An Object-Oriented Run-time System for Parallel Applications

Steve MacDonald, Duane Szafron and Jonathan Schaeffer
Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1
{stevem, duane, jonathan}@cs.ualberta.ca

ABSTRACT

This paper describes three basic specializations of design patterns that can be used in implementing the run-time systems of parallel applications. These specializations were discovered while re-designing and re-implementing the run-time system for the Enterprise parallel programming system. Enterprise allows programmers to create, compile, execute, and debug programs that execute over a network of workstations. The run-time system is responsible for the execution of user programs. It was recently re-designed using objects to rectify problems with the previous version and to provide an extensible base for further research. This papers details the key object-oriented components of the new system. Although these components were developed in the context of the Enterprise system, they are generally applicable to object-oriented run-time systems for general parallel applications.

1. Introduction

This paper describes the new object-oriented run-time system for the *Enterprise* parallel programming environment [SSLI 93]. This run-time system is responsible for the correct execution of Enterprise applications over a network of workstations within the bounds of the programming model presented by the system [Mac 95].

There were two primary objectives of this work: to remove limitations and deficiencies in the previous implementation, and to create an extensible system that can be used as a basis for further research. Although the first goal could have been solved without using an object-oriented design, several deficiencies were solved specifically by using object-oriented techniques. However, the second goal was the main motivation for using objects and is the main focus of this paper.

The previous version of the Enterprise run-time system was written in C, with a Smalltalk user-interface. Since Enterprise is an exploratory research project, there were no clear requirements for the system before it was built. As the research progressed, the system degraded into a large, complex program that could only be maintained by the original developers, and then only with considerable effort. Eventually the system became so complex that it could not effectively be used for further research. By rewriting the code using an object-oriented design and language, we hoped to remove these problems and create a system that is flexible and extendible to meet future needs and requirements.

This paper discusses the design of the new Enterprise run-time system, concentrating on key object-oriented components. The principle contributions and lessons learned were how object-oriented techniques can be applied to:

1. create a flexible system by supporting the programming models of Enterprise in an abstract way,
2. abstract the processing of messages by modeling the responsibilities of different types of processes,
3. create a system where the responsibilities of a process can be dynamic, changing as the application executes, and
4. encapsulate the communications sub-system so that its implementation may be changed without any effects on other parts of the system. This increases portability and allows multiple implementations to be supported.

2. Enterprise

Parallel programming is usually perceived as a difficult task. The programmer is forced to deal with complexities that do not arise during sequential programming, such as non-determinism, communication, and task coordination. In addition, parallel programs use a different set of algorithms which typically do not correspond to the best sequential algorithms, requiring the programmer to use different approaches to solve a problem. Correctness

This is a preprint of a copyrighted article that will appear in TOOLS USA '96, July 1996, Santa Barbara, USA.

debugging becomes a much harder task, largely because of non-determinism. A program may fail only on certain execution paths, which may not occur every time the program runs. Adding additional code or attaching a debugger to the program can alter its execution, preventing the error from occurring. After the correctness of the program is verified, the programmer has to do performance debugging, tuning the application to achieve maximum performance. This last activity is hampered by a lack of tools and methodologies to aid and evaluate the process. In addition, performance debugging can introduce logic errors into the application, which makes correctness and performance debugging a cyclic process.

The Enterprise system provides the user with an environment to write, compile, execute, and debug parallel programs. The main feature of Enterprise is its ability to simplify parallel programming by removing some of these additional complexities. Communication and task coordination are the responsibilities of the run-time system, based on user annotations that specify the parallelism in the application. Correctness debugging in Enterprise is accomplished through a program replay mechanism, which re-executes a program based on events collected at run-time, isolating the effects of non-determinism in a program during the debugging phase. Performance debugging is simplified by the separation of the user annotations and the application code, which allows the parallel structure of the application to change with few changes to the application code. All of these features are provided within a simple programming model that lets the user write their application in sequential C. This programming model allows the programmer to use a familiar, existing programming language and also allows legacy code to be easily incorporated into a parallel application.

2.1 Programming in Enterprise

There are two components to the Enterprise development model: the *programming model* to specify the semantics of the application code and the *meta-programming model* to specify the parallelization techniques the user wishes to use.

The programming model in Enterprise is based on the futures model of parallel computing [Hal 85]. Futures are tagged variables that represent the results of parallel function calls, either through return values or side effect. When a process invokes a parallel call, the arguments are marshaled into a message and sent to the remote process responsible for executing that call. Unlike RPCs (remote procedure calls), the caller continues executing without waiting for a response until a future is accessed. At this time, if the results of the parallel call are not available, the caller blocks and waits for them. The futures model is demonstrated in the following program fragment:

```
result = parallel_f(a, b);
/* Other C code executing        */
/* concurrently with parallel_f  */
. . .
/* Block if result not available */
x = result;
```

The main benefit of the futures model is that the user can write normal C code rather than having to learn a new programming language. This feature also allows existing code to be used with little modification. The foremost exception is that global variables are not supported in Enterprise since each process executes in its own address space. Also, to make effective use of futures, the user should delay accesses to the results of parallel function calls until they are needed. This step is only necessary to increase concurrency; the program will execute without this step, just not as efficiently. There are also some small differences in the semantics of the language, such as the requirement that the user specify the number of elements for any data passed via a pointer. The user also specifies how the data is passed (to the called process only, to the calling process on return only, or both ways). These changes are used to reduce expensive copying and transmission operations and have been made as unobtrusive as possible.

The Enterprise meta-programming model provides the user with an abstraction for the parallel aspects of a program. The abstraction is an analogy between the structure of a program and a business enterprise. The parallel components of a program are referred to as *assets* since a business is organized by charts representing the assets of the business (like individuals, departments etc.).

Assets are really just commonly occurring parallel design patterns and the fundamental components of these patterns [GHJV 93]. Assets are divided into two types: *singular assets* and *composite assets*. A singular asset is akin to one process in the application, or a single person in a business enterprise. It is responsible for executing the sequential user function that has the same name as the asset. Composite assets are structured collections of other assets (both singular and composite) that specify the different parallelization techniques available to the user, the equivalent of an organized group of people in an enterprise. Just as there are multiple ways to group people together, there are multiple composite assets to represent different organizational strategies. These assets are combined to form an *asset graph* that specifies the parallel aspects of the program. Descriptions of the different asset types follow.

An Enterprise program starts with an instance of an *enterprise asset*, which represents the entire program or organization. This asset contains only one component, which represents the parallelization technique used for the program at its highest level. Initially, the enterprise asset contains a single *individual asset*.

An *individual asset* is analogous to a single person in an organization. It is associated with a user function, which must have the same name as the asset itself. When invoked, this code is run to completion. Subsequent calls to the same asset must wait for the previous ones to finish. Individual assets can also be *coerced* into other asset types, which replaces the individual with a new asset of a different type. This operation allows a user to create an arbitrary graph by refining the parallelization technique used on an individual.

A *line asset* is a composite asset that is analogous to an assembly line (a *pipeline*). It contains a *receptionist asset* followed by a fixed number of heterogeneous assets (either singular or composite) in a specified order. The receptionist is a special kind of individual asset that is the entry point of a composite asset. The receptionist has code attached to it and shares its name with the composite asset that contains it. Each asset in the line accepts work from the previous asset, refines the data, and passes it to the next asset. Concurrency is obtained since a subsequent call to the line only waits for the receptionist to finish executing the previous call, not the rest of the assets in the line.

A *department asset* is a composite asset that is analogous to a department in an organization. It contains a receptionist asset and a fixed number of heterogeneous assets (either singular or composite). The receptionist accepts all incoming calls and forwards them to the appropriate component asset. Each call to the department must wait only for the receptionist to become available, not for the entire department to finish with the previous call.

A *division asset* is a parallel recursive structure. It consists of a number of identical assets and a specified depth. Recursive calls are made in parallel until the depth is reached, when further recursive calls are made sequentially. Nodes that perform recursive calls sequentially are called the *leaves* of a division. All non-recursive calls are always made in parallel.

A *service asset* is a resource in an organization that is not consumed and has no ordering restrictions for its use. A clock is a good example of a service.

Enterprise provides the ability to *replicate* certain asset types, creating multiple processes to execute the asset. Replication is specified as a minimum and maximum pair, with Enterprise dynamically creating and destroying processes at run-time as system resources change. Individuals, departments, lines, and divisions can be replicated. The new run-time system supports the dynamic addition of processes to a running application. However, the criteria for invoking this feature is the subject of current research.

Here is the code for an example program. This program is a parallel equivalent of the simple "hello world" C program and is intended for illustrative purposes only.

```
#define SIZE   10
CubeSquare( argc, argv )
int argc;
char * argv;
{
  int i, sq, cu, a[SIZE], b[SIZE];
  for( i = 0; i < SIZE; i++ )
  {
    a[ i ] = Square( i );
    b[ i ] = Cube( i );
  }
  sq = cu =0;
  for( i = 0; i < SIZE; i++ )
  {
    sq += a[ i ];
    printf("%d^2 = %d\n",i,a[ i ]);
    cu += b[ i ];
    printf("%d^3 = %d\n",i,b[ i ]);
  }
  printf("squares %d\n",sq );
  printf("cubes   %d\n",cu );
}

int Square( i )
int i;
{
  SLEEP_RANDOM_TIME;
  /* Appropriately defined macro */
  return( i * i );
}
```

```
int Cube( i )
int i;
{
  SLEEP_RANDOM_TIME;
  /* Appropriately defined macro */
  return( i * i * i );
}
```

The meta-program for this program is shown in Figure 1. It consists of a single enterprise asset which appears as the double line border. The enterprise asset contains a single department asset which appears as the single line border. This department asset contains a receptionist called CubeSquare and two individual assets called Square and Cube. There are two replicas of each of these individual assets. The first call to Square is handled by the first replica and the second call is handled by the second replica. If a third call is made to Square before either of the first two is finished, the call is put in a request queue. The caller (CubeSquare) does not block after any of the calls to Square or Cube. In fact, it does not block unless it accesses a future in the second loop (a[i] or b[i]) that has not yet been returned from a call to Square or Cube.

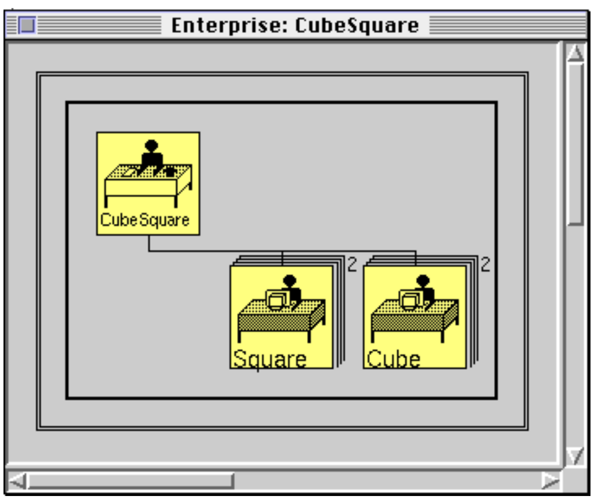

Figure 1: A meta-program for CubeSquare.

2.2 The Enterprise Environment

Enterprise is a complete parallel programming environment. The system consists of three parts: the graphical user interface, the precompiler, and the run-time system.

The user interface provides a set of programming facilities in a uniform environment. The programmer can create, compile, execute, and debug programs. The interface is not important to the remainder of this paper. Interested readers are directed to [IMMN 95] [Lob 93] [LSS 93]. The user interface was originally written in Smalltalk-80, and has subsequently been ported to VisualWorks2.0. For the purposes of this paper, the main feature provided by the interface is the ability to create an asset graph that represents the parallel aspects of an application.

The precompiler transforms the user application into a parallel program. It uses the asset graph to recognize parallel function calls and futures. For each parallel function call, it generates stubs code that marshals and unmarshals the arguments and it creates a function pointer for the user's function. It then inserts calls to the stubs code and the run-time system which imports the array of function pointers created by the precompiler.

Finally, the run-time system is responsible for the execution details of a user application. The remainder of this paper concentrates on this component.

3 The Enterprise Object-Oriented Run-time System

The run-time system is responsible for the correct execution of an Enterprise program. These responsibilities include:

- implementing the library calls inserted by the precompiler,
- launching the application in a distributed fashion, with each Enterprise process using the asset graph to launch the processes it may call,
- processing messages, including all the necessary synchronization and run-time consistency checks,
- managing futures,
- gathering run-time information for support tools such as the animation and controlled replay components, and
- shutting down the program.

The challenge in this work was to efficiently implement all the above functionality within the context of the programming model. In addition, this work was intended as a framework for future research in the Enterprise project, so the resulting system had to be flexible and extensible. This last goal was the main reason for the decision to use object-oriented technology for both the design and implementation of the run-time system. The decision to use C++ was motivated by efficiency and portability constraints.

This section discusses the aspects of the new design that benefited from the application of object-oriented technology. Although the overall system used these techniques, several parts of the design fit within an object-oriented framework better than others. These parts are: the asset graph, the *dispatcher* and *behaviour classes*, and the *communications manager*. A more detailed description of each of these components makes up the rest of this section.

3.1 Asset Graph

The asset class hierarchy is shown in Figure 2. A black square in the upper left corner indicates an abstract superclass. The asset classes rely heavily on inheritance, particularly multiple inheritance for this implementation. The concrete classes directly correspond to the different assets in the meta-programming model. These classes refine the behaviour of any operation that can be applied to instances of an asset, providing specific implementations to match the specified parallel structure. The abstract classes are:

- Asset: This class holds information common to all assets, such as the name.
- Codable Asset: This class represents all assets that have user code attached to them.
- Replicable Asset: This class represents all assets that may be replicated.
- Composite Asset: This class represents all assets that are composed of other assets, either singular or composite. It contains an asset for the receptionist and an array of component assets.
- CompositeReplicable: This abstract class was used to factor operations from its subclasses that used instance information in both the Replicable and Composite Asset classes.

The abstract classes provide common characteristics of the subclasses. Multiple inheritance allows each concrete class to inherit its characteristics cleanly, taking only those attributes that are necessary. It also allows the different characteristics to be separated and dealt with individually, rather than factoring them into other, possibly inappropriate, classes. The previous run-time system represented the asset graph using a superset of all possible asset characteristics and a set of flags to indicate the valid values. The procedures to manipulate the assets were large case statements that switch based on the asset type.

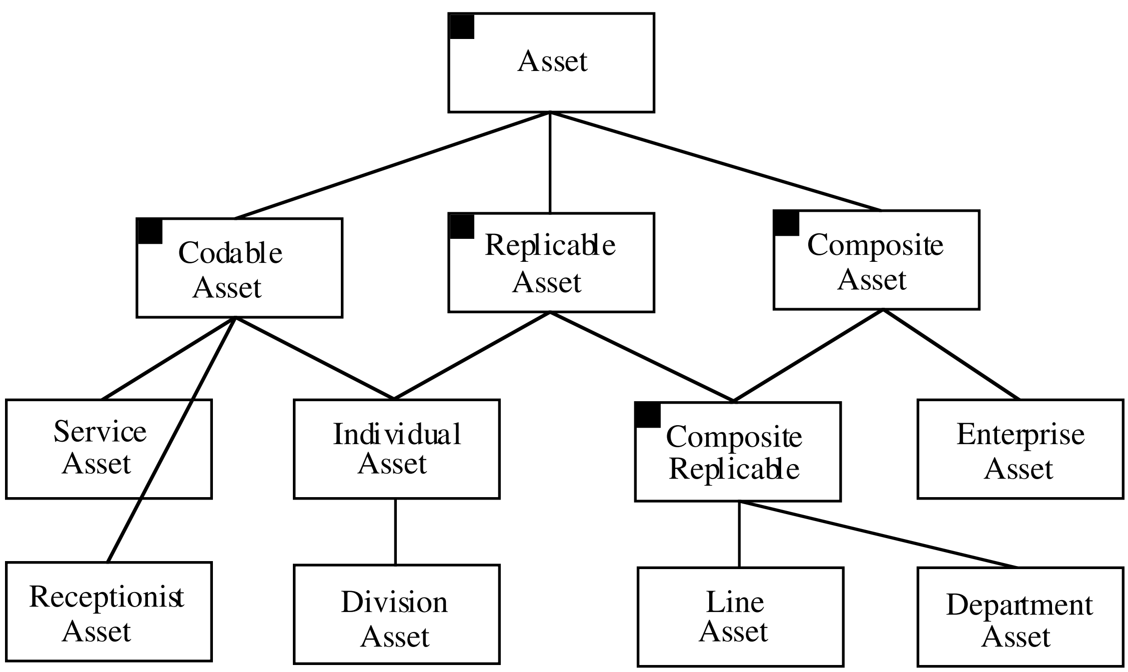

Figure 2: The asset class hierarchy.

This new class hierarchy is also an improvement over that used in the implementation of the user interface, which used single inheritance. Since the interface was written in Smalltalk and the run-time system in C++, each has its own implementation of the asset graph hierarchy. These two implementations are related only in that they attempt to implement a set of classes for the meta-programming model. Single inheritance forced the hierarchy to be factored in such a way that some operations had to be invalidated in subclasses, which violates the idea that a subclass should respond to all the messages of its superclasses. An example of this problem was replication, which was factored into the base class *Asset*. A Smalltalk message to replicate those assets that should not be replicated resulted in a run-time exception. Fortunately, context-sensitive menus precluded the sending of such messages, but it is still a design flaw. This problem was identified during the work on the user interface and reported previously in this conference [LSS 93]. Its resolution in this project helps verify the thesis that multiple inheritance is necessary for large, real-world systems.

At run-time, the assets form a communication graph based on the call graph of the program. For example, Figure 3 shows the *asset graph* for the CubeSquare program from Figure 1. Note that there are hidden nodes in the asset graph that do not appear in the meta-program. In this case the hidden nodes are managers for each replicated asset.

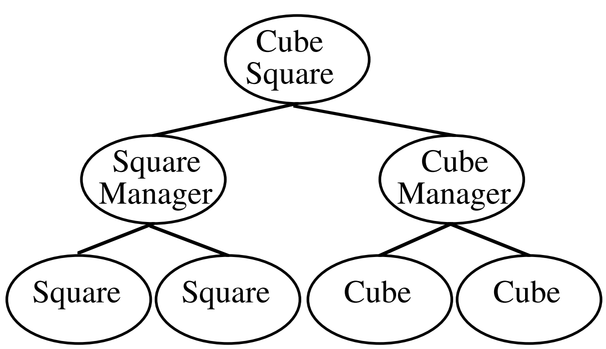

Figure 3: The asset graph for CubeSquare.

The internal representation of the graph is created by instantiating the *enterprise asset*, which contains the entire user program. Instantiating the enterprise asset creates the rest of the graph by reading it from the graph file, the textual representation of the asset graph generated by the user interface. The assets in the program are stored in the receptionist and component arrays inherited from the *Composite Asset* class. Composite assets store their components in an identical way.

One of the benefits of the overall design of the new run-time system is that the asset classes are orthogonal to the remaining constructs of the Enterprise run-time system. As long as all required public operations are still supported, the asset classes can change freely. Required operations include the launch method and methods that derive the relationships between the different assets in the graph for run-time checks. This independence also allows the asset classes to be placed in a separate library so they can be used by other components of Enterprise. For example, the precompiler can use the library to verify that the user's application code matches the asset graph. The meta-programming model itself can now be easily modified. Additional asset types and characteristics can be added, allowing the model to grow and develop as new ideas and requirements emerge. In the previous version of Enterprise, each component that required access to the graph also implemented a set of procedures to manipulate it, requiring that each component be updated when the meta-programming model changed. Since these procedures were tailored for each component, creating a single graph library was impractical.

Here is an example of the asset graph flexibility in action. The meta-programming model for the division asset was changed just as the re-design of the run-time system was completed. Previously, a division was a composite asset consisting of a receptionist and either another nested division or a *representative asset*. The depth of the division was increased through nesting, and the width was increased by replicating the components. The meta-model was changed so that a division is represented as a replicated individual with an additional depth field. While this represented a major conceptual change in the division asset, the effects on the implementation were constrained to the DivisionWorkerBehavior class. This new representation is expected to have many potential benefits, including better utilization of the processes and the ability to specify other composite assets as divisions, removing the limitation that a division consists of a singular asset.

The design of the asset graph uses the *Composite* design pattern [GHJV 93], which combines singular and composite assets. This particular implementation goes slightly beyond a straight-forward implementation to provide additional semantics for the different composite containers. The correct use of the design pattern is shown by noting that it can be used to construct any user graph, which is created using *coercion* (see Section 2.1) to nest arbitrary assets into the graph.

3.2 Dispatcher Classes

The *dispatcher class* hierarchy is shown in Figure 4. Again, the black square in the upper left corner indicates an abstract superclass.

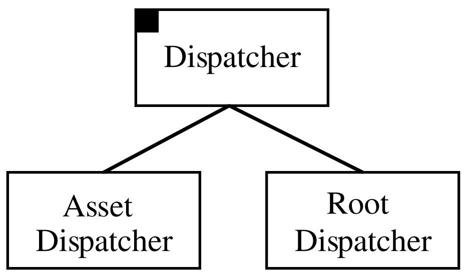

Figure 4: The dispatcher class hierarchy.

The dispatcher classes are responsible for receiving messages and delivering them to an instance of the *behaviour class* for processing. The behaviour classes are further described in the next section. For now, it is enough to know that a behaviour is responsible for the actual processing of messages. The difference between the two concrete classes is the need for buffering. Instances of *AssetDispatcher* put messages that cannot be processed immediately into a buffer. For example, if an individual asset is processing a message and it receives a second message before the first is done, the AssetDispatcher instance puts the second message into a buffer. The *RootDispatcher* class doesn't need to buffer messages since it only looks for a message when it can process one.

The main reason for the dispatcher is that it provides the opportunity for the system to be more dynamic. Since messages are processed by an instance of behaviour, the response of a process to a message can be changed by using a behaviour from a different class. Since it is impossible for an object to change its class from one behavior class to another, we provide a dispatcher that can change the behavior it uses. That is, between messages, the current behaviour of a dispatcher can be deleted and a new one created, giving the process a new role as the application executes. This change is almost transparent; the only change is in the processing of subsequent messages. One of the most important applications of this concept is that an individual asset can be promoted to the manager of a replicated set of individuals if the message queue becomes too large. The same mechanism can also be used to demote a replica manager back to an individual if the message queue shrinks. At the meta-programming level, this is equivalent to automatic asset replication and de-replication and it obviates the need for the user to specify that an asset should be replicated.

A second application of this concept is for a process to change the asset it is responsible for executing. This change is possible because all user code in an Enterprise application is compiled into one executable. That is, each process contains the code for all assets. A *replication pool* process can be created which maintains a queue of idle processes. When a replica manager discovers that its input queue is too large, it can request a process from the replication pool and map a new asset replica onto this process. When the input queue shrinks, it can release this process back to the replication pool. Note that the same process may be mapped to different assets at different times as the application executes. This replication pool could also be allowed to grow if new system resources become available, leaving the pool manager to monitor system activity instead of the replica managers.

3.3 Behaviour Classes

The behaviour class hierarchy is shown in Figure 5. The behaviour classes present all of the possible functions for a process in an Enterprise application. The behaviour classes are used to model the responsibilities of a process based on its function within the Enterprise application. This function may not be the same as its asset, since Enterprise inserts additional processes into an application that are hidden from the user. The most common example of this is a manager of replicated assets.

The current implementation of Enterprise uses the following behaviours:

- SeqRootBehaviour: This behaviour executes an Enterprise program sequentially. That is, all asset calls become procedure calls. The first asset is called sequentially.
- ParRootBehaviour: This behaviour is the root of an Enterprise program that is being executed in parallel. It sends the command line arguments to the first asset in the program and awaits the reply. It is also responsible for processing logging messages, which contain the run-time information used by the animation and reply components.
- ManagerBehaviour: This behaviour represents an external manager process, which is responsible for managing the workers of a replicated asset. However, it is also possible for

a replica manager and the asset that calls it to be put into a single process. In this case, the behaviour for that common process is a SingleAssetBehavior instead of a ManagerBehaviour. To prevent the code that actually manages the replicas from being repeated in these two places, the common code is put in a *Process* class and an instance of Process class is placed in each asset.

- SingleAssetBehaviour: This behavior represents an unreplicated asset.
- WorkerBehaviour: This behaviour represents a worker in a replicated asset. The difference between this behaviour and SingleAssetBehaviour is the need for reply messages. This behaviour must always generate a reply message to indicate the availability of the process to its manager, where the other only needs to reply when there is data to be returned.
- DivisionWorker: This behaviour represents a worker in a division, which must do additional work to properly determine if the process is a leaf of the asset.

The exact function of a process is determined during the launching phase, based on information derived from the graph that determines what processes are required for an application.

This model is a simplified version of what occurs in the run-time environment of an object-oriented language when a message is sent to an instance of a class, except that the object name space has been removed. Here, a dispatcher always receives messages from the network and always sends messages to a particular instance of a behaviour class. The actual processing of the message is done through a simple and abstract interface at the *DispatcherBehaviour* class, which is broken down into other method calls in the implementation. This breakdown in based on the different message types in Enterprise (such as request versus system management messages). These other methods are refined in the subclasses until the desired behaviour is achieved.

The behaviour classes are another orthogonal component of the run-time system, so changes can be made to it without affecting the remainder of the system. This trait creates a system that can be easily modified to include new behaviours as new asset types are created. In addition these new behaviors can be subclasses of existing behaviours for faster development. This feature has been useful during the work in parallel recursive structures in Enterprise, where a new *DivisionManager* behaviour was created to handle additional bookkeeping required for a more efficient implementation of the division asset.

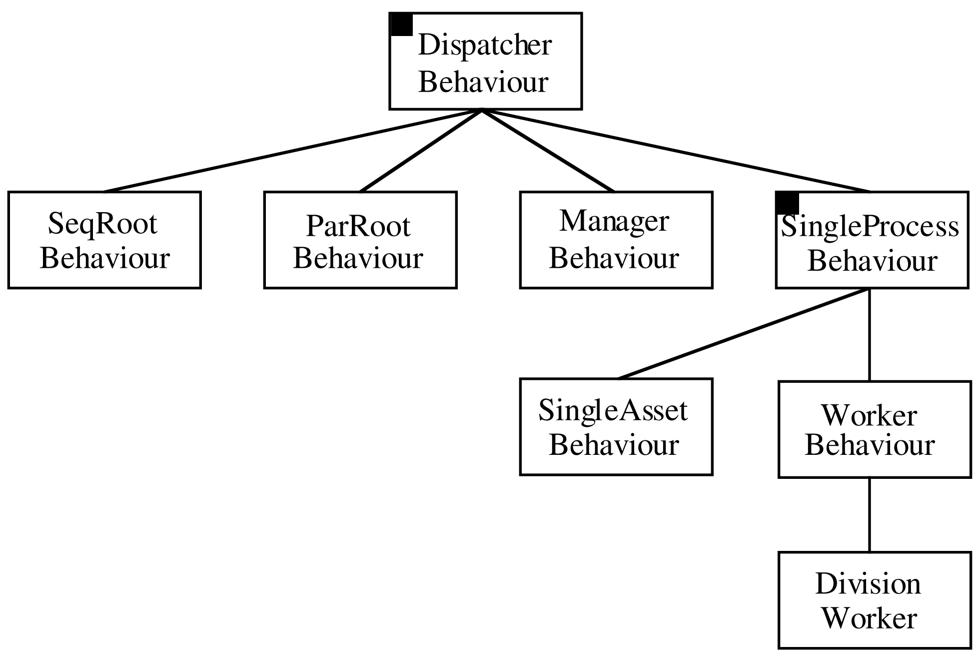

Figure 5: The behavior class hierarchy.

Using dispatcher/behaviour combinations is useful for any program where the behaviour of an object can be broken down into specific cases. In addition, abstract classes can provide a set of characteristics or factored operations that are inherited by the concrete classes. This technique is currently being used to implement template-based parallel I/O, where the template determines the behaviour of a file at a given process [Par 95]. In fact, the dispatcher/behaviour combination is an implementation of the *Strategy* design pattern [GHJV 93].

3.4 Communications Manager

The CommunicationManager class has a small well-defined interface that provides a minimal set of primitives. By providing only a few primitives, it is possible to provide several different implementations without affecting the remainder of the system. Since the interface is the same for each implementation, the only change that is necessary when switching between two different communication systems is to link different libraries into the executable. Enterprise currently has two different communications managers: one written using the PVM message passing system [Sun 90] and another using the Treadmarks distributed shared memory system [ACDK 96]. It should be noted that the Treadmarks system was only used to provide another implementation of the communication primitives; we did not re-write the run-time system to use shared memory.

Another possible design of the communications manager would be to create an abstract class with subclasses for each implementation and instantiate the correct concrete class at run-time. This idea was rejected for two reasons. First, this solution requires that the libraries to support each implementation be linked into the application, creating large executables. Second, only one communications system can be active at a given time. This limitation is programmed into the Enterprise system. It would be possible to remove this limitation, but it is unlikely that this restriction will prove to be a problem. If multiple communications systems are to be multiplexed, there will be extra work in deciding which system should be used, which is best encapsulated inside the communications manager. Also, our experience seems to indicate that communications software uses a combination of signals and timers that makes it difficult to have more than one system active at a time.

The motivation for this class was the result of a dependence on the Network Multi-Processor library (NMP), a locally developed system [MBS 91]. NMP became an integral part of the original implementation, to the point where its limitations persisted even when PVM replaced it as the communications system. The main limitation in NMP is its static process structure, which spawns and connects all processes at application startup. The encapsulation in the current implementation will make it impossible for such a dependence to occur again, since all types and data required for an implementation can be made opaque.

The only other part of the program that may need to change with different implementations of the communication manager is the main program. It may need to perform some startup procedures that are specific to the communications system. An example of this situation is the Treadmarks library, which uses the fork/join model for creating processes. Thus, the main program is split into sections executed by the master process and sections executed by the spawned children. However, this problem is eliminated by providing multiple main programs and linking in the proper one at compile-time. Other startup operations, such as registration in PVM and region creation in Treadmarks, can be done in the constructor for the communications manager class.

The communications manager is an example of an *Adapter* design pattern, which provides another interface to a library or class. This pattern prevents the choice of the underlying communications system from becoming integrated into the run-time system. By providing a simple interface, we increase the range of potential implementations for this class.

4 Related Work

Since this work is an implementation of a new run-time system created specifically for Enterprise, there is little previous work that is directly related. However, there are other projects that attempt to deal with the different problems of a general run-time system for parallel systems using object-oriented techniques.

The first system is the *Nexus* run-time system, which was built as a compiler target rather than a user-level library [FKT 95]. This system is used in the implementation of the Compositional C++ and FORTRAN M programming languages. Nexus permits the use of multiple communications protocols, providing a single interface but several different protocol modules. The choice of protocol is made at run-time based on a list of communication protocols supported by the receiver, which is received as part of initiating a remote request. This kind of work can be supported in Enterprise using the communications manager, so long as there are no conflicts between the different protocols or their implementations. In fact, this idea will be explored further as distributed shared memory is added to the Enterprise system [Nov 95]. In doing so, we wish to use real shared memory between tightly-coupled processors and distributed shared memory (provided by a library like Treadmarks) for loosely-coupled processors.

The *Concert* run-time system tries to improve the efficiency of concurrent object-oriented programming languages by differentiating between the costs of various operations in the system [KC 95]. To reduce the overall cost of an operation, an optimal implementation for different cases (i.e. local versus remote) is provided and used at run-time. To prevent constant checks to determine the proper implementation, Concert uses *speculative compilation techniques* to *inline* operation invocation to the cheapest version available, based on compiler-generated assertions. If an assertion becomes false, the run-time system is responsible for replacing the inlined operation with a more general, possibly more expensive, version of the operation. In Enterprise, the use of the dispatcher and behaviour pair mirrors this kind of speculative inlining. Each process is initially assigned a function and corresponding behaviour that is cached in the dispatcher, rather than evaluating this information for each request message. If this behaviour becomes incorrect, it can be changed at run-time to a more appropriate one. Enterprise could also apply the same technique with some of the operations it provides by extending the behaviour classes further, allowing the new subclasses to override methods with cheaper versions where possible.

Finally, we consider the *Mentat* run-time system [GWS 93]. Like Enterprise, Mentat uses the futures model of computation and uses compiler technology to insert calls to the run-time code into the user application. The implementation of the system uses a layered approach to isolate system-dependent code, increasing portability. The new Enterprise run-time system takes the same approach with its communications manager, encapsulating it inside a class to remove potential dependencies and to allow multiple implementations. In addition, Mentat has done more work on scheduling, attempting to find efficient ways to find good process-to-processor mappings. Such work is necessary in Mentat since additional processes are created as a Mentat application executes. Enterprise does little in the way of scheduling; processes are placed on processors in a round-robin fashion (subject to any user-specified constraints) at application startup. When dynamic process addition is implemented, a more robust scheduling scheme will be useful.

5. Conclusions

This paper presented the object-oriented components of the new Enterprise run-time system. This new system was written to correct some limitations and deficiencies in the old implementation and to provide a flexible and extensible basis for further development and research. The latter reason was the motivation for the use of an object-oriented design and implementation. The new system was written in C++ to meet efficiency constraints that were not addressed in this paper. The four components described in this paper are the asset graph classes, the dispatcher classes, the behaviour classes, and the communications manager.

The asset graph, a specialization of the *Composite* design pattern, relies on inheritance to separate the characteristics of an asset from the concrete classes that represent the different meta-programming model assets. This separation allows the meta-programming model to change by creating new characteristics and asset types.

The dispatcher and behaviour classes allows for process-dependent behaviour using a specialization of the *Strategy* design pattern. By decoupling the reception of a message from its processing, the new system can modify its behaviour during execution. The behaviour classes also allow new process functions to be created and easily incorporated into Enterprise.

Finally, the new system uses encapsulation to hide the implementation of the communications protocol in Enterprise by using the *Adapter* design pattern. This feature prevents an implementation from becoming integrated into the system and, because of the simple interface, gives us flexibility on the actual implementation of this class.

All of the above was designed and implemented using object-oriented techniques to provide a flexible, extensible base for further research. From the results, it is clear that we have met our four goals. The system consists of about 14,000 lines of C++ code and has been supporting an active group of about 10 researchers for eight months.

Acknowledgments

This research was supported in part by research grants from the Natural Sciences and Engineering Research Council of Canada and a grant from IBM Canada. We would also like to thank other members of the Enterprise team: Ian Parsons, Diego Novillo, Nicholas Kazouris and David Woloschuk for helpful feedback after using the system.

References

[ACDK 96] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations." *IEEE Computer*, Vol. 29, no. 2: 18-28, 1996.

[FKT 95] I. Foster, C. Kesselman, and S. Tuecke, "Nexus: Runtime Support for Task-Parallel Programming Languages." Technical Report ANL/MSC-TM-205, Argonne National Laboratory, February 1995.

[GHJV 93] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design." In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, pgs 406-431, 1993.

[GWS 93] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Processing." Technical Report CS-93-40, University of Virginia, July 1993.

[Hal 85] R. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems* , Vol. 7, no. 4: 501-538, 1985.

[IMMN 95] P. Iglinski, S. MacDonald, C. Morrow, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk, "Enterprise User's Manual, Version 2.4." Technical Report TR 95-02, University of Alberta, January 1995.

[KC93] V. Karamcheti and A. Chien, "Concert - Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware." In *Proceedings of Supercomputing '93*, pgs 598-607, 1993.

[Lob 93] G. Lobe, "The Enterprise User Interface and Program Animation Component." Master's thesis, University of Alberta, 1993.

[LSS 93] G. Lobe, D. Szafron, and J. Schaeffer, "The Object-Oriented Components of the Enterprise Parallel Programming Environment." In *Proceedings of Technology of Object-Oriented Languages and Systems Conference (TOOLS) II*, pgs 215-229, August 1993.

[Mac 95] S. MacDonald, "An Object-Oriented Run-time System for Parallel Programming." Master's thesis, University of Alberta, 1995.

[MBS 91] T. A. Marsland, T. Breitkreutz, and S. Sutphen, "A Network Multi-Processor for Experiments in Parallelism," *Concurrency: Practice and Experience*, Vol. 3, no. 1: 203-219, 1991.

[Nov 95] D. Novillo, "Transparent Shared Memory in Multiprocessor Environments." Ph.D. Candidacy document, September, 1995.

[Par 95] I. Parsons, "Parallel I/O Templates for Enterprise." In *Proceedings of the 1995 CAS Conference, CD-ROM Edition*, 1995.

[SSLI 93] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, "The Enterprise Model for Developing Distributed Applications," *IEEE Parallel and Distributed Technology*, Vol. 1, no. 3: 85-96, 1993.

[Sun 90] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2, no. 4: 315-339, 1990.