with a specific issue (e.g. access control, screen management, transaction management, persistent storage) These are then combined to achieve aggregate behaviors (e.g. a screen management system including access control, a transactionally managed, persistent storage system with access control). Since these modular chunks often represent commonly used facilities, and since there are many of them, there is typically no strict hierarchical decomposition of these capabilities. We will discuss this issue further in Section 5.

As we indicated at the beginning of this section (page 10), the basic model of computation is functional, but functions are *Generic* dispatching to a **method** based on the types of the operands. The dispatch takes into account all of the arguments.

Conventional Object Oriented languages (e.g. Java) choose the method to invoke by selecting that method whose type signature is the most specific match to the first argument. This is entirely consistent with the "inherit and extend" metaphor. However, the metaphor of behavior combination leads in other directions.

In general, a Generic Function needs to first identify a set of methods that are relevant to the presented operands; it will then combine the behaviors of these relevant methods into a combined method that is invoked. Associated with each Generic Function is a **Method Combination** which guides how the combined method is formed. CLOS provides several built in forms of method combination, the simplest of these are:

- **AND**: Forms a sequence of relevant methods and then executes each in turn checking the return value. If a method returns NIL, the combined method exits.
- **OR**: Forms a sequence of relevant methods and then executes each in turn checking the return value. If a method returns T (more precisely any value other than NIL, the combined method exits.
- **Progn**: Forms a sequence of relevant methods and then executes each in turn. The return value is the returned value of the last method.
- +: Executes all relevant methods, adds the values returned by the methods
- **Max, (Min)**: Executes all relevant methods, returning the Max (Min) of the values returned by each.
- **List**: Executes all relevant methods, forms a list of the results
- **Append**: Executes all relevant methods, each of which must return a list as a value. These lists are then appended, forming a single list as the combined result.

However, there is a more complex form of method combination that is used as the default when no other form of method combination is specified. This is called Standard method combination.

A method intended to participate in this form of combination is defined as follows:

```
(defmethod <generic-function-name> <method-qualifier>
    <arglist>
```

```
<code>
)
```

The argument list consists of pairs, each pair contains a parameter name and a type specifier. The type specifier may be omitted, indicating that the method is relevant for arguments of any type in that position. If the type specifier is included, it is the name of class, indicating that this method is relevant only if that argument is an instance of the specified class. The type specifier may also specify that the argument must be a specific instance. When a generic function is called, the first step is to identify the applicable methods; a method is applicable if all of the operands satisfy the type specifier in the method for that argument position. The most specific method is that one whose type specifiers are subtypes of all other applicable methods. This is, by default, checked in a left to right order, although one is allowed to specify an alternative ordering. This means that normally one takes that method whose first argument's type specifier is most specific; if there is a tie then the next argument's type specifier is consulted etc.

The **method-qualifier** is one of **before, after, around**. The combined method consists of all of the **before**, **after** and **around** methods and the most specific primary method (primary methods are those which have no qualifier). The combined method is executed by first calling the most specific around method; if the code of that method invokes the function `call-next-method` then the next most specific around method is called, when it returns the rest of the code of the first around method continues executing. When the least specific around method invokes `call-next-method`, then all of the before methods, the single primary method and then all of the after methods are executed unconditionally. The return value of this block of methods is that of the primary method, which is returned to the invoking around method, which is free to return either that value or any other values. The ordering of the methods in the combined methods given above is the default, but other orderings may be specified in the `defgeneric` form[2].

In summary, the idea is that:

- We execute the primary method and return its value
- We execute the before methods for effect to conduct set up actions
- We execute the after methods for effect to conduct clean up actions.

[2]It is my opinion that the need to use these ordering rules to get a desired effect is typically indicative of a failure to modularize correctly

- The around methods are *wrappers* that can conditionally decide whether the primary method is invoked at all and that can modify the operands passed in and the values returned from the rest of the combined method.

Normally, the way these facilities are used is to define relatively modest sized *mixin* classes, with only a few associated methods. Each such class represents a modular chunk of behavior, often using before, after and around methods to mix its capabilities with those of other classes. We will amplify this in Section 5 on Aspect Oriented Programming.

4 Reflection

Given this class model, we now can say amplify our description of the object abstraction as follows:

1. All objects are instances of some class which acts as their type.

2. Everything in the system is an Object.

A consequence of this is that Classes, Functions and Methods must be objects and therefore they must be instances of some class. Indeed, there are a set of classes for this purpose: All classes are instances of the class called `Standard-Class`, Functions are instances of the class called `Standard-Generic-Function` and methods are instances of the class `Standard-Method`. These classes are typically referred to as **Meta-Classes** because they describe the behavior of the class system itself. Of course, these meta classes are themselves objects so they must be instances of some class. Since these are all classes, they are just instances of that class that includes all classes, which is `Standard-Class`; thus, `Standard-Class` is an instance of itself. Note that this *instance-of* relationship between classes is different that the *subclass* relationship. A class is an instance of its meta-class but it is a subclass of its superclasses; its type is its meta-class, it's constructed by combining its superclasses.

The set of generic functions that operates on Meta Classes describes how the class system behaves. Since these are generic functions acting on classes, one can define new Meta-Classes and new methods operating on the Meta-classes, leading to variant behaviors in the Class system. The set of generic functions establishes the framework for this extensible class system, we therefore think of these generic functions as constituting the Meta-Object Protocol or MOP, the framework for incrementally tailoring the behavior of the class system itself.

The ensemble of generic functions in the MOP controls the degrees of freedom available for creating variant object system behavior. CLOS provides generic functions that control how methods are added to generic functions, how the method dispatch process behaves, how combined methods are formed, etc. It also provides methods that control how instances of a class are created and initialized, how the slots are accessed. Most importantly it provides methods establishing how an instance is upgraded when its class is modified i.e. it provides a protocol for upgrading the existing objects to correspond to a new class definition, as evolutionary changes are made to the system.

In section 9 we will describe how the MOP can be used to institute an access control system for the entire system of objects.

4.1 Making System State Observable

All system state is reified in the Lisp Machine system as objects whose classes specify the interactions allowed. The system stack, key system processes (in fact all processes) have appropriate classes defined. This makes it particularly easy to develop browsers and commands that show the system state and where appropriate allow it to be modified. In section 7 we will describe the HCI tools that allow one to easily build direct manipulation interfaces for these purposes.

There is a method for the `describe-object` generic function defined on the class `basic-object`; this class is mixed into virtually all other classes. Therefore, all objects inherit a default `describe-object` method which (using methods in the MOP) finds all the names of all slots in the object and then finds the values of those slots and then prints these as a table of slots and values. More tailored methods for `describe-object` are defined for specific classes of objects; however, to first order, virtually all system state can be made visible in this way.

The stack for each process is also self-describing. Each stack frame indicates what function (or method) is executing while that function gives a map of how the stack stack frame is laid out. This allows a stack frame to be browsed; however, because the object abstraction is pervasive, every slot in the stack frame contains an object reference and the referenced object has a `describe-object` method defined by its class. Thus, not only the stack frame, but all objects referenced by it can in principle be made visible and directly manipulable. The following transcript illustrates this:

```
(defun bar (x) (foobar x (+ 100 x)))

(defun foobar (x y)
  (let ((z (make-instance 'frob :a x :b y)))
    (sleep 1000)
    z))

  the user types (bar 10) and the system doesn't respond because it's
  sleeping for 1000 seconds inside foobar.  The user hits the break key:

Break: (BREAK) executed

FOOBAR  (from LIFE:/homes/hes/trace.)

 Arg 0 (X): 10
```

```
 Arg 1 (Y): 110
 Local 2 (Z): #<FROB 401172765>

FOOBAR <- BAR

  here the user clicks on Local 2 in the display above:
(LISP:DESCRIBE #<FROB 401172765>)
#<FROB 401172765> is an object of class #<STANDARD-CLASS FROB 21012571625>
Instance Slots:
  A:                            10
  B:                            110
#<FROB 401172765>

  here the user clicks on the entry for BAR in the backtrace:
BAR
   Arg 0 (X): 10
```

At this point, the user may elect to continue the computation from the breakpoint or to terminate it.

Earlier (page 8) we outlined several questions that a visibly controllable system should be able to answer. The above is a relatively trivial illustration that it is easy to find out:

1. What are you doing?
2. How did you get here?

In the section 7 we will illustrate how the object abstraction enables the easy construction of pervasive, direct manipulation interfaces to this information.

5 Aspect Oriented Computing

It has always been clear that a simple functional decomposition of a program into hierarchical components is far to simple to capture the complexity of all but the simplest programs. Among the sort of issues that complicate things are time or event management, memory management, or keeping logically related code together and separate from logically independent code. This is true no matter what basic programming approach one takes, and so it is also true of the object oriented approach to programming. In fact, it is in the realm of object oriented programming that it is easiest to see how this additional complexity is manifested.

The simplest approach to object oriented programming has objects which inherit from a single sequence of classes, which own a set of methods, and whose methods are uniform for objects of a given class. That simple approach breaks down in every way.

If we need to log actions taken, for example, we may want to have objects inherit from multiple class sequences. This type of multiple inheritance means that objects can inherit from classes whose attributes cut across the normal class hierarchies. By collecting the class attributes for logging in one class, which can be inherited by objects of any compatible class, we are keeping that code together, and separate from orthogonal codes.

But logging may need to behave differently depending on which object classes are involved with the action to be logged. Therefore, the kinds of logging wanted may be a subset of the cross-product of the classes of objects involved in the action, rather than just a subset of the number of classes involved. Rather than arbitrarily deciding to allow some objects' methods to know a bunch of information about other objects, we can define multi-methods that differentiate over multiple argument types.

So far, we have made our data structures more highly dimensional in order to allow more axes of organization, and we have allowed our methods to be defined and dispatched according to a cross-product of potential classes of arguments, rather than in terms of a single class. The next step is to provide method combination, to give a kind of multi-dimensional approach to method definition and dispatch itself. By having before, after and around methods, one can control with a great deal of precision the computation of the effective method to be dispatched on any given call. More importantly, one provides this kind of precise control not by defining it directly, but by defining a set of orthogonal methods, each with its own self contained logic.

So the basic structures for aspect oriented computing are provided by multiple inheritance, multi-methods and advanced forms of method combination. Lisp Machine Lisp also provided other cross cutting approaches, including "Advising" functions,

and the ability to define new method combinations. Aspect-J [3] applies many of these ideas to Java. The general style is called **Aspect-Oriented programming**.

There are three main steps involved in aspect-oriented programming:

1. Define a cross cutting issue, the framework to be imposed for dealing with this issue, and the functionality to be guaranteed by that framework.

2. Identify the points of application of the aspect within the code

3. Weave the aspect code into the program at these points.

Let's consider an example involving two cross cutting issues: access control, and synchronization.

For each cross cutting issue, we create a protocol for how the desired properties are maintained. E.g. critical regions imposed by access to a lock, so before you do the operation you get the lock and afterwards you release it. Often, there is more than one protocol: e.g. another synchronization is that a single thread does the critical operations, so all other threads enqueue requests for these to happen (awt and swing screen access for example).

Next step is to identify all the points in the main line code(s) that interact with this cross cutting issue. This may be a simple as matching the names of generic functions, or generic functions with specific argument types, or generic functions invoked from within specific types of call sites. CLOS provides OK facilities for the more static of these, Aspect-J has a more extensive language all of which could be implemented in CLOS at least as easily.

Then use the language of mix-ins to weave in the code that couples the application process to the framework invoking the appropriate protocol steps. E.g. find everyplace that you draw on the screen, wrap those with around method that grabs the lock.

6 The OODB

A database is a collection of related data items, structured and indexed in such a way as to make it relatively easy to accomplish programmed access to some aspects of the data. Early work on such databases was inspired by the needs commercial systems, and assumed collections of information too large to be contained in physical or swapped memory. Other suppositions of this early work was that the data to go in data bases would be more regular than random data in a users total set of storage, that access would mostly be programmed, rather than interactive, and speed was more important than flexibility. Early structuring techniques included networks, hierarchies and trees, and relational tables.

Meanwhile, in the Artificial Intelligence community, databases were developed to reside entirely in memory, and were more highly structured and flexible. Statice [9] was the first commercial Object Oriented DataBase (OODB), and attempted to bridge the gap between these two disparate approaches. The model was to make objects in memory persistent, but also to provide classic DB capabilities: persistence, backup, transactions (acid properties), indexing and a query language:

(for-each (x <object-type> :where (<constraints on the object>)) <code operating on x>)

At the time that Statice was built, there were basically three kinds of persistent data:

1. : structured, indexed data contained in databases
2. : structured binary information, stored in files, readable only by programs, included for example program code and digital images.
3. : unstructured textual information, stored in files, readable and indexable only by humans.

We now have so many files (virus scanning of my fixed disks in the computer in my office scans over 300,000 files!), so many different kinds of structured data, so many different document types, and so many different programs, we can no longer assume that the above breakdown makes sense. In particular, we can no longer assume that a hierarchical directory structure provides sufficient semantic power to organize our varying structured binary and text information. All our persistent information should be stored as structured objects, with querying and multiple indexing supported. Arguably, the query language described above, and the indexing and transactions features, are useful features for volatile storage as well.

Thus, the unit of granularity should be the object, not the file.

In the Genera experience, it became obvious that when programming environment operates at too coarse a level of granularity things becomes awkward. It is all too obvious that individual pieces of code belong in multiple places; should we organize files by classes with all the methods for the class, or by generic function, with all the methods implementing that function? The eventual lesson drawn was that we should move to a fine level of granularity: that of the individual definition. Once we do this, we can aggregate these any number of ways: all methods for a particular Generic Function, all methods whose nth argument is of a particular class, all methods that participate in a particular protocol, etc. Most programming changes are modest sized, incremental changes to a few definitions. But when things are organized in files, this leads to small changes causing large effects; in Java, for example, a change to one line of a method causes the entire file to be recompiled. In short, file based systems provide an arbitrary organizational structure, which our tools must overcome. File based Version control systems are trying just to recreate the incrementally that per object storage would have provided in the first place.

One might argue that this makes sense for code but not for documents. However, major industry initiatives, such as Open Doc and to a lesser extent the Web, recognized that even for conventional documents, there is a modular structure of smaller units. Concordia, the document system of the Lispm, managed things in terms of "records" that were hyperlinked into multiple larger aggregations (inclusion links) as well a hyperlinked for cross reference (reference links). It's arguable that writing text in this way works better, for just the same reason that managing code in modular chunks works better.

One can easily imagine a document base consisting of all the proposals, progress reports, technical papers, bibliographic material etc. that an academic manages. This would be better managed in an indexed object base including modular chunks of textual units. If we also add to this, all the graphics, tables, spreadsheets, email, appointments and todo lists that an individual maintains and routinely links together (in his head but not in his tools) and the case becomes stronger for an OO approach to these items.

Once documents are stored as collections of objects in an object oriented database, we will begin to see the development of tools for intelligently assembling and combining components into documents. If we have a proposal and a set of experiment descriptions and a draft paper being prepared fro publication, we should be able to assemble a progress report using an intelligent program, rather than simply cut and paste manually in a text editor. The section 9 on access control will show that there is also potentially great synergy when security properties are considered.

Unfortunately, Statice was never really integrated into Genera, so we don't know

exactly what would have changed. Personal experience tells me that email might have been much better.

7 CLIM: Presentation Based Interfaces

We have spent some time examining the Object Oriented computing model used in the Lisp Machine and pointed out how it first of all allow much greater sharing of infrastructure, provides a completely reflective view of the state of the system and also supports an Aspect Oriented style of composition. We turn in this section to the examination of how the object oriented model affects the human computer interface, in particular, we will examine the an object oriented approach to multi-modal, direct manipulation interfaces. Like much of the Lisp Machine, there is history of implementations of these ideas. The earlier version was called Dynamic Windows, and actually is the dominant and pervasive HCI framework in the system. A later version, called CLIM (for which there are portable implementations) refined the ideas a bit and integrates seamlessly with CLOS (as opposed to Flavors, its predecessor object system). Where there are conceptual differences we will explain things in CLIM terminology, since it is more modern.

CLIM (the Common Lisp Interface Manager) is a thoroughly object oriented human computer interface toolkit, from which one can rapidly configure a direct manipulation user interface. It falls within the tradition of Model-View-Controller interfaces: what is on the screen is a *presentation* of information in an underlying model. In this case, the underlying model consists of a collection of application (or system) objects. The controller is a method that selects a set of objects from the model that it then renders onto the view in an appropriate form. When the model changes, the controller updates the view. The controller also responds user inactions with the view by modifying the model.

The view actually consists not just of the pixels on the screen; in addition, it contains a collection of *output records* which are objects describing what pixels should be on the screen. There are variety of classes of output records corresponding to basic shapes (polygons, ellipses, lines, etc). However, there is a special class of output record called a *presentation* which, in addition to geometrical information, also has a reference to a model object and to that object's type. Presentations are therefore the intermediary between the model and the screen.

Two generic functions form the core of CLIM: `present` and `accept`.

Present takes an object, a type and a stream (representing an output device), as arguments. `Present` creates an presentation output record referring to the object, its type and its geometric rendition on the screen. One can define *presentation* methods that describe how instances of a class of objects should be rendered on different types of output streams. More generally one can graphically render the presentation of an object in completely different ways in different contexts as follows:

```
(with-output-as-presentation (<stream> <object> <type>)
  <arbitrary graphics code
   that references object
   and draws on stream>
 )
```

CLIM's *presentation types* are actually an extension of the CLOS type system, allowing parameterizations not included in CLOS (this is a bit of a wart in CLOS). This is the primary reason why the type is specified explicitly. In general then, there may be several presentations of the some object each with its own graphical rendering and each with a unique parameterization of the presentation type.

Accept takes a stream and a presentation type as arguments. It waits input of a object of the specified presentation type. This input may occur either by typing a textual representation of the object or by pointing at any presentation of the object. One can define `accept` methods that describe how to parse textual representations of objects of a particular presentation type on a particular input stream. A call to `Accept`, establishes an *input context* in which only input of the specified presentation type is allowed; presentations on the screen that match the type specification are mouse sensitive (and highlight as the mouse passes over them) while other presentations simply cannot be selected.

Thus, what is on the screen is a direct representation of the objects in the model, respecting their identity (i.e. even though it would be a bad thing to do, identical looking presentations may still refer to distinct objects) and type.

Applications are then structured as a set of views interfacing to a common collection of objects (not that these objects may also participate in other applications because they are all still part of a single memory), through a common set of *commands*. Commands are a class of objects representing application behavior that are in effect an extension of a the notion of a function. A command takes a set of arguments each of which has a presentation type; given this information, the system generates a multi-modal parser for the command (it is multi-modal because each argument may be provided either by typing a textual description or by directly pointing at any presentation of the object). A command also has a body, an executable procedure that is passed the command arguments which are typically objects in the model; commands typically either modify some application object or control the graphical rendering or both. Since `commands` are a class of objects, there is also a presentation type for commands. Commands may be *presented*, for example, by rendering the command name in a menu. The top level loop of an application consists of *accepting*

a command, executing the command, and then updating all of the views (this is a generalization of the idea of a read-eval-print loop).

When `accept` is invoked it establishes an input context in which only input consisted with the type specified is acceptable. This includes objects that have been presented with a type that is a subtype of that requested. However, there is a second way in which an object may be made acceptable. *Presentation Translators* are methods that can coerce an object of one presentation type into one with a different presentation type. A simple example, is a translator that can turn an object presented as a float into an integer (by rounding, for example).

Presentation translators are characterized by the input presentation type, the output presentation type, and a *gesture*. A gesture is a user action (in practice a chorded mouse click, such as control-middle mouse click). A translator is invoked when the input context is a supertype of its output type, and the user makes the specified gesture at a presentation that is a subtype of its input type. The translator has a body, a procedure, that is passed the input object and produces an output of the required type.

The most important use of translators, however, is *command translators*; these translate other objects into `commands`. A typical command translator associates a particular gesture at an object with a command that operates on that; for example, clicking right on the object `foo-1` might translate to the `delete-object` command with argument `foo-1`. There are also a class of drag-and-drop translators which involve drag-and-drop gestures, producing both a *from* and *to* object.

Thus, to build a direct manipulation interface, one first thinks about the class of objects one wants to manipulate. Next one thinks about the operations one wants to perform, turning these into command. Finally, one thinks about what set of gestures should invoke these commands, turning these into command translators. Certain of the commands will appear on fixed command menus, others can be typed into *listener* of the application, others will only be invoked by appropriately gesturing at presentations; some commands will be invoked in all of these ways. This leads to enormously fluid, direct manipulation object oriented interfaces, in which one can choose how much of the internal state of the application to make visible and how much to make controllable.

One final component of CLIM is higher order formatting procedures, in particular, `formatting-table` and `formatting-graph-from-roots`. Each of these is in effect a software pattern for creating a certain type of display (tabular or DAGs, directed, acyclic graphs). For example, formatting-graph-from-roots takes 3 arguments, a set of objects that are the roots of the DAG, a function that given an element of the DAG, returns its children, and a function that displays an element of the DAG. This

last function is free to use `present` to display the element. In effect, the core of a class browser is then:

```
(define-application-framework class-browser
 ()                              ; included classes
 (root-nodes)                    ; state variables
 <... window definitions>
 )

(defmethod display-graph ((frame class-browser) stream)
  (formatting-graph-from-roots
    (root-nodes frame)
    #'class-direct-subclasses
   #'(lambda (object stream)
      (present object 'class stream))))

(define-class-broswer-command (com-set-root-node)
 ((the-node 'class))
 (setf (nodes *application-frame*) (list the-node))
 )

(clim:define-presentation-to-command-translator
  node-to-set-node
  ((class                        ;the presentation type
   com-set-root-node             ; the command
   class-browser                 ; the application it works in
  :gesture #\midddle)            ; invoked by mouse middle button
  (object)                       ; the object clicked on
  (list object))                 ; the arguments of the command
```

8 Hardware and OS support for the model

There were several hardware implementations of the Lisp Machine, beginning with wire-wrapped TTL boards in the MIT prototype, and culminating in single chip implementations (one at Symbolics, called Ivory, and one at Texas Instruments, called the Compact Lisp Machine, partially sponsored by DARPA). We will describe the features of the Ivory chip, since we have personal experience with it and also since it was described in more detail in published documentation and research papers.

A rather surprising fact is that no more than 5 percent of the real estate of the chip is dedicated to mapping the object abstraction into hardware. At the time of the Ivory chip design, real estate was tight, in the current era of abundant chip real-estate one might choose to move some additional functionality into hardware. Therefore, the features described here should be thought of as a minimal of necessary features. These features are intended to support type integrity and to provide hardware support for the memory conventions of the object abstraction.

The Ivory chip was a tagged, 32-bit machine. Each word of memory and of the internal registers consisted of a 32-bit datum plus an 8-bit tag encoding data type and memory extent information. Otherwise, the chip was a relatively conventional stack machine. There was an internal 256 word stack cache that played the role normally played by processor registers. Three registers index into the stack cache:

1. **The stack pointer**: points to the top of the stack
2. **The frame pointer**: points to the base of the current stack frame
3. **The locals pointer**: points to the first word in the stack frame above the passed in arguments. This is normal the first word of data local to the current frame

All instructions implicitly reference the top of stack and one other operand, indexed by an 8-bit offset from one of these addresses. None of this is particularly necessary for supporting the object abstraction. This style of instruction set was chosen for historical reasons (the previous Lisp Machine hardware had been a stack machine) and because it allowed a compact instruction set, with two instructions per machine word. A more conventional RISC-like, 3 address instruction format could have been used. Indeed, there was a detailed design of such a chip; however, this chip was never implemented. All of the features that we will go on to describe in the rest of this section were included in this design.

8.1 Support for the Garbage Collector

As we explained earlier the two critical capabilities required for the garbage collector are the **read barrier** and **write barrier**. These were provided by some minor features in the page table as well by two internal processor registers, each containing 32 bits:

- **The Old Space register**: The entire virtual address space (32 bits of word addresses) is broken into 32 zones of equal size. Each bit in this 32 bit register corresponds to a "zone" of memory. A bit in this register is is set to 1 if the addresses spanned by the corresponding zone are regarded as being in **Oldspace**. The software is expected to allocate virtual memory in accordance with this convention. The lowest zone holds *Ephemeral Space*, which is managed by generational garbage collection.

- **The Ephemeral Space register**: Ephemeral space is in turn divided into 32 zones. Each bit in the register corresponds to one of these Ephemeral zones. Each zone is further divided into two halves; the corresponding bit in the Ephemeral Zone register indicates which half of the zone is old-space.

These registers then allow the read and write barriers to be implemented as follows:

- **Read Barrier**: Whenever the processor reads a word of memory into an internal register, it checks the tag of word read to see if it is an object reference. If it is, then the processor uses the upper 5 bits of the address part of the word read to index into the Old Space register. If this bit of the Old Space register is set, a trap is initiated. (A trap is implemented as a special type of function call. It pushes a frame header and does an indirect jump through a trap vector to the appropriate code. One advantage of the stack machine architecture is that a trap incurs little overhead).

- **Write Barrier**: When the processor writes a word to memory it examines the data being written. If the data type indicates that the datum is an object reference, then the processor tests to see if the upper 5 bits of the address are all zero; if so, then the datum is a reference to an object in ephemeral space. Ephemeral space is broken into 4 groups, each of 8 ephemeral zones. Corresponding to each group are 4 bits in a page table entries (both in the hardware TLB as well as in the page table in memory maintained by software). The processor fetches the TLB for the page into which the datum is being

written and turns on the bit in the TLB entry corresponding to the referenced group of ephemeral space. In detail, bits 31-27 = 0 means the address is in ephemeral space; bits 26-25 of an address indicate the ephemeral group; bits 26-22 indicate the ephemeral zone. So bits 26-25 of the datum being written select which bit of the TLB is turned on. When the TLB entry is spilled into the Page Hash Table in memory, these bits are copied out. Thus each page in the Page Hash Table has an indicator set if it references a group of Ephemeral space. To initiate an Ephemeral GC, the PHT is scanned to find pages that might contain references to the appropriate zone of ephemeral space. Each such page is then scanned to find the reference if any; the referenced data form the roots of the Ephemeral GC.

8.2 Hardware Tagging

As mentioned in section 8.1 above, it is critical that the processor can distinguish object references from other types of data. To facilitate this, all words in memory or processor registers are actually a combination of a 32-bit datum field and an 8-bit tag. The tag is further broken into a 6-bit data-type field and a 2-bit field used to encode memory extent. These will be explained below.

The 6-bit data-type field encodes 64 built in hardware types. Several of these are *immediate* types; in these the 32 bit datum is not a reference to the data but is rather the data itself. Numeric data that can fit into 32-bits (i.e. single precision integers and floats) are immediate types. Instructions are immediate data in which two half-word instructions are packed into the 32-bit datum (actually the instruction data types "borrow 4 bits from the tag, so that the instructions are actually 18 bits and there are 16 instruction data-types). The remaining data-types are *pointer* data types, including array and object references, function references, list-cell references, etc.

8.3 Encoding Extent

The information encoded in a basic-object reference is somewhat limited, it includes the *identity* (i.e. the address) and the *type* of the referenced object. Notably, the *extent* of the referenced object is not included. This is instead encoded in the object itself. Data structures are arranged in one of two ways: either they are represented as list-structure or as structures with headers. List structure data are arranged as pairs of words, where the first word is a reference to the first object in the list and the second word is a reference to the rest of the list (the `car` and `cdr` of the list).

Other data structures begin with a header word, and this header includes the size of the structure.

Since list structure always consists of pairs of words there is no need for a header, the size is always 2. However, there is optimization that allows list structure to be more compactly encoded, and this is also used to encode object extent. As mentioned, each word contains an 8-bit tag split into a 6-bit data-type and 2 other bits. When possible, the references to the contents of a list are stored sequentially in memory. In this case, the `cdr` pointer is implicit and is instead code in the 2-bit extent field. This can take on four values:

1. **cdr-Normal**: The next word is a normal object reference to the `cdr` of the list. This is used when the storage is not compacted.

2. **cdr-Next**: The next word is the beginning of the rest of the list, rather than a reference to it. This is used when the storage is compact.

3. **cdr-NIL**: This word contains a reference to the last element of the list. The next word begins a new structure; the `cdr` of this word is an implicit reference to the object `NIL`, the empty List, which is by convention the end of all lists.

4. **cdr-None**: an illegal value.

Compact lists, and structures, then consist of a sequence of words whose `cdr-code` bits are all `cdr-next` except for the last word which is `cdr-NIL`. A non-compact list consist of a compact list, up to the last two words, the first of which has a `cdr-normal` `cdr-code` and the second of which has `cdr-NIL`. Thus `Cdr-NIL` always marks the end of a structure.

These bits are used to aid the GC as it transports an object to copy space, since they always mark the object's extent.

As mentioned, all non-list data structures have a header as their first word, and this word encodes the size of the object. This includes all arrays and all objects, both of which are addressed in the instruction set using an offset from the header. The memory system calculates the effective address of the data and checks that the reference is in bounds. The check is performed in parallel with the memory operation, but the hardware traps out of bounds references before any side effects occur. [Actually, all memory writes actually involve a split read write cycles because the location being written must be checked for the presence of invisible pointers. Thus, the bounds checks for writes are performed during the read part of the operation. To do this, the effective address is calculated on the first cycle; on the second cycle the memory read is issued and the bounds check is performed. If the bounds check

fails, the instruction traps and the read-data is discarded, thereby avoiding any side effects. If the cycle was a memory write, the trap aborts the instruction before the write data is ever issued.] This scheme was used to avoid the need for a second alu to do the bounds check while doing the effective address calculation (due to the limited chip area); in current chip technology, there would simply be two parallel alus and the memory operation wouldn't be issued if it were illegal. The scheme used actually doesn't slow down the memory pipeline since the bounds check is performed during the time occupied by the memory latency. Block moves (see section 8.5) perform the bounds check on the first step, allowing the memory pipeline to be kept full, while preserving safety.

8.4 Type Checking

The memory and hardware registers consist of **object references**; this is to say that each word includes both a datum and a data-type tag. In the case of the hardware supported numeric types, the data is immediate, i.e. the datum in this case is a numeric value (32 integer, 32-bit IEEE float, etc.) encoded in the format indicated by the data-type tag. Non immediate data include the address of the referenced object and its data type.

What makes the data type tags particularly valuable is that they are checked on every operation to make sure that the operands are consistent with the operation being performed. This is done in a hardware lookup table. The Ivory was a microcoded chip; one field of each microinstruction is a partial index into a lookup table, the rest of the index is formed by the data type tag of the first operand to the instruction, plus one more bit indicating whether the two operands have the same data type. This composite address then selects a line from the hardware type-checking trap table; this line indicates whether a trap should be initiated and if so, what type of trap and if not. What this means for numeric operations is that the instruction should proceed if the two operands are of the same type and both of those are either single precision integers or single precision floats. A more modern chip that provides more hardware capabilities, might code this differently, but the general approach should be clear from the above description. For non-numeric operations the tests include:

- For array referencing instructions, the test is that the data type of the operand is one of the array data types.
- For **generic dispatch** instructions (the basic step of OOP), the check is that the operand is an **instance** object type. If so, then a built-in hardware hash

function is used to support a fast table lookup in a method table referenced by the object, guaranteeing that only legitimate methods will be invoked.

- For most data motion instructions (e.g. `push` or `pop`) there is no check needed.

8.5 Other Hardware Support

The hardware also provides support for "invisible pointers"; these are used to indirect references stored in the memory. If the processor attempts to read an invisible pointer, it instead reads the object referenced by the invisible pointer (there are special memory cycle types that don't behave this way, used by sub-primitive levels of the system). The memory system checks the tags of a datum as it is read (we saw this in section 8.1 above when we discussed support for GC); if the type is an invisible pointer then the memory system turns around and reissues the read, using the address in the invisible pointer.

Another inexpensive feature is provided by a set of "block read" registers which are provided to allow fast sequential scans of memory, optionally following invisible pointers, and optionally when certain types of data are encountered. Special instructions read (or write) memory using the address in of the four block address registers; the address is incremented by 1 on each read. These are used by the GC as well as a variety of low level system utilities (i.e. page table searching).

Both page table searching and method dispatch require fast hashing. The hardware provides a hash function that operates in a single cycle to support these operations.

8.6 Summary

The total cost of these features is actually quite minimal in terms of chip area and execution pipelining. The basic pipeline used in Ivory is a five stage pipeline, with side-effects confirmed in the last stage. All traps actually operate at this stage of the pipeline; the traps that are motivated by the object abstraction impose no further complexity than would be required for conventional trapping (e.g. overflows, page table permission violations). There is nothing preventing such techniques from being introduced into a modern chip architecture, particularly given the vastly increased availability of chip real-estate (the Ivory chip had approximately 1 million transistors). The largest cost is the memory size overhead for the tags, a 25 percent increase. However, memory has now become very much cheaper than when the Ivory chip was designed, and we are (slowly) moving into the 64-bit era (where an 8-bit tag would cost only 12.5 percent overhead). Tagging, as pioneered in the Lisp

Machine, facilitates the embedding of the key aspects of the object abstraction in the hardware. In the future, it is possible that tags might hold other relevant data, such as security levels or the pedigree of the data. Tagging might also be a critical enabler for achieving greater parallelism at the chip level, because as we pack more information into the tag, the processor is better able to understand what it is doing with the data.

9 Introducing Access Control

So far we have seen that a coherent object oriented view can be constructed that uses "objects all the way down". Objects form the core of the computing model, of the human-computer interface and of persistent storage. This approach leads to a much more compact infrastructure, because there are no arbitrary boundaries the prevent sharing and because the uniform adherence to storage constraints and the object abstraction means that core capabilities are not endangered (as they would be in an environment unified around a less structured abstraction). Finally, we have seen how this approach facilitates decomposition into interacting frameworks each dealing with an individual aspect of the overall design.

The argument that there is no need to protect kernel capabilities from other parts of the system is critical, because it allows kernel capabilities and normal user facilities to share common substructure and to communicate through shared access to pool of objects. However, this argument is based at least partly on the assumption that the users of the environment are benign players. This is a reasonable assumption in many cases for a single user, personal workstation: the user make may mistakes but is certainly not trying to wreck his own environment and will not take extraordinary measures in that direction. In addition, because of the respect for storage conventions and the object abstraction, network protocol based attacks are very difficult (there is no possibility of buffer overflow, for example).

The distinction between kernel space and user space(s) provides one very useful property however. It allows a very simple proof that core system resources are not reachable by user processes and that the resources of each individual user process is not reachable by the others. In an unstructured world, reachability is equivalent to being in the same address space, because anybody has the ability to manufacture an arbitrary address and follow it. In an object oriented world, reachability is equivalent to being within the transitive closure in a graph of object references rooted in a set of root objects. Object references, however, cannot be arbitrarily minted. Thus, the task in an object oriented world is to show that a user process is born with a set of root objects that do not lead to unwarranted access to kernel resources. This analysis is actually more complicated because the process is allowed to invoke methods on the objects is has access to, and the reachability relationship is actually over the object references returned by such operations.

In conventional operating systems, however, there is another set of core system resources; these live not in memory but in the file system. These are protected not by the kernel–space user–space distinction, but by *Access Control* mechanisms, in particular, the *root* versus normal user distinction and by *user, group* oriented

Access Control Lists. Thus, there are two distinct mechanisms used to guarantee security properties in conventional operating systems, one for memory, a second for persistent storage. Furthermore, both of these operate at an overly coarse-grained level of granularity. As we pointed out in the section on persistent storage (see 6) the natural unit of storage is the object, not a file that typically contains a textual representations of a multitude of objects. This is a particularly egregious failure when considering access control; one wants to be able to state access rights concisely and clearly in terms of the natural conceptual units of the system and file based storage frequently muddies the conceptual structure.

In the rest of this section we will propose and outline the mechanisms involved in a unified, object-oriented approach to access control. The essence of this idea is:

1. We adopt a dynamic **Role-based** access control framework.

2. Each user process maintains its *Role Register* reflecting the role that the process is currently playing. The right of a process to enter into a role must be authenticated (we assume the existence of appropriate strong cryptographic techniques for this purpose and do not discuss such techniques here).

3. There is a system-wide *Condition Register* reflecting an overall assessment of threat level.

4. The contents of these two registers are typically complex objects situated within a class hierarchy.

5. We treat access control using *aspect-oriented* programming techniques.

6. In particular, we use an extended version of **Method Combination** to wrap critical Generic Functions with a new type of **around** (i.e. wrapper) method, called *access-control* methods. These run before all other methods involved in the combined method and, like normal around methods: they can modify the arguments passed on to the rest of the combined method, they can modify the values returned by the rest of the combined method, they can conditionalize the execution of the rest of the combined, they can invoke a different method instead of the rest of the combined method, and they can signal an abnormal condition.

7. A variant form of **method dispatch** is used for such protected Generic Functions. This dispatch takes into account the contents of the per process **role register** and the contents of the system wide **condition register**.

8. **Access-Control methods** are written just like normal methods, except that their signature include type qualifiers for the per-process role and system-wide condition as well as type qualifiers for the explicitly provided operands.

9. **Access-Control methods** are also provided as part of the **Meta-Object Protocol**. These methods are used to control access to meta-level operations such as installing access-control methods (without this, it would be possible to use this mechanism to launch deniable of service attacks).

In effect, what access-control methods do is to determine which "agents" (the role of the process) are allowed to perform what operations (the generic function) on what objects (the arguments passed to generic function) given an overall situational assessment (the condition register). They do this compactly, taking advantage of the class inheritance structures in all dimensions.

In addition, since both volatile (the contents of virtual memory) and persistent state (the contents of the object repository) are "made of the same stuff" (CLOS subject to the MOP), the same mechanism can be used to express the protection and privacy rules for both volatile and persistent information. Finally, because underlying object model is dynamic, one can easily make incremental changes to the access control rules by adding, removing, or modifying exiting methods.

The lowest layers of this scheme might well map into hardware capabilities similar to the tag checking features of the Lisp Machine hardware (see section 8).

10 Looking Forward to a Plan-based Model

In the previous section we have seen that a single object abstraction, like that provided in the Lisp Machine, can provide integration and synergy between the computational model, the user interface, and the persistent storage of the system. We have also seen that the CLOS class system with its Meta Object Protocol provides and elegant way to incorporate a systematic role-based access control framework. Finally, we have shown that a relatively limited set of hardware features (e.g. data-type tagging, old-space registers, ephemeral zone registers) allows delayed binding and runtime integrity checking to be provided with little impact on performance.

We have also seen that a systematic object abstraction allows the state of the runtime system to be reified, making it inspectable and controllable. Thus, for each process within such a framework there is always an explicit representation of what the process is currently doing and how it got to that point. However, the object abstraction fails to make explicit why the process is doing what it is currently doing (i.e. what is the purpose of the current step, what will it do next) and whether the overall system is behaving reasonably.

In this section, we will describe another layer of abstraction, which we call the plan layer, that can be integrated with the object abstraction. This layer of representation provides explicit representations of *purpose*, i.e. it provides an explicit representation of how the steps of a plan establish one another's preconditions, and of how they collectively achieve their overall goal. Whereas the object abstraction provides and elegant way of expressing how a computation should be conducted, the plan abstraction provides an elegant representation of why the computation works and of what must be true for it to work.

10.1 What is a plan?

The notion of viewing a program as a plan goes back to work in the early 1970's on the Programmer's Apprentice (PA) [6] and to early work in deductive automatic programming [2]. The programmer's apprentice in particular, originated the idea of preserving the plan structure as an explicit representation. The representation of plans in the PA in turn drew on work by Sacerdoti [7] on non-linear planning. We will use something very similar to the plan representation of the PA in this discussion.

A plan consists of a hierarchical decomposition of a computation into plan steps, each of which is annotated with a set of prerequisite, post and invariant logical conditions; these can be read as saying that if the prerequisite conditions are true on entry to that step, and then the invariant conditions will hold during the entire

execution of the step and the post-conditions will hold on exit.

Each plan step has a set of input ports through which it accesses all data upon which it operates and a set of output ports through which flow all data that the plan step creates or modifies. Type restrictions on the ports are expressed as by the step's prerequisite and post conditions as are structural constraints between the data items. Overall structural constraints are expressed by the plan step's invariant conditions.

Plan steps are connected by abstract dataflow and controlflow links. A dataflow link connects the output port of one plan step to the input port of another and specifies that the datum produced at the output port of the first will eventually arrive at the second. These are abstract links in that they do not prescribe how the data moves; a function call can implement a dataflow links, but so can the act of storing a datum in some area commonly accessible to the two plan steps. Similarly, control flow links are taken as abstract constraints on program execution; the plan step at the input side of a control flow link must precede the step at the output side, but how that constraint is imposed is again a degree of freedom left for later binding.

Plan steps can be grouped into a single larger step that functions as a component one level higher in the hierarchy. Dataflow links connect the input ports of this surrounding plan step to the input ports of the initial steps in its subplan; dataflow links also connect the output ports of terminal steps of the subplan to output ports of the surrounding plan.

Finally, plans contain branch nodes that split control flow; and join nodes that reunite the computation after a condition branch. A branch node contains a set of branches each of which has a precondition. That branch whose precondition is satisfied is executed. Controlflow links connect branches to other plan steps and a controlflow link becomes active when its input plan step completes execution. A plan step is executable when the data for all of its input ports has become available and when all of its incoming controlflow links are active. Once the plan step has been executed, data moves from its output ports along the dataflow links to ports of other plan steps. Join nodes have a several join segments, each with a set of input ports; the join node has a single set of output ports with the same names as those of the segments. The intent is that data arrives at a single join segment, representing the termination of a path begun at a branch, and then flows through the join to the output port, enabling downstream plan steps to execute. Any execution order (including parallel execution) that is consistent with the above constraints is a legal ordering. Plans can be viewed as Petri nets, or abstract dataflow machines.

A critical part of the discipline of using plan based structures is to preserve the intended spirit. Plans are abstract, design-level structures; each plan step should

have a clear specification in terms of its prerequisite, post and invariant conditions. Other annotations, for example more complex timing constraints, can be layered over this basic framework, as long as the spirit is preserved.

A plan also contain links that connect the specification conditions (i.e. the prerequisite, invariant and post conditions) of its substeps. A *prerequisite link* terminates at a prerequisite condition of a plan step and originates at a set of postconditions of prior plan steps; it provides a deductive trace demonstrating that the precondition of the output plan step will be satisfied as long as the postconditions of the indicated upstream steps were established. A *main step* link terminates at a post condition of a plan and originates at a set of postconditions of substeps of that plan; it contains a deductive trace demonstrating the postcondition of the containing plan will hold as long as the postconditions of the indicated substeps hold. There are also *initial condition* links that terminate at the the preconditions of a substep of the plan and originate at a set of preconditions of the constraining plan.

Collectively we refer to these links as *purpose links*. The set of purpose links explain that logic of the plan: as long as the plan's preconditions hold, we can show that the preconditions of the initial substeps will hold. But then the invariant and postconditions of these substeps must hold as well. These in turn imply that the preconditions of substeps downstream will hold, justifying their invariant conditions and postconditions. Ultimately, the postconditions of the final plan steps justify the postconditions of the overall plan. These links can be thought of as the trace of a deductive planner that derived the plan given a description of the initial and goal state. Thus, a plan states the promise that a certain goal can be achieved, as long as a given set of preconditions are met; it also explains the purpose of each plan step by showing how it contributes to this mapping between input and output state.

There are several ways in which plan-based abstractions can be used in programs and these have been explored in several different research projects:

- A plan like representation can be used as an **executable specification language**. Brian William's RMPL [10] language is an example of this approach. In RMPL based systems, there are two major components, a control program (expressed in RMPL) that specifies a sequence of partially ordered sub-goals to be accomplished. RMPL variables represent a high-level, abstract view of the state of the underlying system that the program controls. The control program expresses its control actions, by setting these abstract state variables to new values. The second component of the program is a deductive controller, together with a model of the device being controlled. The deductive controller repeatedly uses its actual sensors, uses its deductive capabilities to abstract the sensor readings, and compares this current abstract state with that specified

by the control program. If they disagree, then the deductive controller uses its plant model to device a sequence of concrete control actions that will bring the abstract state of the system into accord with that specified in the control program.

- A plan like representation can be used as a **hybrid computational model**. In this case, the higher level structure of the program is expressed in a plan language. Plan steps can be decomposed into further plans, or into *primitive plan steps* that are expressed as procedures in an Object Oriented language such as CLOS. In this case, the higher level plan structures provide the benefits that result from an explicit representation of the purposeful relationships between program steps, at the cost of interpretive overhead. The lower levels (which is where most of the computational effort is spent) is executed efficiently, but with without the explicit representation of these relationships. The boundary is at the programmer's discretion and can be made dynamically. This was explored in the Programmer's Apprentice [6].

- A plan like representation can be used as an **active monitor** of an otherwise normally coded program. Here, the plan representation is run in parallel with normal program; however, wrappers are used to intercept some set of events in the program's execution trace. These are abstracted and compared with the set of events predicted by the plan representation. When they diverge, then we know that either the program or the plan is in error. If the plan is assumed to be correct, then the program is halted, diagnostic activity is invoked to figure out why the program failed and recovery actions are invoked to try to accomplish the overall goal. Alternatively, during program development, a divergence between the plan an the program's execution trace may be taken to indicate that the plan is in error and in need of revision. This idea is presented in [8]

A key feature of each of these approaches is that during the program's execution we have an explicit representation of a decomposition that matches the programmer's design of the computation. The module boundaries of a plan represent a more abstract and more meaningful decomposition than that provided by procedure boundaries or other constructs provided by the programming language. Each step of the plan has a clear specification and a clear purpose.

A Plan specified a method for achieving a goal given that certain prerequisite condition hold. There may, of course, be many such methods. However, each of these methods will do the task in a different way, leading to different qualities of

service. A substep in a plan contains a set up prerequisite and post conditions; these may be interpreted as a statement of a goal, and a request to find a subplan capable of achieving that goal given the prerequisite conditions. Any plan will do, but some specific plan will provide qualities of service that are much more useful in the current context. Everything else being equal, we would want to use that plan; however, that plan may require resources that aren't available or that are very expensive, so we should actually chose that plan that delivers the best tradeoff between cost and benefit, which is to say that **decision theory** governs the selection of a plan to achieve a goal. This leads to an analogy to object oriented programming. At the OOP level we have generic functions, alternative methods and type signatures guiding the dispatch; at the plan level we have goals, alternative plans, and decision theory guiding the dispatch. This approach has been investigated in the Oasis program [8, 4].

The decision theoretic approach easily generalizes to situations in which components of the system may fail, either due to natural causes, bugs or intentional attack. To do this, one builds a trust model, that indicates how likely each computational resource is to fail (and more specifically, how likely it is to fail in each of several specific ways). When examining which method to select, one then uses the trust model to calculate the expected value that would be delivered by a particular method using a particular set of computational resources. The expected value is the value of the quality of service that would be delivered if everything works multiplied by the probability that everything will work. In addition, we add in the cost of failing (i.e. a method that fails may cause harm in addition to not doing what we wanted it to do) multiplied by the probability of failure. We then select a method that maximizes the expected value. As the system executes, it monitors its own behavior and diagnoses its misbehaviors, it then updates the trust model to reflect the information gathered in diagnosis. This modification of the trust model makes certain methods less desirable than before and others more so; therefore, the system adapts to the change of trustworthiness, steering clear of methods that use suspect resources.

10.2 Use of Machine Learning

Execution at the plan level establishes a semantically meaningful context within which to apply statistical machine learning techniques. For example, in the previous section we talked about subplan selection as a decision theoretic problem in which we choose that method that is most likely to provide a useful quality of service at the lowest cost. Making this decision presupposes that we already know the quality of service that a method will produce. Alternatively, we could gather statistics as

we go, and attempt to learn a model of the method's execution profile over time. For a while, the system might behave suboptimally, but eventually it will become informed about how alternative methods behave and begin to make more accurate choices.

These gathered statistics can also be taken as a profile of normal behavior for a particular method. Behavior that falls far outside of this envelope can be regarded as anomalous, and possibly symptomatic of malfunction. However, in contrast to other statistically driven anomaly detectors, this approach uses the plan structure as a semantically meaningful backbone around which to organize these statistics.

11 Summary

We have that increasing levels of abstraction can form unifying frameworks within which we may explore issues of transparency and controllability of a system. The bulk of this report highlights the distinction between the Object abstraction and the raw bits and bytes that form the core of modern mainstream systems. The former is more compact and elegant, because it shares useful capabilities without artificial barriers. It is also potentially significantly more secure since it is not vulnerable to most of the basic techniques used to attack current systems. In addition, the object abstraction forms a natural core around which to build elegant access control models.

At an even more abstract level we can begin to imagine plan-level computing. We outlined three different ways in which plan-based computing could play a role: as an executable, model-based specification language, as a hybrid programming medium, or as an active monitor of normal code. The plan level introduces a yet more advanced notion of method selection, that of decision theory. The decision theoretic framework is particularly attractive because it is capable of making dynamic decisions about the trustworthiness of its computational resources allowing it to adapt to failures, bugs and attacks.

We began by considering the total cost of ownership of a system, pointing out that the operational costs involve diseconomies of scale. Thus, increasingly it is not the cost of producing software or even the much larger cost of maintaining a software system, but the actual cost of operating it that is now dominant. Most of this operational cost is the personnel cost of system administrators (both those who explicitly hold the title and the rest of us who spend an inordinate amount of our time doing system administration tasks).

System administrators are not programmers. That is, they don't use traditional programming languages to write systems programs or applications, the tasks we normally associate with programmers. They do need the behavior of their systems

to be transparent and they need to be able to control their system in an intuitive manner. Since they are not programmers, they can't use a programmers tools to get an effective view of the operation of their systems.

However, it should also be noted that although sysadmins are not programmers, they spend much of their time programming. They do this using truly terrible tools; sysadmins spend most of their time, writing *ad hoc* programs in scripting languages that are nearly incomprehensible. Worse yet, their task forces them to use not just one, but many such scripting tools, since increasingly every application comes with its own. Many of these small scripts become embedded in the system (they are prototypes that become products) and they then are added to the maintenance burden. This is a reenactment of the underlying system structure; it's "raw seething bits" all over again.

There is no good reason for any of this. It appears that both the object abstraction and the plan-level abstraction offer enormous leverage. A good scripting language can exist at the OO level. In fact, there are several excellent scripting tools based around Scheme, a minimal but elegant Lisp language. A programming environment for such a scripting system would be as rich as that for an object oriented implementation language because they would be the *same* language and environment tailored to suit the needs of two slightly different communities (system implementors and system administrators). As we argued earlier, the object abstraction naturally leads to direct manipulation tools that can present a more natural interface, because the objects that form the "basic stuff" of the system are direct analogues to the objects that system designers and system operators think about. Rather than attempting to sort out problems using file systems and registry data bases, sysadmins should be using powerful object browsers, that can give a comprehensible view of the current state of a system, and which provide comprehensible navigation tools. That is the promise of object and plan abstraction.

References

[1] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[2] C. Green. Application of theorem proving to problem solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 67–87. Kaufmann, San Mateo, CA, 1990.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[4] Michael McGeachie and Jon Doyle. Efficient utility functions for ceteris paribus preferences. In *Eighteenth national conference on Artificial intelligence*, pages 279–284. American Association for Artificial Intelligence, 2002.

[5] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of ACM Conference on Lisp and Functional Programming)*, pages 235–246. ACM, 1984.

[6] C. Rich and H. E. Shrobe. Initial report on a lisp programmer's apprentice. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 443–463. McGraw-Hill, New York, 1984.

[7] E. D. Sacerdoti. The nonlinear nature of plans. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 162–170. Kaufmann, San Mateo, CA, 1990.

[8] Howard Shrobe. Model-based diagnosis for information survivability. In Robert Laddaga, Paul Robertson, and Howard Shrobe, editors, *Self-Adaptive Software*. Springer-Verlag, 2001.

[9] D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. An object-oriented database system to support an integrated programming environment. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI Design.*, pages 117–129. Prentice Hall, 1991.

[10] B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 9(1):212–237, 2003.