DIAMONDS: Principles and Philosophy*

Umesh Bellur   Gary Craig

Kevin Shank

Syracuse University, Dept. of Electrical and Computer Engineering

Douglas Lea

SUNY Oswego, Dept. of Computer Science

CASE Center Technical Report # 9313

June 1993

Abstract

This report describes the design principles for DIAMONDS, an object-oriented software environment in support of heterogeneous distributed systems. DIAMONDS is a core project of the CASE Center's Object-Oriented Methods and Tools (OOMT) research effort. It is organized as a set of interrelated *proof-of-concept* prototypes and underlying research issues. DIAMONDS concentrates on a sound distributed software design model with strong statically analyzable semantics. This development methodology will be supported by both efficient run-time processing and dynamic resource management. It is also the intent of the research to provide tools in support of distributed software development.

*This work was supported in part by Rome Laboratory Contract number F-30602-93-C0108. A version of this report has been submitted to IWOOOS-93.

1 Motivation

The DIAMONDS project at Syracuse University is dedicated to the design of an environment for the development of object-oriented distributed systems software. This is first and foremost a software development methodology effort. Heterogeneous distributed systems present significant challenges to both existing software design methods and supporting infrastructure (run-time support services). The two major driving forces within the DIAMONDS project are:

1. the specification and validation of a sound object-oriented software development methodology for heterogeneous distributed processing, and
2. the development and support of an object (software) composition model which permits a system to effectively exploit an application's inherent concurrency with efficient communication and management overhead.

A distributed system is comprised of networks of heterogeneous processors which serve as a pool of computational resources. Distributed software systems are difficult to specify, design, implement and maintain. This task is made more difficult when the target architecture is highly varied or constantly changing, as is the case for heterogeneous distributed systems.

The overall goal of DIAMONDS is to produce a methodology and supporting infrastructure such that the mapping of an application to system resources dynamically reflects algorithmic parallelism, computational needs, and the current state of the system. In such an environment, application subtasks would be dynamically mapped to system resources based on the application's current state of computation and on the current resource availability.

The philosophy for accomplishing the goals of DIAMONDS is to construct a development model, a set of tools, and a run-time infrastructure in unison. Without the unique interplay between the components, the opportunity for success is greatly diminished. Thus, DIAMONDS differs from the many related projects which deal solely with one or another of these pieces[4, 7, 10, 8]. DIAMONDS seeks to improve the ability to produce effective distributed software.

Quality is enhanced by a development methodology providing a smooth transition path from analysis through to implementation. Resource consumption and performance issues are addressed by a comprehensive object composition model and a well matched infrastructure.

Central to this effort is a design model in which software is constructed out of fine-grained active objects and their interactions. These objects represent autonomous computational entities and thus provide for a high degree of explicit concurrency. This model supports a development process which is characterized by its continuity; providing a smooth transition path from abstract analysis and design to concrete implementation[5]. Current efforts make use of a corresponding concrete design notation, `odl`[9].

Although a high degree of concurrency is *possible* in systems with large numbers of active agents, in practice many of these objects will exhibit tight coupling with a subset of their collaborators. More importantly, a system composed of a vast sea of objects, each in its own process does not lend itself to efficient execution. Management and communication costs would be overwhelming.

The path to a solution is to combine objects into larger entities, while preserving all applications-level semantics. Methods, mechanisms and tools revolving around object composition, while maintaining a desirable separation of concerns across development and execution, form the principle glue of the DIAMONDS system:

1. OO design methods are intrinsically compositional at the functional level: Complex objects are logically composed via links to simpler objects.
2. Static analysis of a design introduces resource-based object composition. This analysis helps translate a sea of objects into a set of manageable *object clusters*, each containing possibly many design-level objects. Static analysis identifies highly probable patterns of object interactions by (among other analyses) distinguishing between asynchronous and synchronous object communication. Synchronously coupled objects are often placed in

a common cluster The collection of objects within a cluster exhibit little concurrency and can benefit from highly optimized communication protocols. In addition, the run-time need deal with fewer and larger clusters than application-level objects.

3. The run-time system maps and remaps object clusters onto yet larger units, *ensembles*. Ensembles are unique address spaces (protection domains) dynamically managed to exploit physical system resources.

The infrastructure seeks to maximize performance through optimization of the code associated with a cluster. The clusters are directly supported by lightweight control agents in the run-time system. A resource management facility enforces systemwide policies and establishes (on a dynamic basis) the number of ensembles which are allocated to an application. It also assists the run-time in placing clusters within the set of ensembles. The placement is also reconfigurable to handle both the changing needs of the application as well as the changing state (resource availability) of the distributed system.

Another way of looking at this separation is in terms of exposing and exploiting application concurrency, via:

1. explicit expression, using concurrent constructs in the design notation or programming language,
2. implicit expression, detected by static analysis, and
3. monitoring and profiling run-time behavior.

The remainder of this report provides details of the various aspects of the DIAMONDS project. Section 2 describes the main features of the underlying development methodology. Section 3 discusses the composition of objects into clusters. Section 4 describes the execution environment. We conclude with a brief report of current progress.

2 Design Methodology

The object-oriented paradigm is evolving from a set of programming techniques to a mature software engineering methodology. A range of OO software development methods represent variations on a common OO systems development paradigm, that may be briefly and informally characterized as follows:

1. Discover some tentative objects needed to fulfill functionality requirements.
2. Model these objects (and/or others discovered in the course of elaboration) and their interactions by assuming that *all* of them are autonomous computational entities (i.e., "active objects").
3. Develop computational designs logically behaving in accord with these models. (There may be almost arbitrarily small differences between this and the previous step.)
4. Map this design onto available physical configurations, software services, tools, and languages.

This model is implicit or explicit in many formal and informal OO system development methods and practices. For one informal example, groups of OO designers using CRC cards actively "role-play" instances of classes while discovering their responsibilities and collaborators[1]. Such development methods stress autonomy and tend to discourage premature commitment to task-specific control flow. Other notable consequences of this development model include:

- Systems are logically *composed* of many interacting active agents, rather than decomposed into task-oriented processes.
- Design-level objects come in all shapes and sizes, ranging conceptually from tiny (e.g., Integers) to huge (e.g., Banks). However, even "huge" objects tend to be representationally small, since they obtain most functionality by delegating actions to other potentially autonomous objects.
- The decision of whether particular objects are to be implemented in an "active" versus "passive" manner is (or would be, given appropriate sup-

port) a relatively late development issue. At the level of software analysis and design, there are often no meaningful intrinsic differences between objects that may be implemented "passively" versus those that may be implemented in stand-alone processes.

- Protocols involving event propagation, callbacks, multicast, and explicit routing are natural components of many models and designs. For example, model-view-controller style change notifications are usually best construed as asynchronous one-way notifications.
- Typical applications include more state-dependent and state-modifying operations than stateless services.

These features in turn stem from reliance on object models with the following properties, applicable to *any* object, from a simple Boolean, to a BankAccount, to a TextEditor:

1. Every object has a unique identity.
2. Every object is an instance of some class. Classes may be defined as subclasses of others.
3. Instances are constructed and maintained via generator objects.
4. Static object features are defined via attributes, relations, and constraints.
5. Every object is defined as a sequential computational agent, performing at most one operation at any given time.
6. All visible state transitions and operations are defined as *atomic* with respect to external observers.
7. Operations are *guarded* with state-based constraints. Objects are normally defined to contain queues to buffer pending untriggerable messages.
8. All object interaction occurs via one-way (asynchronous) and/or bidirectional (procedural) message passing.

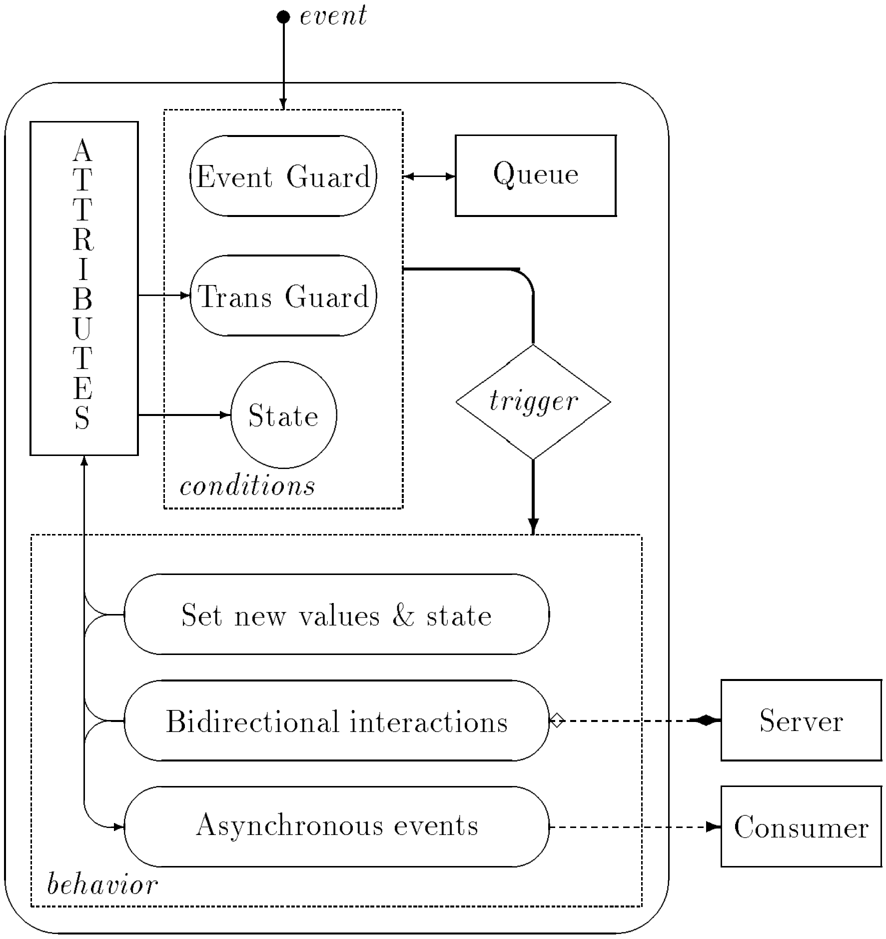

Fig. 1. Basic active object.

Fig. 1 illustrates the fundamental properties of an active object. It has a message handling/dispatch unit, local state, and actions comprising only of changes to local attributes, the raising of additional events, and, for certain objects (unshown here) the construction of new objects.

The resulting model applies independently of granularity issues. "Active" mechanics are sometimes needed even for "small" objects, for example, CountingSemaphores versus other Counters. However, efficient support for fine-grained objects is an obvious problem on most existing systems. It is simply inconceivable to place *every* object in its own process. The resulting overhead would not be limited to process management. It also includes maintaining information associated with access control, mobility and persistence, as well as name management and process-level garbage collection.

3 Clustering

We are investigating prospects for semi-automating several aspects of "physical" OO design under this general model; i.e., the mapping from logical objects to nodes, processes, and address spaces. Our approach centers on the *clustering* of objects into coarser granularity run-time entities that are better supported by contemporary hardware and operating systems. The fundamental notion is very simple: Fine-grained active objects are too small and numerous to be managed efficiently (or at all) as self-contained processes on contemporary distributed architectures. By grouping objects together into clusters, applications may be efficiently correctly supported and managed without altering their semantics.

This approach is guided by a simple observation on the computational model underlying typical active object frameworks: Any given active object or set of active objects may be *simulated* by a process-level "universal active object". (Such objects are well-disguised forms of Universal Turing Machines.) In fact, there is a wide range of intermediate forms that may be exploited when packing design-level objects into clusters.

Successful clustering provides a number of other performance benefits, including:

1. **Structuring Concurrency:** Objects that can perform activities in parallel can be contralocated (placed on different clusters). This allows us to exploit concurrency in a structured manner.

2. **Optimizing communication:** Objects which display a high degree of synchronous interaction can be colocated by placing them in the same cluster. Further, a run-time system may choose to place interacting clusters within a single address space. This in conjunction with late binding of communication mechanisms can avoid the overhead incurred by the

general use of heavyweight distributed communication.

3. **Better Virtual Memory Performance:** When cluster sizes and placements are guided by virtual memory parameters, the locality patterns that cause objects to be placed in a common cluster can lead to superior virtual memory performance.

4. **Better Persistence Performance:** A cluster may serve as the unit of persistence. The state of a cluster of objects may be saved and restored as a unit, further exploiting the locality of reference inherent in the application.

5. **Lower Management Costs:** The management costs associated with address space management, context switching, etc., of many small clusters may be balanced with resource requirements, concurrency losses, etc., stemming from the use of fewer very large clusters. These trade-offs may be applied in a system-dependent manner.

6. **Simpler Resource Mapping:** A set of objects which require specific resources may be bound as a unit for mobility and placed on any node meeting the specifications.

A fully automated clustering tool (or compiler for an active-object based OO language) would determine the best clustering and physical placement for every design-level object generated by the running system. (Because all systems involve many situation-dependent mappings, we do not take full automation seriously, but will describe it as an ideal.) Manual or automatic clustering may be employed in any context in which:

- There are a (fixed or variable) number of (active) *cluster objects*, each of which represent processes capable of simulating passivized objects.
- The "program" for the system contains a number of object constructions ("`new X`") to be executed at run time.

The goal of clustering is for a developer, tool, or compiler to arrange that object constructions are translated into a form that speeds execution of the application by removing, simplifying, and/or improving the accuracy of expensive run-time decisions about when, where, and how to construct clusters and objects.

We are attempting to automate the analyses that provide this clustering information. Despite its difficulties and imperfections, the use of static information appears to be among the few serious options for obtaining reasonable performance in distributed OO applications. When static clustering suffices, message protocols may be substantially weakened and optimized. For example, messages among two nonmobile statically colocated objects may even be inlined, while those among others may undergo several layers of communication protocols.

3.1 Cluster Formation in DIAMONDS

Cluster formation within the DIAMONDS environment is based on static analysis of the source code. This analysis detect the underlying concurrency and communication patterns among classes and objects. Our cluster formation model, like any other, must grapple with two difficult issues:

Granularity: Since the need for clusters arises out a need to bridge the granularity gap, granularity and placement issues must be free parameters. What is the appropriate size for a cluster? Is cluster size architecture dependent? Large cluster sizes could result in high cost of swapping or mapping clusters in and out of virtual memory as well as high cost of mobility (migration) while small cluster sizes could result in a number of "cluster faults" as well as high cross context communication costs.

Metrics: There may be no perfect metric that predicts the exact execution costs associated with any placement. This is compounded by the fact that affinities between any two particular dynamically constructed objects are often impossible to determine. However, a number of approximations may be gleaned from relatively inexpensive static analysis of the classes present in a design. Regardless of their nature, these metrics result in estimated *affinities* between objects, indicating whether they should be co- or contra-located.

Cluster affinities among objects comprise a weighted graph in which nodes represent objects and the arc weights represent the *affinity* between the objects they connect. (Equivalently, this graph might be represented as a symmetric $N \times N$ matrix of affinity factors.) Affinity factors represent the "desire" between two objects to reside in the same cluster. We use a model of *affinity* factors instead of dissimilarity indices since we start by assuming that in an ideal world (unconstrained by process management overhead, communication latencies, etc.) every object would be isolated into its own cluster. Hence clustering can be thought of as a means of composition rather than partitioning. For use in clustering algorithms, affinity factors should be well-behaved. Affinities should be represented on a continuous scale with a fixed (zero) value representing the total lack of affinity, and $A_{ij} \leadsto \infty$ representing strongest support for colocation.

The primary quantity that metrics should estimate is the degree of potential concurrency (or rather, its inverse) among objects. It is normally impossible to obtain perfect information about concurrency, mainly because:

- Most available information is about *classes*, not individual objects.
- Many potential uses of class information are themselves uncertain. A reference to an object of one nominal class may refer to an instance of any subclass.
- Most available information is *static*, describing potentials across the lifetimes of objects, not particular traces.

These are the same kinds of uncertainties facing, for example, developers of optimizing compilers for sequential object-oriented programming languages where, despite initially insurmountable-looking problems, much progress has been made (e.g., [3] and Section 4.2.1).

The general form of our model is one of increasingly refined affinity estimates. Readily measurable static class-based metrics are used to provide initial estimates. Metrics are designed to be *additive* through subclassing. These metrics may be in turn be refined to include information about individual objects and their dynamics. Simple static class-based measures may be expected to distinguish the large number of placements in typical programs in which objects should be routinely colocated. For example, objects of classes used solely as helpers via synchronous communication and whose identities are never otherwise exported should have extremely high affinities with their clients. Such cases are readily captured by metrics currently in development[2].

3.2 Using Affinity Information

An affinity matrix ideally specifies the attraction between any two objects in an application. From this, the system must estimate the best course of action for placing a `new` object given the current resource state of the application; i.e., the number, location, and contents of existing clusters.

In the simplest possible situation, these decisions may be fully static. When the exact numbers and properties of all objects are known, a static tool may provide *absolute* placement guidance. It may convert all application-level statements of the form

```
new X,
```

to the form

```
new X in C,
```

where `C` is a cluster. The run-time system may interpret these by constructing cluster `C` if it does not already exist, and then locating the object within it.

Some degree of static placement is often possible in most systems. However, in more realistic situations, a static tool will not know the current resource state of an application when a given `new` command is encountered. A tool may often still provide *relative* placement guidance, translating the `new X` to the form

```
new X near y,
```

where `y` is another object. For example, `y` might be the object that made the call to `new`. The run-time system will need to locate `y` and then, if possible, add the new object to `y`'s cluster. In the common case where `y` is indeed the object requesting the construction, this may be further optimized to indicate that the object should be constructed in the current cluster.

In other cases, a tool will not be able to provide any pre-execution guidance. This occurs, for example, when no affinity dominates any other. Here, the run-time system will need to make the choice itself. However, as outlined in Section 4.4, the affinities provided by static analysis may be used in conjunction with execution statistics collected by resource managers to aid these decisions.

4 Infrastructure

The composition model in combination with the emphasis of late-binding to a target architecture and the desired support for a *heterogeneous* distributed system, places significant requirements on the run-time system. The run-time must provide direct support (in the form of light-weight address spaces) for object clusters. It has ultimate responsibility for the construction, placement, mapping, and maintenance of object clusters.

The DIAMONDS run-time model extends our basic composition model by composing collections of object clusters into individual address spaces termed *ensembles*. This enables clusters be large enough to be efficiently managed while at the same time be small enough to be able to exploit the concurrency of the underlying computational substrate.

We are designing and building a prototype run-time system for DIAMONDS. For verification purposes the prototype must enable the demonstration of the key capabilities and unique aspects of DIAMONDS while also serving as a platform to study various subsystem designs, including:

- General support for distribution,
- Support for clusters,

- A proxy service and mechanism with strong support for cluster migration,
- Support for heterogeneity,
- Interfaces for various servers such as:
  - Resource Management
  - Deferred Compilation/Optimization
  - Garbage Collection
  - Persistent Storage

The current prototype effort is being built on top of a Mach-based microkernel with a Unix Server for use as a development environment. We intend the microkernel to provide starting points for:

1. scheduling of DIAMOND objects
2. inter-object communication (via Mach IPC)
3. resource management support (notably Virtual Memory)

In this setting, an ensemble is a Mach `Task`, and a cluster is represented by a Mach port (which is subsequently registered as a Network port).

4.1 Run-Time Abstractions

The central run-time abstraction is the cluster. It serves as the basic unit of mapping, persistence and execution.

An architectural overview of a cluster is shown in Fig. 4.1.

As shown in the picture a cluster consists of 3 basic components:

1. A message queue. The queue is divided into 2 subqueues, *active* and *pending*. The active queue contains messages to objects which are ready to run. That is, there are no *guards* (preconditions) that are unsatisfied. The pending queue holds messages to objects that have unfulfilled guard conditions.
2. A class directory, a form of a standard OO lookup table that allows code to be associated with an object.

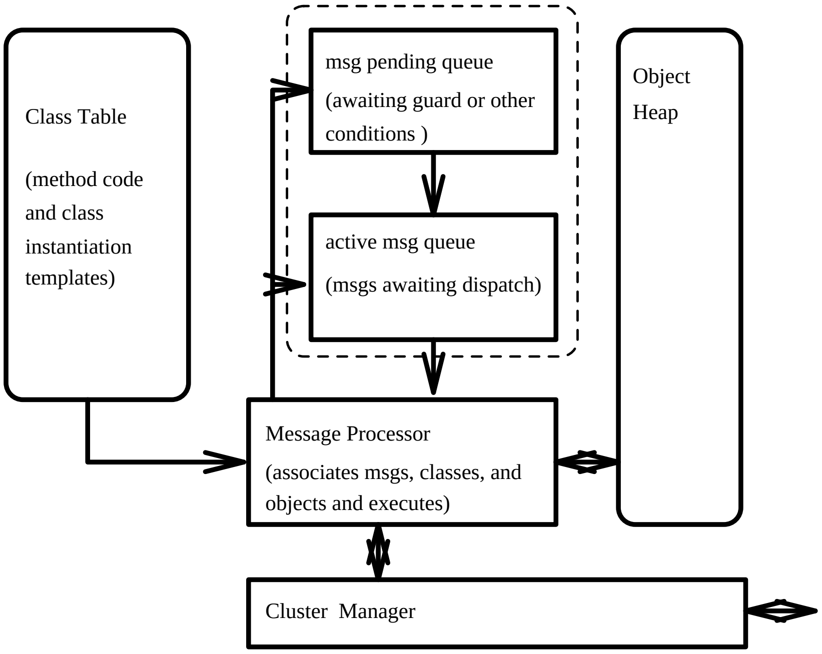

Fig. 2. Major Cluster Components.

3. An object heap, the memory holding objects that have been allocated within the cluster.

The other major components of a cluster are the execution agent and the cluster manager. The execution agent is the main computational thread within a cluster. The execution agent executes a loop including the following principal steps:

```
while not halted by the cluster manager do {
    pull top message off of active queue
    look up class and method code of destination object
    handle message parameters
    invoke method on object
    execute to completion, queuing all message sends
}
```

The cluster manager interacts with the outside world on behalf of the cluster. In this role it both accepts messages from others, and dispatches messages generated by cluster objects and destined for another cluster. It also maps all global object references to and from the local reference, creating local proxy objects when necessary. The manager participates in other activities such as garbage collection, persistence (check-pointing) etc.

4.2 Execution Model

We use a completely asynchronous communication protocol for all message interactions, including those that are viewed as "synchronous" at the language level. Here, an RPC or procedure style invocation is modeled as a one-way message send to the target object. The message contains information allowing the receiver to invoke a method on its sender in order to communicate replies. Execution within the invoking object is suspended until this "callback" is received.

Conceptually, this allows all methods to "run to completion". If an object issues an invocation that bears a reply, then this message terminates the current method, the object sits dormant until it receives the callback message. This message triggers a separate "continuation" method, which may then be run.

Our distributed execution environment is ultimately based on a pure implementation of this protocol. Prior to optimizations, all method invocations and replies, whether synchronous or asynchronous are performed via one-way message sends. This tactic may be seen as an automated version of continuation-based strategies used in Mach [6]. It is quite different than the more familiar stack based procedural invocation model. Because each object holds (most of) its state information itself, this model avoids common problems emerging from having state information scattered across machine dependent kernel level stack memory.

However, it is impractical and inefficient for each object to maintain its own *complete* execution state. We introduce `Context` objects to allow stack-like processing when it is more efficient and desirable. Per-method locals and control state information are allocated in user level `Context` objects. A `Context` object is indeed an object (although not a very interesting one) and can be passed as a reference from one cluster to another.

Method invocations normally create new `Context` objects. A `Context` contains enough state to be able to process one method invocation (although a single application-level method may in fact be split into several parts.). It does this by keeping track of the program counter, maintaining its own expression and parameter stack, and knowing which `Context` object to invoke when it is done.

As part of the Context creation process, any parameters being passed to the invoked method are extracted from the invoking message and pushed onto the new `Context`'s parameter stack. An execute message for this `Context` is then queued in the proper message queue. If the method called contains no callbacks (i.e., does not communicate back to its sender), the new `Context` is simply queued on the firable message queue. Otherwise, we include a reference to the current `Context` object in the invoked method's `Context`. In normal procedural interaction, this reference allows for a terminating callback when the method finishes. After completion, the execution agent picks some other `Context` to execute.

While a method is being processed, it may make asynchronous one-way sends and/or synchronous procedure style calls. In either case, the message is queued. However, if the object must block waiting for a reply, the context suspends and is later resumed when the callback message generated by the target is pulled from the message queue.

4.2.1 Code binding

The above execution model is taken as a basis for more aggressive performance enhancements. Our approach is centered on *deferred compilation.* By deferring compilation as late as possible, we allow for a blend of interpretation and execution of compiled code. It is in this manner that we can exploit the riches of a heterogeneous environment, while not sacrificing efficiency.

Basic object code is maintained in an interpretable byte coded format, in much the same manner as most Smalltalk systems. However, we also employ variants of tactics used in the Self compiler[3] to statically and dynamically transform byte codes to optimal machine code.

We preserve a transparent execution path from interpretation of raw byte codes to execution of machine specific object code. At different points along this path we interject specific optimizations that take into account:

- current cluster composition
- architectural features of available processors
- feedback from profiling information reflecting common execution paths.

4.3 Naming

Distributed execution relies on services maintaining object naming schemes and related inter-object communication protocols. The DIAMONDS architecture assumes that many objects will not be exported beyond their encapsulating subsystems. Such subsystems will usually be wholly contained within an object cluster. In some instances, the subsystem can uniquely be determined (is statically specified) as a single object cluster.

Exploitation of empirical locality of reference forms the basis for system support services of all sorts. Here, locality allows significant reductions in an application's global namespace. However, we would still like to allow *any* object reference to be exported implicitly and on demand, rather than require the user to explicitly export such object references. Exported object references must be location-transparent. Further, although the overall DIAMONDS architecture nearly eliminates the need for object (as opposed to cluster) migration, the naming scheme must not preclude it.

Object references are represented using object descriptors. Each cluster manages a corresponding object descriptor table (`ODT`). A global object name (reference) is comprised of a *cluster id*, ODT entry id (relative to cluster), and a class reference. Within application-level objects, references are represented using only the ODT entry ids. Remote references are represented via local proxies[11]. In addition to naming information, the ODT contains a flag indicating whether an entry is a local object or a proxy, a reference to the objects representation (if local), a reference to the object's dispatch table, and a reference count indicating the number of proxies currently in existence for the object.

4.4 Resource Management

System resource management is concerned with system-wide resource utilization and throughput as well as application response times. The DIAMONDS system resource manager (RM) allows transparent and efficient distribution via:

1. allocation and placement of *Ensembles* and
2. dynamic composition.

Heterogeneous distributed systems pose significant difficulties to resource management. In particular the notion of a processor pool available for scheduling, is not feasible given the disparity in machine binaries, architectures, and performance characteristics. The task is made more difficult by the lack of detailed information about the internal demands and needs of the applications. In DIAMONDS, support for heterogeneous processing will be available via generic virtual machine and intermediate code, as optimized though deferred compilation.

Clustering provides a basis for most other aspects of resource management. Clusters are inherently more manageable when it comes to contemplating the system-wide impact of a placement decision. Further, the clusters are of a reasonable size, both in complexity and processing activity, for migration. The DIAMONDS RM is optimized for cluster migration, rather than object migration.

The RM must monitor system resource status and the status of the various applications. This will be implemented as a hierarchy of interrelated services (e.g., for performance monitoring of physical resources).

The run-time infrastructure of individual *ensemble managers* also provides a significant source of information for the RM, Each ensemble manages statistics about its clusters and (cooperatively) its interaction with other ensembles. These statistics include:

- number and size of the clusters
- average number of dispatchable methods within each cluster
- rate of outgoing and incoming messages and their ratio.
- windowed reference counts of interactions between clusters.
- aggregate per-ensemble and inter-ensemble statistics

These run-time statistics may be used in order to guide:

- the placement of new objects within clusters
- the placement of clusters within ensembles
- the need to construct additional ensembles to satisfy an application's resource requirements
- the need to migrate a cluster across ensembles

Most of these decisions and actions are performed cooperatively by the RM and the run-time system. For example:

- The application's run-time system itself places objects within clusters. Some placements require the combination of resource statistics with initial static object affinity measures.

- The application may be able to detect the need for additional ensemble-level resources and make the appropriate request to the *ensemble allocator*.

- The application may detect a severe imbalance and send a request to the *migration service*. The cluster managers cooperate with the migration service by exporting a protected interface (and protocol) for suspending activity and logging state, and similarly for restoring state and resuming.

The RM may also perform reconfiguration in an application friendly way when resource constraints are imposed; e.g. change in resource policy, overload situations, and node and/or communication failures.

5 Summary and Current Status

Prototypes of clustering tools have been completed and are currently being validated via simulation studies. Early prototypes of the run-time and skeleton RM services are scheduled to be complete in the Fall of 1993. A more complete prototype of the DIAMONDS infrastructure is scheduled to be delivered to Rome Laboratory in late Spring 1994.

References

[1] K. Beck and W. Cunningham. A Laboratory for Object-Oriented Thinking. In *Proc. OOPSLA*, pages 1–10, 1989.

[2] U. Bellur. Static Clustering of Active Objects in Fine Grained Object Oriented Distributed Systems. Ph.D. Dissertation in preparation.

[3] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proc. OOPSLA '91*, pages 1–15, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[4] Arunodaya Chatterjee and William A. Rogers. Integrated Resouce Allocation in ESP. Technical Report ESL-ESP-194-92, MCC, October 1992.

[5] Dennis de Champeaux, Doug Lea, and Penelope Faure. *Object-Oriented Software Development*. Addison-Wesley, 1993.

[6] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. of 13th ACM Sym. on Operating Systems Prin.*, pages 122–36, October 1991.

[7] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic, and B Tangney. Dynamic Clustering in an Object-Oriented Distributed System. In *OOPSLA Workshop on Objects in Large Distributed Systems (OLDS-2)*, 1992.

[8] S. Krakowiak, A. Freyssinet, and S. Lacourte. A Generic Object-Oriented Virtual Machine. In *1991 International Workshop on Object Orientation in Operating Systems*, pages 73–77, 1991.

[9] D. Lea. ODL: Preliminary Language Report. Technical report, Syracuse University CASE Center, June 1993.

[10] R. Lea, P. Amaral, and C. Jacquemot. COOL-2: An Object Oriented Support Platform Built Above the Chorus Micro-Kernel. In *1991 International Workshop on Object Orientation in Operating Systems*, pages 68–72, 1991.

[11] Julien Maisonneuve, Marc Shapiro, and Pierre Collet. Implementing references as chains of links. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 236–234, Dourdan (France), October 1992.