MELDC: A Reflective Object-Oriented Coordination Language

Gail E. Kaiser[*] *Wenwey Hseush*[†] *James C. Lee*[‡]
Shyhtsun F. Wu[§] *Esther Woo* *Eric Hilsdale* *Scott Meyer*

Department of Computer Science
Columbia University
500 W. 120th Street
New York, NY 10027
Tel: 212-939-7081, Fax: 212-666-0140
{kaiser, hseush, wu}@cs.columbia.edu

Technical Report CUCS-001-93

January 15, 1993

Abstract

A coordination language, MELDC, for open systems programming is presented. MELDC is a C-based, concurrent, distributed object-oriented language built on a reflective architecture. Unlike other language research, the focus of MELDC is not only to study what specific language features should be designed for solving certain open system problems but also to provide programmers a high-level and efficient way to construct new features without modifying the language internals. The key to the reflective feature is the metaclass that supports *shadow* objects to implement *secondary behaviors* of objects. Thus, the behavior of an object can be extended by *dynamically composing* multiple secondary behaviors with the object's primary behavior defined in the class. In this paper, both the MELDC programming model and the reflective architecture are described. Then, we introduce the mechanism of *dynamic composition* as well as its application in building distributed and persistent systems. In particular, a soft real-time network management system, MELDNET, is built on top of MELDC to monitor the EtherNet performance. Finally, the current status of MELDC is given.

[*] Kaiser is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants from AT&T, BNR, Bull, DEC, IBM, Paramax and SRA, and by the New York State Center for Advanced Technology in Computers and Information Systems.

[†] Hseush is supported in part by the NSF Engineering Research Center for Telecommunications Research.

[‡] Lee is supported in part by the Center for Advanced Technology.

[§] Wu is supported in part by an IBM Fellowship and in part by the NSF Engineering Research Center for Telecommunications Research.

Contents

1 Introduction

The concept of "coordination language" was introduced by Carriero and Gelernter [CG89] to designate a class of programming languages suitable for describing the behavior of *open systems*. Ciancarini [Cia90] suggests the following definition for open systems:

> An open system is a dynamic set of agents both cooperating and conflicting for the use of a dynamic set of services and resources. The agents, the services and the resources are heterogeneous; they operate both in parallel and in concurrency; they communicate; they have some goals (what they would like to do), some duties (what they should do), some rights (what they may do), and some constraints (what they must not do).

The development of open systems in distributed computing is a result of using computer and network technologies in real-world human society. The complexity of open systems mirrors the complexity of human society. Coordination languages usually extend the declarations and statements of some base *computation* language, such as C and Pascal, with additional facilities to support distributed and/or parallel computation. Many coordination languages and models have been proposed for open systems programming. Among these approaches, the most popular one seems to be the concurrent object-oriented language approach, since it provides a natural environment for expressing concurrency and encapsulating distribution in objects and messages. Objects are naturally suited to represent real-world entities with private memory and predictable behavior, and messages are the communication media among objects. Most object-oriented coordination languages focus on providing immediate language features for building open systems and hard-code these features into the language internals. One example would be supporting atomic actions that guarantee serializability; however, it might then be difficult to build applications with concurrency-related correctness criteria other than serializability, such as epsilon-serializability [Pu91]. Other object-oriented languages support persistence, remoteness, monitoring, authorization, authentication, etc. as immediate language features.

Like other languages, the ultimate goal of the MELDC language is to support a wide range of high-level features for programmers to cope with the problems of implementing open systems. Unlike other language research, our focus is not to study what specific language features should be designed for solving certain open system problems, but to investigate the language architecture with which programmers are able to construct — without modifying the language internals — new features in a high-level and efficient way.

MELDC is a C-based, concurrent, distributed object-oriented language built on a *reflective* architecture. The core of the architecture is a micro-kernel (the MELDC kernel), which encapsulates a minimum set of entities that cannot be modeled as objects. All components outside of the kernel are implemented as objects in MELDC itself and are modularized in the MELDC libraries.

MELDC is reflective in three dimensions: structural, computational and architectural. The structural reflection indicates that classes and meta-classes are objects, which are written in MELDC. The computational reflection means that object behaviors can be computed and extended at runtime. The architectural reflection indicates that new features/properties (*e.g.*, persistency and remoteness) can be constructed in MELDC. These properties can be attached to and removed from objects at runtime. The reflective architecture provides high flexibility to customize or extend object behaviors in an elegant way. For example, a programmer builds a simple type of persistent objects that do not survive catastrophic system failures and then builds a comprehensive version of persistent objects that survive system failures by applying redundancy to the simple ones. In MELDC, persistency is not a language primitive, but just another property that can be constructed for objects. The semantics of persistency and policies to implement it are defined in MELDC.

Since micro-kernel facilities cannot be replaced or modified by the MELDC programmer, several common choices are supported by the kernel and can be designated by the programmer using compiler switches. For example, MELDC intends to support a variety of parallel and distributed applications that have different concurrency characteristics. Some applications require a small number of long-lived threads while others need a large number of short-lived threads that are created and destroyed dynamically. Thus MELDC provides three different thread packages (interleaving stack, one-stack-per-thread and heap-based) [HLK92], which can be chosen with a compiler switch. Other compiler options enable the programmer to choose pre-emptive versus non-pre-emptive schedulers and either merging or overriding behavior for multiple inheritance.

The MELD project has been one of the major foci of the Programming Systems Laboratory at Columbia University since 1987. The original design of the MELD language focused on supporting multiple programming paradigms at multiple levels of granularity. MELD integrates four paradigms: object-oriented, macro dataflow, module interconnection and transaction processing [KHPW90, KPHW89]. Starting in 1990, the MELD language was completely redesigned and reimplemented from scratch to produce MELDC, which is closer to C, has fewer but more sophisticated "features", and a cleaner architecture with many of the facilities implemented in the MELDC language itself.

2 The MELDC Object-Oriented Programming Model

A MELDC program consists of a collection of active objects, which send and receive messages to and from other objects (local or remote objects). An object is a runtime entity that has its own private data and control. The state (instance variables) and the behavior of an object are defined by a second program element known as its *class* in object-oriented programming. The class of an object states how the program behaves in reaction to different messages delivered to the object. A sequence of actions corresponding to a received message is encapsulated as a *method*. In C++, classes are not considered as objects at runtime. They are abstract data types and are defined statically. The creation of objects is "declared" through C++ statements and the behaviors of objects are embedded in the code generated by the compiler. In MELDC, we consider a class to be an object, which is defined by another object known as a *metaclass* (a class of classes) in object-oriented programming. Similar to Smalltalk, MELDC metaclasses are exposed to the programmer and classes are explicitly created as instances of a metaclass[1]. A default metaclass (called `Metaclass`) is created statically. Objects and classes other than the default metaclass are created at runtime by sending `Create` messages to the relvent metaclass. A MELDC program starts with the metaclass already created and a set of `Create` messages to be delivered. There are no senders for these initial messages. A subset of the messages are first delivered to the metaclass to create classes and then the rest of messages are delivered to these classes to create

[1] Similar to CLOS [DBM88] and ObjVlisp [Coi87], it is possible to have multiple metaclasses that are instances of the default metaclass. There are numerous complex issues regarding metaclasses, which are not within the scope of our discussion.

global objects.

One of the important concepts in MELDC is supporting dynamic extension of an object behavior based on the reflective architecture. The extended behavior of an object is referred to as its *secondary behavior* to distinguish it from the *primary behavior* defined in the class of the object. Extending object behavior in MELDC is characterized by two properties: (1) composability and (2) decomposability. Composability states that primary behavior, which implements the interface of objects, can be modified by composing with multiple secondary behaviors without changing the objects' interface. Decomposability describes the reverse property of composability. Primary behavior encompasses an object's functionality as defined in the object's class definition or provided through an inheritance mechanism. Secondary behavior on the other hand, encompasses dynamically added functionality which is in most cases orthogonal to primary behavior and to other secondary behaviors.

Not everything in MELDC is considered as an object — some program entities are not implemented as objects for reasons of efficiency. Most entities inherited from the C language are not objects: Examples are C statements, C basic types and variables, and C declarations. Some other program entities cannot be treated as objects under MELDC's non-static object-creation scheme, such as synchronization entities and threads. The *kernel* of our object-oriented model encapsulates those entities that are not considered as objects. Messages, for example, cannot be objects in the MELDC architecture, for if they were, the creation of a "message object" would require a second message `Create` to be sent to the metaclass. However, this second message can only be created by sending a third message to the metaclass, leading to an infinite regress.

3 MELDC **Reflective Architecture**

The key to designing reflective systems is *uniformity*. A language is considered "reflective" if it uses uniform structures to represent data as well as control entities (*e.g.*, programs) [IC88]. A reflective language provides the capability of computing control entities in the same way as computing data. For example, Lisp is a reflective language, because it uses the same structures (lists) for both data and programs. The meta-description of control entities (the control of control entities) are usually defined in the form of *interpreters*, which may use different internal structures from data and programs. An architecture for reflective languages is shown in Figure refreflection.arch.ps. Programs and data are represented in the same form. Programs are data to be processed by the interpreter. In this architecture, program units can be treated as data and computed by other program units. Some data that have been processed by some program components can be promoted as programs and processed by the interpreter.

In object-oriented programming, data and control (functions) are encapsulated in objects. Objects communicate with others through message passing to achieve some computations. Uniformity means that all entities in the system are treated as objects. However, in our programming model, some control entities such as messages and threads cannot be treated as objects. The MELDC kernel consists of all such control entities. All entities outside of the kernel are treated as objects and are implemented in MELDC. For example, the default metaclass code is written in MELDC and compilable by the MELDC compiler [2]. One of our goals is to study the fundamental

[2]Many object-oriented languages/systems claim full uniformity (treating all entities as objects). However, some of entities (*e.g.*, messages, threads, metaclasses) can only be considered as objects conceptually, not in the real

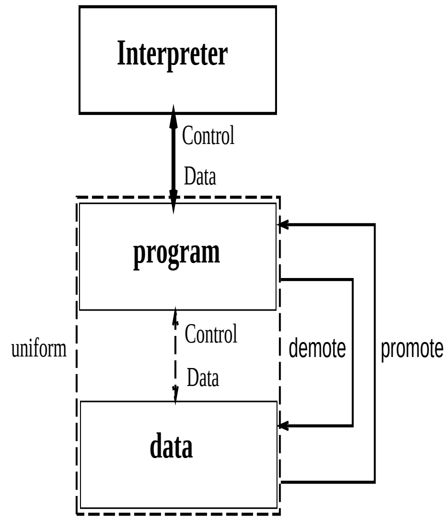

Figure 1: An Architecture for Reflective Languages

limits of what entities cannot be treated as objects and is not implementable in the language itself.

An object-oriented language is reflective if the behavior (computation) of a program can be computed, extended or reasoned about itself in the form of objects and messages. The key to the MELDC reflective feature is that the metaclass (see Figure refmeldcarch.ps) supports *shadow* objects (some sort of meta-objects) that implement *secondary* behaviors (some sort of meta-behaviors) of objects. The behavior of an object can be extended by dynamically composing multiple secondary behaviors with the primary behavior of an object that is defined by its class. This mechanism is referred to as *dynamic composition*. Shadow objects that implement the secondary behaviors are said to be attached to the object. The primary behavior of the object and its state can thus be computed or reasoned about by the attached shadow objects. The metaclass and the MELDC kernel form the foundation of the MELDC reflective architecture. Most other objects in our architecture are built on this reflective foundation through the mechanism of dynamic composition of object behaviors. A few objects such as a the system object (interface to Unix), shell objects (interface to shell environments) and memory objects, which support interfaces to the underlying system, do not rely on the reflective feature. Figure refmeldcarch.ps shows the architecture of the MELDC runtime system.

The MELDC runtime consists of a kernel (the MELDC kernel), a runtime object library (the metaclass, a system object and memory objects) and a common library. The MELDC kernel together with the runtime objects are necessary components for program execution. The MELDC kernel supports primitive functionalities for messages, threads, synchronization primitives and dynamic composition. The metaclass is the origin of a program execution, which interfaces with the kernel. Memory objects support memory utilization. The system object class which models the operating system underneath, is the next layer. A MELDC process is currently implemented as a UNIX process. If the underlying operating system changes, another corresponding system

implementation.

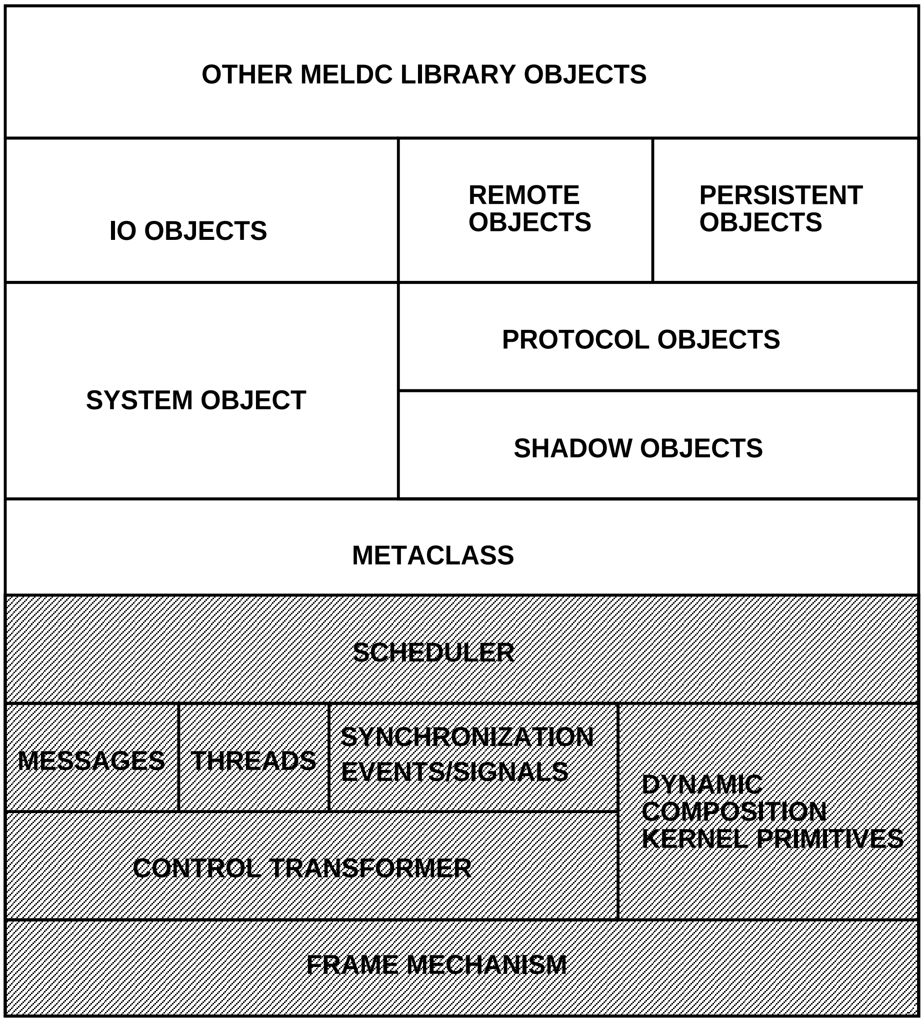

Figure 2: The MELDC Architecture

class can be used. The IO objects support a high-level IO interface. Remote objects and persistent objects support distribution and persistency. Objects in the common library (*e.g.*, monitoring objects, debugging objects or instrumenting objects) are constructed by dynamic composition on the MELDC reflective foundation. Our design of the architecture clearly reflects a system that supports uniformity, object uniformity as well as control uniformity, transparency, and orthogonality of a set of building block objects essential and sufficient to support the functionalities the MELDC language provides.

3.1 Dynamic Composition

Like the inheritance mechanism in object-oriented programming, dynamic composition is a mechanism to structure object behavior. Both inheritance and dynamic composition provide a high degree of reusability as well as extensibility. They allow programmers to design their programs in a hierarchical way and reuse components. The usefulness of inheritance has been demonstrated in many existing systems. However, there are a few limitations on the mechanism of inheritance. In most object-oriented languages, as in C++ and Smalltalk, inheritance is a predefined concept. By *predefined*, we mean that once the instantiation of the object has taken place, the behavior of the object is fixed throughout its life-time. Adding a new behavior to an object can be done only by destroying the old object and creating a new one with a new definition. This limitation makes any dynamic enhancement to a system painfully difficult. For example, adding a new protocol to an existing network object should not require a network shutdown or create any interference with the rest of the system. In the ideal case, modifying an object's behavior dynamically should not affect any other parts of the system.

Most inheritance mechanisms are implemented with the algebraic form of union: A complex class is constructed from a set of smaller class as united together, combining their data structures and methods. However, this type of inheritance mechanism is unable to modify the existing objects' behaviors. For example, if a programmer wants to trace an object, a common technique is to add a print statement to every method. There are many problems with this approach. Removing the modifications will be as difficult as adding them, and also extremely time consuming. Sometimes the number of modifications is very large; hence the likelihood of introducing new bugs is very high. MELDC supports both inheritance and dynamic composition.

Our idea of dynamic composition is inspired by *mixins* and method inheritance in Flavors [Moo86, Boo83, Hen86]. The mixins are ordinary classes, but their sole purpose is to extend the functionality of other classes. When a Flavors object is invoked, one or multiple methods can be selected from the set of inherited methods. The programmer can predefine the multiple methods' calling sequences. Analogously, our dynamic composition depends on ordinary classes, but whose sole purpose is to define instances that can be composed with other objects to extend their behavior. Dynamic composition consists of taking such a behavior object and attaching it to an ordinary object. The resulting object is effectively a cross-product of the two objects, but each individual object's identity is perfectly preserved. As in Flavors, multiple methods will be called when a message arrives at the object, but the method calling sequences is determined by the order in which the objects were combined. Unlike Flavors, the object composition is dynamic: the programmer can dynamically attach a "mixin" object to a regular object at run-time.

Composition thus provides a system with a high degree of reusability. Modifying a behavior no longer entails the modification of the program. Dynamic composition allows the programmer to dynamically enhance (add or eliminate any composed behavior at any time) a statically defined object; this cannot be achieved through any inheritance mechanism. Dynamic composition is a very powerful tool in implementing distributed applications. In the later sections, we demonstrate how dynamic composition can be used to reduce the complexity of implementing a large-scale distributed system.

Let's look at a simple example of dynamic composition. A class `Savings_Account` describes two methods `deposit` and `withdraw`. Depositing and withdrawal are the primary behaviors of every instance of `Savings_Account`. Yet a bank manager may decide to audit the activities of a particular savings account. To do so he needn't modify the definition of the class, he can simply attach a secondary behavior "audit" to the account object. The audit behavior does not affect the primary behaviors, `deposit` and `withdraw`, but reports those activities to the manager. The secondary behavior is simply a modifier to `deposit` and `withdraw`, so that the program behaviors are now auditive deposit and auditive withdraw. Attaching a secondary behavior simply modifies the primary behaviors in a transient and orthogonal fashion. The manager can remove the auditing from the object at any time.

The idea of reflection is one of the important concepts that allows a program to dynamically alter its own behavior [Mae87, Mae88]. Reflection has been implemented in many languages (e.g., ACTORS [Fer88], ABCL [Wat88], and ObjVlisp [Fer89]). The dynamic composition provided by our MELDC language is one of many forms of reflection. However, MELDC limits the power of reflection in order to improve the efficiency of the language. In most reflective architectures, method execution is performed by interpretation and objects are allowed to alter other or their own interpretation mechanisms. The interpretation introduces a performance penalty on all operations, even though most of the time, objects do not need to be reflective. In MELDC, only those objects that are composed trigger the reflection mechanism, so the performance of other objects is not degraded by the facility for dynamic composition.

3.2 Uniformity

In the MELDC programming model, it is impossible to treat every entity as an object. All entities that cannot be treated as objects are encapsulated in the kernel. MELDC has two types of uniformity, object uniformity and control uniformity. The entities outside the kernel follow object uniformity and the entities in the kernel follows control uniformity. Object uniformity means treating every entity as an object. Control uniformity means that all control entities can be transformed to other control entities.

In MELDC's runtime system, there are several entities that can not be modeled as objects.

- MELDC does not model a message as an object. A class is itself modeled as an object at runtime; it is an instance of the metaclass, which defines the method for object creation. To create an object, a `Create` message is sent to the class object. If a message is modeled as an object, then a recursion will lead to the consumption of all system resources. A message `m1` is sent to the class object to invoke the creation of an instance of the class, leading to the creation of the object that represents message `m1`, which requires a message `m2` to be sent to the message class object, leading to `m2's` creation ... – this process never terminates. While it is possible to model messages as objects if the system is a static system, where all messages are pre-created at compile time, MELDC is designed as a dynamic system, where the number of concurrent messages and threads cannot be determined at compile time. In such a system, treating classes as objects (adhering to the notion of metaclass) and treating messages as objects are mutually exclusive and contradictory.

- Resource management for threads can not be conceptualized by objects and message passing. If our resource manager, whose responsibility includes allocation of activation record (frames), an invocation of a method will cause a message to be sent to the resource manager object to trigger the allocation of frames. Again, this is another recursive process which continues forever.
- Low level synchronization primitives, such as the mechanism of locking objects and blocking threads cannot be implemented at the object level either. There is no language facility which can be used to lock the objects which performs synchronization.

With the above limitations, since *absolute* object uniformity is virtually unachievable, we believe an object-based architecture built on top of a small and efficient kernel is optimal. Thus our kernel implements only the necessary and fundamental notions of the system which cannot be modeled as objects is made as small as possible.

It is just as important to establish *control uniformity* as to achieve *object uniformity*. Without a well defined and uniform way of controlling how objects interact with each other, the system, as it grows in complexity and scale, will result in chaos. We have established that objects interact only through message passing, and for the flexibility and ease of programming, the MELDC programming language supports different kinds of messages:

- external messages for remote objects
- internal messages to invoke a method
- an event or a signal detected by the runtime system

The underlying control mechanism of all these seemingly different message passing schemes is structured and always follows a standard transformation path. Should different messages have their own unique control mechanisms, as the system grows in complexity, these control mechanisms may conflict with each other and produces unpredictable results in a heterogeneous environment. The MELDC kernel implements the transformation of messages and their control and thus supports control uniformity.

3.3 The MELDC Kernel

The MELDC kernel is shown in the shadowed area in Figure 2. In our model, internal messages, serving as the control entities (function calls) in object-oriented languages, external messages, serving as the medium in interprocess communication, events and signals, the medium for intraprocess communication, and threads are *isomorphic* manifesting themselves differently at the conceptual level to programmers. At the system level, these messages are ultimately transformed into internal messages which trigger the invocation of methods. For example, an external message from a remote object is transformed into an internal message when it arrives at a machine. An internal message is transformed into a thread when it arrives at an object.

One of the major tasks in the MELDC Kernel is to provide this service. Our system also distinguishes threads and messages from objects. The MELDC runtime system does not model a thread as an object. An internal message *metamorphoses* into a thread, which represents the execution of the method triggered by the message arriving at an object. A function call on an object is modeled as a sequence of four steps:

1. The calling thread is transformed into a message
2. When the message arrives at the object, the message is transformed into a thread, which executes the called function
3. When the function completes, the thread is transformed into a reply message, which carries the return value
4. When the reply message arrives at the calling object, it is transformed back to a thread, which continues the code after the call statement

Threads and messages are each identified by a globally unique id. Since the transformation is handled by the MELDC Kernel transparently, it provides an easy way to implement remote procedure calls as well as remote light-weight threads.

The integration of internal messages, external messages, events, signals and threads greatly facilitates building dynamic systems in a distributed environment. A dynamic system is characterized by the dynamic feature of programs, which preserves the compile-time information (*e.g.*, function/variables names and addresses) throughout the runtime execution, such that external objects (*e.g.*, the end-users) can dynamically bind program elements to certain properties. One notable example of dynamic binding is to allow end-users to dynamically invoke a method. At runtime, the end-user merely composes a message which carries information about the function name and the values of input parameters. The system also supports dynamic type checking since it is often necessary to ensure correct program behavior.

While some programming paradigms provide some sense of dynamic function invocation with the use of dispatch tables, which are to be filled in by programmers explicitly, MELDC provides a transparent format of dynamic binding. The programmers are not required to provide the compile-time information as part of their programs. When an external message, in a pre-defined format arrives at a process, the system readily transforms it into an internal message which triggers the execution of a method. Our notion of transparency between external messages and internal messages creates an illusion that objects live in a network without machine and process boundaries. An object can communicate with any other object, as long as it can name the remote object. This is possible since MELDC enforces a globally unique naming convention for objects. An end-user can pretend to be an object in the network, as long as he/she can simulate and interpret messages. A dynamic system built in MELDC can treat I/O devices, the X servers or other network entities as remote MELDC objects; this feature again illustrates our principle of object uniformity.

Our architecture defines clearly a transformation mechanism between different types of control entities, which include external messages, function calls, events, signals, and internal messages. We treat internal messages as the fundamental control entities in our system and all other forms of interaction between objects will ultimately be transformed into this format. This control uniformity greatly reduces the complexity needed in the Kernel to handle various types message passing. Moreover, this design avoids the pitfalls of incompatibility between different control mechanisms in large scale distributed systems. The behavior of the runtime system in handling message passing can easily be modified by revising the transformation mechanism component in the Kernel. Should a new message passing entity be needed, it can be implemented fairly easily; the Kernel merely needs to define a new transformation mechanism.

3.4 MELDC Thread Packages

Many applications like telecommunications, network management and real-time stock trading might have different requirements for thread policies. For example, an application who has few long-term threads might choose a thread policy that is not very efficient in thread creation, but a very efficient synchronous call mechanism. On the other hand, an application who has a lot of short lived threads, might want to have a thread allocation mechanism that is efficient on thread creation and use very little memory. A single implementation of the thread package might not satisfy the requirement of all the applications. The MELDC compiler provides a switch which allows programmer to choose from the three different thread implementations to suit their application. The interleaving thread, should be used by the application which has large number of short term threads. On the other hand, if the maximum size of the stack is well known, and applications do not require a large number of threads, the stack base approach is probably the best approach. Finally, if memory is scared, and the performance of thread creation is not critical, then the heap-base approach could be used.

3.5 Metaclass

In MELDC, `Metaclass` is the default class of classes, including itself. It is the first and the only object that exists when a program execution starts. `Metaclass` is a first-class object, which is also implemented in MELDC like any other classes. `Metaclass` has four functionalities:

- object creation and destruction
- dynamic composition
- class/object knowledge inquiry
- a generic interface to access “external” objects.

3.5.1 Object Creation and Destruction

`Metaclass` defines two methods `Create` and `Destroy` for classes to create and destroy objects. A message `Create` (or `Destroy`) sent to a class activates the object creation (or destruction) code defined in `Metaclass`.

```
obj = Class.Create(opt\_param1, opt\_parm2, ...);
Class.Destroy(obj, opt\_param1, opt\_parm2, ...);
```

There are two important issues, object initialization/termination and object management. Like the constructors and destructors in C++, MELDC allows programmers to specify initialization (in the `init` method) and termination (in the `term` method) for objects. When an object is being created, `Metaclass` sends an *asynchronous* `init` message to the object. When an object is being destroyed, `Metaclass` sends a *synchronous* `term` message to the object. The parameters (*i.e., opt_param1, opt_param2, ...*) passed to the `Create/Destroy` are passed to the `init/term` method. In C++, the calls to constructors and destructors are generated at compile time, since classes in C++ are purely compile-time entities. In MELDC, a class is a runtime object, which can be constructed at any time during execution[3]. The MELDC compiler cannot tell that whether a dynamically created object is a class or not, and does not treat `Create` and `Destroy` different from any other message[4]. These two reasons prohibit MELDC from short-circuiting message passing to `init/term` methods. The `Create/Destroy` must take the input parameters and pass them to the `init/term` code. In order to accept and bypass variable number of parameters, the MELDC language provides a mechanism called *optional parameters*. Regarding object management, each class maintains a list of objects and provides necessary information to other objects. Based on thr information, for example, a management object is able to send messages to the objects of the same class.

3.5.2 Class/Object Knowledge Inquiry

`Metaclass` also provides the mechanism for objects to inquire the definition and the status of a class. Some useful information includes (1) type descriptions of methods, parameters and instance variables, (2) names of methods, parameters and instance variables, (3) offsets of instance variables and methods in object structures, and (4) offsets of parameters in message structures.

3.5.3 Generic Interface

`Metaclass` provides a generic interface to access "external" objects. An object is external if it may reside outside the process's address space. Examples includes persistent objects, remote objects and database objects. When external objects reside outside the process's address space, they may be structured in different forms and identified by different naming schemes. In the MELDC programming model, a MELDC program consists of a set of internal objects, which interact with several *domains* of external objects. Each domain of external objects is identified by an internal object, called a *protocol object*, which implements the interface to access the external objects. Protocol objects serve as communication gateways to external objects and each of them may have its own naming scheme to identify objects. MELDC do not enforce a global unique naming space for objects. Instead, it is up to MELDC programmers to define their protocol objects to identify different domains of external objects. A protocol object must have two required methods (`init_object` and `register_object`).

`Metaclass` provides two methods,`GetObj` and `PutObj`, for programmers to establish and remove a connection with an external object.

[3]A simple approach is to dynamically construct new classes through dynamic linking. An object is considered as a class as long as (1) its class is `Metaclass` and (2) its instance variables (defined by `Metaclass`) contains the definition of some objects.

[4]C++ treats `new` and `delete` as special operators.

```
obj = Class.GetObj(obj\_name, protocol\_obj, opt\_param);
Class.PutObj(obj, Obj\_name,  protocol\_obj, opt\_param);
```

GetObj creates a local representation of an external object and setup necessary communication to the external object. The operation is to "localize" the external object and make it accessible by other local objects as if it is an internal object. It calls the init_object in the protocol object to perform any domain-specific initialization task. PutObj "externizes" a local object and destroy the local copy of the object. It calls the register_object. Communication protocols, policies and concurrency control are specified in protocol objects. For example, a programmer can implement a protocol object for distributed objects. An object created through GetObj with the protocol object is transparently accessed by other local objects. Its implementation is fully hidden in the protocol object. For example, to improve the performance of his application, he may implement three protocol objects. One replicates objects in multiple sites, one moves remote object to the local site, and one simply use remote procedure calls for each message passing. The implementation relies on the mechanism of dynamic composition, which use shadow objects as the access points. See next section for details. Similar to Create method, optional parameters are passed into the protocol object. The Getobj and PutObj provide a uniform interface to access external objects.

3.5.4 Dynamic Composition

Metaclass provides two methods for dynamic composition: Attach and Detach. See next section for details.

4 Dynamic Composition of Object Behavior

MELDC provides the mechanism of *shadowing* to implement secondary behaviors. The idea "shadow" implies a dynamic, transient and orthogonal effects upon primary behavior. A secondary behavior of an object **o** is implemented by being attached with a second object, called the *shadow object* of **o** (referred to as S_O). We say that object **o** is *shadowed* by S_O or S_O casts shadow upon **o**. The primary behavior of S_O is the secondary behavior of **o**. The shadow object S_O alone is a regular object, whose primary behavior is defined by its own class . It has its own private data as well as its own threads of control. It can be shadowed by yet another object, which would then implement a secondary behavior of S_O. An object cannot be attached directly or indirectly to itself as a shadow object. From our previous example, an account object can be shadowed by an audit object, which can be shadowed by a monitoring object (attached by a higher manager). However, it is impossible for an audit object to be audited by itself.

The attachment used to establish the connection between objects and their shadow objects is a method (attach) which is defined in metaclass. To attach S_O to **o**, a message is sent to the class of **o** with four parameters:

$$\text{Class}_O.\text{attach}(\mathbf{o}, S_O, m_{entry}, m_{exit})$$

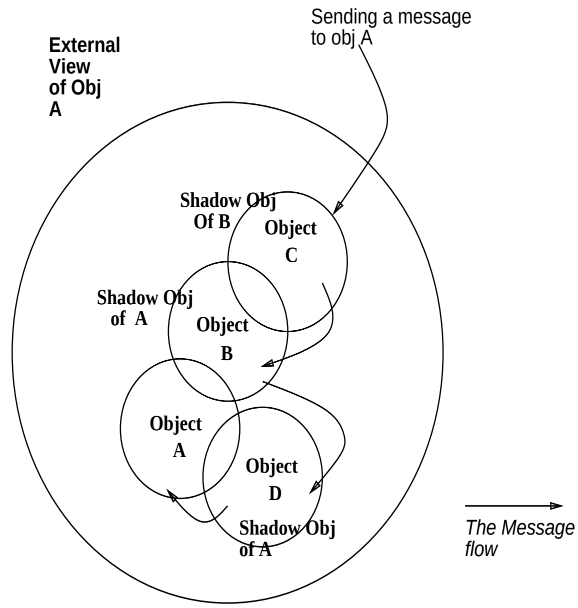

Figure 3: Simple Examples of Shadow Objects

The third parameter, m_{entry}, indicates the entry method to the shadow object S_o and the forth, m_{exit}, indicates the exit method. Any message m sent to object **o** will be trapped and forwarded to the method m_{entry} of S_o before it is delivered to **o**. After **o** completes executing method m, m_{exit} of S_o is invoked.

An object may have a sequence of shadow objects, S_1, $S_2 \ldots S_n$, attached to it, where S_n is the last one attached and S_1 is the first one being attached. The entry method of S_n is the first one to be executed and that of S_1 is the last. After **o** completes executing its method, the exit method of S_1 is executed first and that of S_n is the last one to be executed. Any shadow object S_o may be detached from object **o** as long as there is no active message traveling through the set of shadow objects. If there are any active messages, the detachment operation is delayed until the message is completed. Shadow objects are the basic building blocks to dynamically change program behaviors. Figure 3 shows the control flow of an object shadowed by a set of objects.

4.1 MELDC Shadow Object Case Studies

The dynamic composition is a useful construct used to reduce the complexity for programming a large scale object system. Dynamic composition provides a clean separation between object behavior and runtime policy. By separating runtime policies and encapsulating them within object boundaries, programmers can design a system that intelligently alters its behavior while achieving reasonable efficiency. The shadow objects are the mechanisms to clearly and efficiently implement solutions of the dynamic composition. In our first example, we consider how to transparently access remote information and object migration. Our other solution to a system problem deals with the property of persistence. Here, we attach a shadow object to an object as protection against loss of the object's data. This shadow object behaves completely transparent in terms of primary behavior, but assures a programmer a determined level of safety.

4.1.1 Objects with Persistence

Often in large systems, it is desirable to imbue objects with the quality of *persistence* for purposes of fault tolerance or system-related aspects such as transaction and database. This persistence can have many forms, but the common thread among them is that the data for the objects with the property of persistence should be safe from deletion upon the object's (or system's) demise. Acquiring persistence, should not in any way alter an object's original behavior. It should be transparent to the objects. The persistence by itself makes very little sense, it has to co-exit with objects. The persistence is acquired dynamically by the objects. The property of persistence makes it an ideal candidate for dynamic composition. The persistence can be implemented as a shadow object. Attaching persistent shadow object to an object, transforms the object into a persistent object. Once the object has acquired the persistence behavior, it will stay persistent until the object is destroyed or detached from the persistent property. Notice adding and altering the persistent behavior of an object do not effect the objects , and the object is completely unaware of this behavior. One of the biggest benefits of implementing persistence qualities with shadow objects lies in their dynamic nature. Traditionally, the policy of the persistent behaviors are incorporated as part of the language system. A program has very little control over persistence policy. Treating the persistence as a secondary behavior will separate the concept of persistence from the system as well as the objects. Systems can dynamically configure the objects with a different policy. For example, not every object in a system requires the same degree of persistence (degree of persistence implying the rate of data flushing). It is wasteful to assign a high degree of persistence to every object, simply because of the requirements of a few objects in the system. A policy function can dynamically determine the persistence requirement of an object and designate a persistent policy to each object. This policy function can also periodically reevaluate the persistence requirement of each object, and adjust their need accordingly by attaching a different persistent object.

4.2 Object Migration and Transparent Remote Access

Sometimes it is difficult to convince someone that remoteness can be considered as a secondary behavior. To a process, any object not located within it is consider a remote object. A remote object can be viewed as a local object that exhibits a remoteness property. Conversely, a remote object can also be viewed as a physically distinct object.

What distinguishes these two views is the method needed to access these remote objects. In the first case, since the object is treated as local, the programmer only need to send a message to the object to communicate with it. In the second case, the object is physically far away and the programmer must go through a complex interface and "build a bridge" to the object.

4.2.1 Object Migration

By treating a remote object as a local object with remoteness property, object migration is simply attaching a remoteness property to a local object. Once the object is given the remoteness property, the object is automatically migrated to the remote environment. The object's local representation

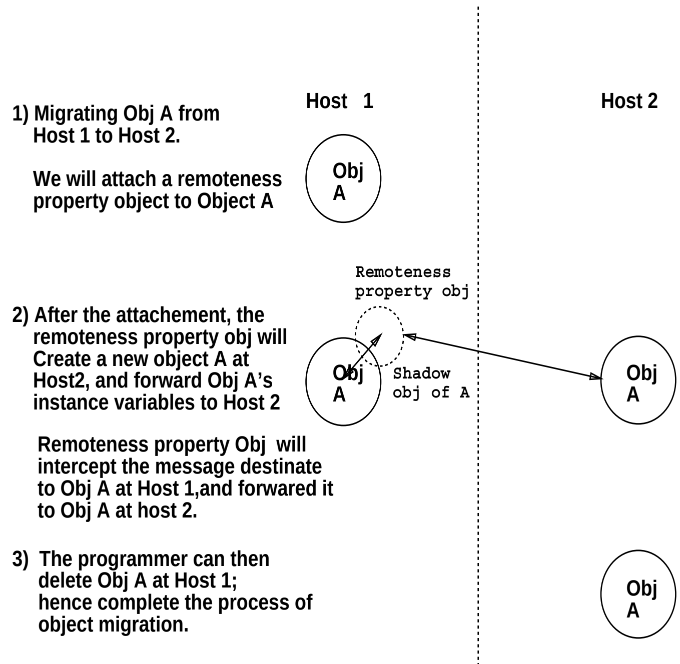

Figure 4: Simplified Illustration of Object Migration

became a dummy stub, which will be used to forward all the messages to the migrated remote object.

Object migration has not yet been implemented in MELDC. Its design very closely follows the model described above, with a few enhancements and limitations. From the users point of view, migrating an object to the remote machine is accomplished by attaching a remoteness property object to the migrating object. Once the attachment is complete, the physical entity of the object will be forwarded to the remote machine and the user can still access the object through the local stub. Programmers do not have to be concerned with the complex details of network communication, linearization and name resolution. These are all hidden away by the remote property object. There is a limitation with the current design, since MELDC class definitions do not include the actual source/object code. Migrating a class is not possible unless the source/object code has been built into the executable. The object can not be migrated unless its class exists at the remote process.

In order to avoid exponential explosion of local representation, after the object is migrated the programmer has the option of deleting an object's local representation. Deleting the local object's representation has no affect on its remote counterpart. Deleting a local object is different than detaching the remoteness property. Detaching the remoteness property from a migrating object will move the object back to the local environment. Deleting the migrated local object will make the remote machine the permanent home of that object. Figure 4 provides a high level abstraction of MELDC object migration.

4.2.2 Transparent Remote Access

Transparent Remote Access is a special case of object migration, which has been implemented in MELDC. The idea is to access a remote object as if it were a local object. The idea is to treat the remote object as a migrated local object with its local stub removed. There are many common classes which are shared between object migration and transparent remote access (i.e. The network communication, linearization algorithm, etc.). The implementation described in the previous section is still applicable in this case. The differences between them are in the initialization process. During the initialization time, instead of forwarding all the object's information to the remote machine, the transparent remote access shadow object will create a local stub and attach a shadow object to it. The resulting configuration will be identical to the configuration of object migration. Once the attachment is complete, the user can send information from local to the remote machine transparently. At the current implementation, detachment is different than the concept in object migration. Detachment will not move the object from the remote back to the local environment. Detachment will simply destroy the local object stub. This implementation might change after the completion of the object migration. We might want to use the same model as object migration.

5 An Application: MELDNET

MELDNET is a performance monitoring system for *EtherNet* and it is built on top of MELDC. MELDNET serves as a prototype revealing technical difficulties in implementing an object-oriented real-time monitoring system [WK93]. Since MELDC is running on several versions of the UNIX time-sharing operating system, MELDNET is *soft real-time*[5] because the timer facility we used is not precise.

One major advantage of using MELDC is its support for shadow objects and protocol objects in a distributed environment [LHHK92]. In MELDC, if the real managed object is owned by the information agent and a remote managed object is owned by the monitoring system, then the remote object does not contain the actual value. If a management application performs a GET request on one particular remote object, this request is *delegated* to the real object under the agent. The return value from the real managed object is returned to the management application originating the request. The interaction between the remote and real managed object, as depicted in Figure 5, is transparent to the application, and it is not hard real-time. As described in [WK93], it is more efficient if the remote object keeps a copy of the value and the real managed object periodically sends new values to update the remote replicated copy as in Figure 6. Please note that the remote object in this case is simply a way of *caching and replicating* remote management information with periodic refreshing [LHM$^+$86]. Furthermore, the quality of the information service (QOIS) is decided by the refreshing rate.

Besides object access transparency, the shadow object in MELDC provides another performance advantage in distributed real-time network monitoring. Usually, every request from an application to an agent needs to go through an application layer protocol (e.g., CMIP or SNMP).

[5] The major difference between hard and soft real-time: In a hard real-time system, meeting all the time constraints is guaranteed. But, in soft real-time system, the information about time constraints is treated as a hint to the scheduler for how to schedule the tasks.

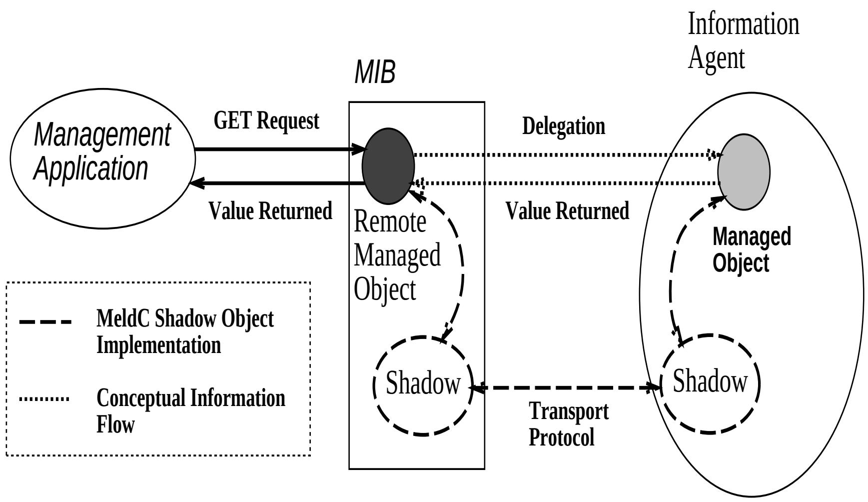

Figure 5: The Remote Managed Object in MELDNET

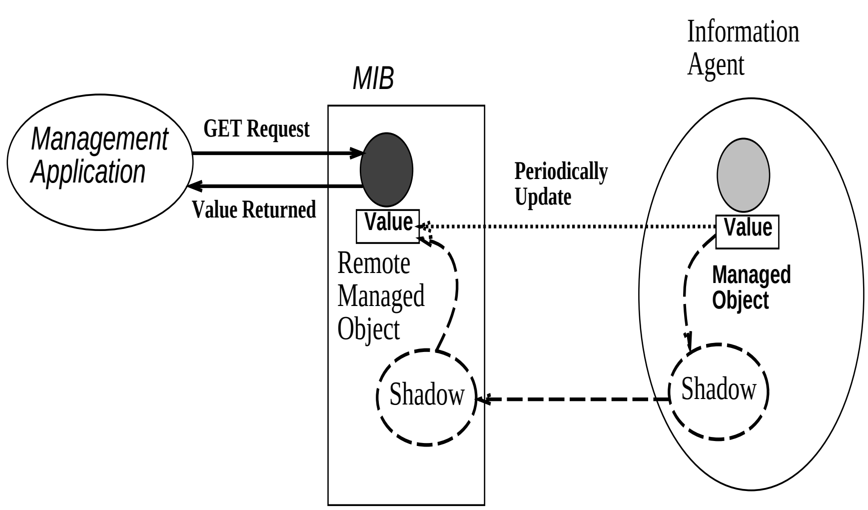

Figure 6: The Remote Replicated Object in Real-Time Monitoring

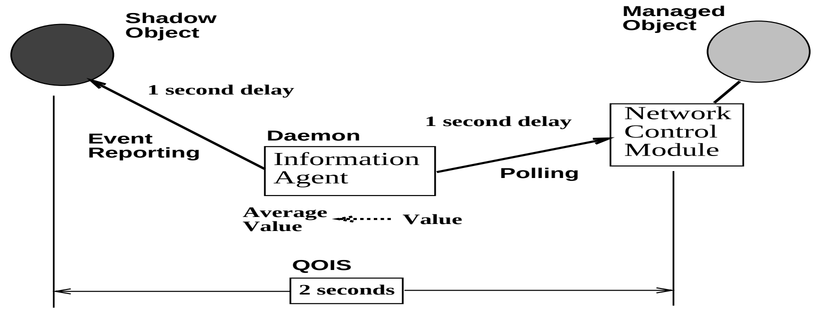

Figure 7: The Daemon Approach

The protocol processing, which includes object identifier checking, type checking, and access mode checking, is time consuming. Thus, it is difficult to achieve hard real-time in CMIP or SNMP. In MELDC, the interaction between the managed object and its shadow is directly on top of a transport layer protocol. All the checking is done only once at the creation time of that shadow object. Therefore, using shadow objects in MELDC achieves better performance in accessing information from the agent.

Since in the current implementation no timer can be attached to the remote managed object in MELDC, we used two pairs of remote and real objects differently in order to implement the periodic event reporting as shown in Figure 7. First, we put a real managed object in the real-time MIB and its remote managed object is left in the agent side (Figure 8). The agent still polls the information periodically from the network controller. However, it also periodically computes the average value and performs a SET operation on the shadow. This SET request is delegated to the real one in the real-time MIB and the value of the real managed object is updated. Thus, if the cycle time of the SET operation is short and the delay between the remote and the real object is short, the QOIS is guaranteed.

One question about the MELDNET implementation is: "*Will there be too many shadow objects with this approach, which takes a lot of memory resources?*" In [WK93], we argued that in a *hard real-time* monitoring system, at one time instant, management applications can only access a *relatively small, bounded, specific* set of managed objects with specified QOIS. If the number of shadow objects is large, the system will not have enough communication and computation resources to handle all the QOIS. Thus, at any moment, the number of shadow objects is small and fixed. Furthermore, shadow objects in MELDC are easy to create and destroy dynamically.

6 Summary of MELDC language features

This section will summarize the important features in MELDC to give you a better understanding of the overall MELDC language. Figure 9 shows the graphical interaction of the MELDC features.

6.1 Message Passing

In MELDC, objects communicate with each other through the message passing mechanism. There are three different types of messages:

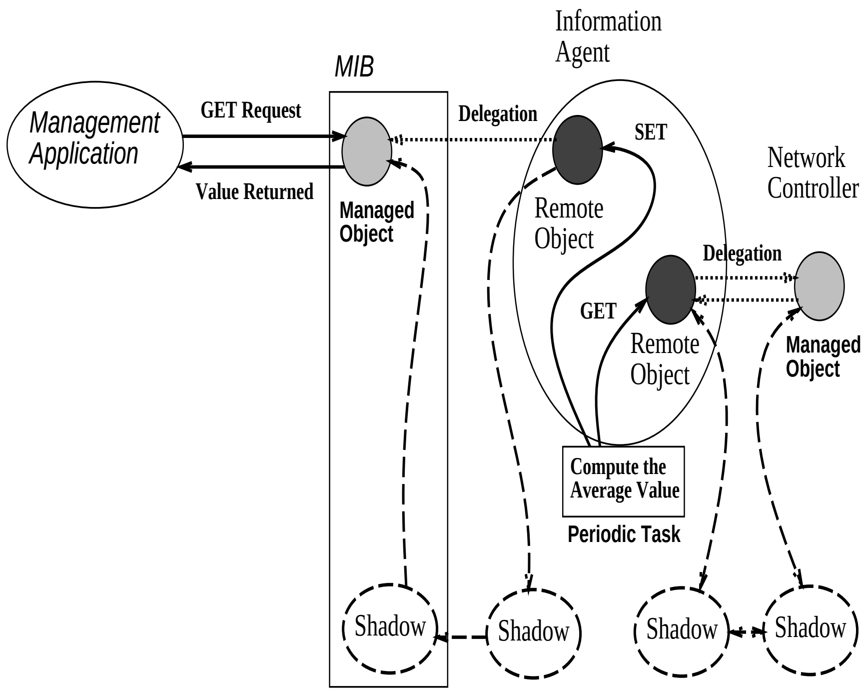

Figure 8: The MELDNET Implementation of the Shadow Object

- Symbolic
- String
- C like (fast message)

An object can send one of these messages either synchronously or asynchronously. When a MELDC object receives a message, its default behavior is to match the message with one of its selector and fire off the method corresponding to that method. This type of message is called a symbolic message. A symbolic message is similar to a function call. A simple improvement might be for an object to treat the incoming message as a regular expression. The object can then match this regular expression against the actual method name. This type of the message is called a string message. Using the MELDC messages passing schemes can entail much overhead. Sometimes, it would nice if one can fire off a C function call and by-pass the MELDC thread mechanism for the purpose of efficiency. This type of message is called C like message.

Synchronous messages in MELDC are similar to C function calls. The caller will wait for the completion of the callee before the caller continues execution. For an asynchronous message, on the other hand, the caller will not wait for the callee. Instead, the callee will be executed on a separate thread.

MELDC allows any given method to have a variable number of formal parameters. Optional parameters is a very useful feature in a programming language. It allows the programmer more flexibility by making modules more generic. One can use this mechanism to implement the parametric polymorphism, like in C++, which allows different functions to executed depending on the type of parameters.

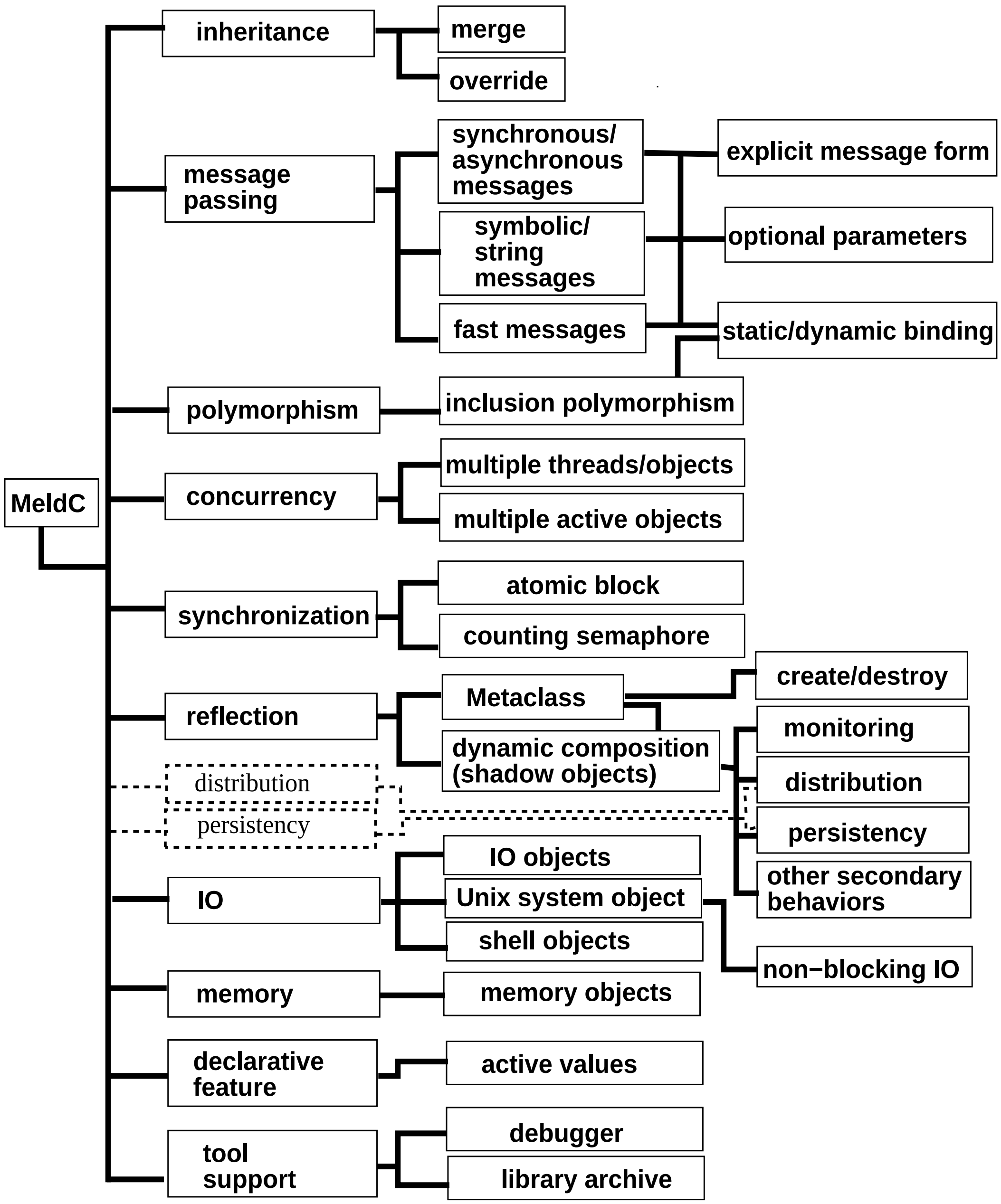

Figure 9: Important MELDC Features

6.2 Concurrency/Synchronization Mechanism

Since MELDC is claimed to be a concurrent OOPL, the concurrency obviously plays a very important role in our language. In MELDC, there are two different levels of concurrency:

- The concurrency between objects
- The concurrency within an object

Many active objects can execute simultaneously and multiple threads can be executed in an object at the same time. MELDC threads are lightweight threads, during context switching very little information is saved, hence it is less expensive than heavy-weight threads.

The synchronization mechanism is one of the crucial components in a concurrent language. In MELDC, two different types of synchronization controls are provide for the programmer:

atomic block: The objective of *Atomic Block* is to protect instance variables from being accessed or modified by more than one thread at the time. Once a thread enters the atomic block, any other threads attempting to send messages to this object will be suspended until the atomic block is released. An atomic block provides the mechanism to synchronize the threads within the same object.

counting semaphore: MELDC also provides the counting semaphore to synchronize the execution between threads executing in different objects.

6.3 Inheritance

MELDC provides two different types of inheritance:

override: The override semantic is similar to C++, which during a name conflict the child's behavior will override the parent's behavior.

merge: Unlike override, the merge semantic does not consider multiple declarations of a method in an inheritance tree a conflict. MELDC simply executes all of the methods. The merge semantic is an experimental inheritance semantic. There are few problems that have not yet been correctly resolved, such as the return value for the merge function. Nonetheless, it could be a possible alternative to the override semantics.

6.4 Reflection

Two very important ideas in MELDC are derived from the concept of reflection. MELDC's metaclass is a form of *Structure reflection*, and MELDC's dynamic composition is a form if *Computation reflection*.

MELDC's metaclass is the class of all the classes in MELDC. The metaclass defines the behavior of all classes and is implemented in the MELDC language. Any object oriented language without the concept of a metaclass will have to hard code an object's creation and destruction as a part of the language system. In the case of MELDC, the `Create` and `Destroy` methods are actually written in MeldC, both of which are methods of the metaclass.

Sometimes a programmer needs to change the behavior of a MELDC program "on the fly." The power to dynamically change program behavior can be extremely useful in debugging large programs and auditing large object-bases. It can be used for more advanced uses, as well. All of MELDC's distributed programming aspects are based on the concept of dynamically modifying program behavior to form a link to other MELDC processes. In MELDC, however, this power does not come from self-modifying code. Rather, MELDC offers the dynamic extension of object behavior through the use of the reflective (Computational reflection) architecture.

6.5 Active Value

Active values are variables that cause side effects (typically assigning a value to another variable) to occur immediately after their value is changed or they are assigned a value. For example, if we have an equation which provides the relation between Fahrenheit-Celsius. Using the active-value, it does not matter whether F or C is changed, the MELDC will ensure the resulting values will be correct by changing F or C proportionally.

6.6 UNIX I/O Interface

UNIX processes can be executed concurrently inside the UNIX kernel. For example, any two UNIX processes can concurrently initiate I/O to the same device. Unfortunately, this is not the case for the MELDC light-weight threads. Even though UNIX does provide the asynchronous I/O facility, the task might be too complex for the regular programmer. MELDC provides a class which simulates the synchronous UNIX I/O for each light-weight thread. The use of a UNIX I/O interface can give the user the appearance of a blocking I/O and yet at the same time, allow other threads to continue their execution.

6.7 Tool Support

There are two important tools that are commonly used by the MELDC programmer:

- the debugger
- the library archive

The MELDC debugger is built on top of gdb. This gives mcgdb a means to debug MELDC code with a standard interface. It also allows the use of gdb's ability to step through and examine the C statements which compose the majority of a MELDC program's methods. The mcgdb will also allow a programmer to exam the instance variables of an object.

The library archive, mar, allows a MELDC programmer to group a set of related classes together and form a class library. The library archive also provides many facilities like revision control and the ability to determine the dependencies of a feature, which automatically places/extracts them in/from the same archive. The mar command is also a good mechanism to organize the classes in the MELDC language.

6.8 Memory Management

The implementation of MELDC in C implies that the global heap space is shared by all objects. This might not be an acceptable solution. Instead, we believe each individual object should have its own memory space. That is, each piece of the memory allocated should have an owner object associated with it. This means that only the owner is allowed to free the object's memory. By localizing the memory allocation, we reduce the chance of memory leak and yet at the same time, it is extremely helpful in debugging a program with a memory problem.

7 Conclusion

The goal of any high level language is to provide a comfortable programming environment for the application programmers. This is one of the important objectives but not the only one behind the MELDC language design. We would also like to provide a language environment where the programmers have the option to modify or add those high level features provided by the language *Remote*, *Persistent* .... without changing the core definition of the language. We believe the concept like remote object and persistent object, should not be consider as part of the language. Instead these propertes should be encapsulated inside an object and the language should provide a mechanism (the shadow object) which allows any object to attach the remote and the persistent behavior dynamically(*Dynamic composition*).

The MELDC 2.0 implementation consists of about 15,000 lines of C, lex and yacc for the compiler, 4,300 lines of C and 500 lines of assembly code for the kernel, plus 10,000 lines MELDC runtime written in MELDC itself. It runs on Sun4s with SunOS 4.1 and DecStations with Ultrix 4.2, although there are several limitations on the DEC version. This is the first external release of MELDC, but version 1.0 has been used internally as an educational language for undergraduate courses. The release includes a user manual, compiler and runtime implementation guides, a MELDC variant of the gdb debugger and a sample program for network monitoring.

References

[Bob88] D.G. Bobrow. Common lisp object system specification x3j13 dcoument 88-002r. *SIGPLAN Notices*, 23, September 1988. Special issue.

[Boo83] Grady Booch. *Object Oriented Design With Applications*. Benjamin/Cummings, 1983.

[CG89] N. Carriero and D. Gelernter. Linda in Context. *Comminications of The ACM*, 32(4):444–458, April 1989.

[Cia90] Paolo Ciancarini. Coordination Languages for Open System Design. In *International Conference on Computer Languages*, pages 252–260, March 1990.

[Coi87] Pierre Cointe. MetaClasses are First Class: the ObjVlisp Model. In *OOPSLA'87*, volume 22, pages 156–167, 1987.

[Fer88] Jacques Ferber. Conceptual Reflection and Actor Languages. In Pattie Maes, editor, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[Fer89] Jacques Ferber. Computational Reflection in Class based Object Oriented Languages. In *OOPSLA'89*, volume 24, pages 317–326, 1989.

[Hen86] J. Hendler. Enhancement for fMultiple Inheritance. *SIGPLAN Notices*, 21(10):100, 1986.

[HLK92] Wenwey Hseush, James C. Lee, and Gail E. Kaiser. MeldC Threads: Supporting Large-Scale Dynamic Parallelism. Technical Report CUCS-010-92, Department of Computer Science, Columbia University, July 1992.

[IC88] Mamdouh H. Ibrahim and Fred A. Cummins. KSL: A Reflective Object-Oriented Programming Language. In *International Conference on Computer Languages*, pages 186–193, October 1988.

[KHPW90] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich, and Shyhtsun F. Wu. Multiple Concurrency Control Policies in an Object-Oriented Programming System. In *2nd IEEE Symposium on Parallel and Distributed Processing*, pages 623–626, Dallas TX, December 1990.

[KPHW89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush, and Shyhtsun Felix Wu. MELDing Multiple Granularities of Parallelism. In Stephen Cook, editor, *3rd European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 147–166, Nottingham, UK, July 1989. Cambridge University Press.

[LHHK92] James Lee, Wenwey Hseush, Eric Hilsdale, and Gail E. Kaiser. Dynamic Orthogonal Composition in MeldC. In *Workshop of Objects in Large Distributed Applications*, Vancouver, British Columbia, Canada, October 1992.

[LHM+86] Bruce Lindsay, Laura Haas, C. Mohan, Hamid Pirahesh, and Paul Wilms. A Snapshot Differential Refresh Algorithm. In *SIGMOD*, pages 53–60, Washington, D.C., May 1986.

[Mae87] Pattie Maes. Concepts and Experiment in Computational Reflection. In *OOPSLA'87*, volume 22, pages 147–155, 1987.

[Mae88] Pattie Maes. Issues in Computational Reflection. In Pattie Maes, editor, *Meta-Level Architectures and Reflection*, pages 21–34. North-Holland, 1988.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In *OOPSLA*, volume 21, pages 1–8, 1986.

[Pu91] Calton Pu. Generalized Transaction Processing with Epsilon-Serializability. In *Proceedings of Fourth International Workshop on High Performance Transaction Systems*, Asilomar, California, September 1991.

[Wat88] Takuo Watanabe. Reflection in an Object-Oriented Concurrent Languages. In *OOPSLA'88*, volume 23, pages 306–315, 1988.

[WK93] Shyhtsun F. Wu and Gail E. Kaiser. On Hard Real-Time Management Information. In *IEEE First International Workshop on System Management*, page to appear, Los Angles, California, April 1993.