University of Alberta

Library Release Form

Name of Author: Steven MacDonald

Title of Thesis: An Object–Oriented Run–time System for Parallel Programming

Degree: Masters of Science

Year this Degree Granted: 1996

. . . .

Steven MacDonald
203–10635 85th Avenue
Edmonton, Alberta
Canada, T6E 2K6

Date: . . . .

University of Alberta

An Object–Oriented Run–time System for Parallel Programming

by

Steven MacDonald

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Masters of Science.

Department of Computing Science

Edmonton, Alberta
Spring 1996

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Object–Oriented Run–time System for Parallel Programming** submitted by Steven MacDonald in partial fulfillment of the requirements for the degree of Masters of Science.

. . . .
Jonathan Schaeffer

. . . .
Duane Szafron

. . . .
Ron Unrau

. . . .
Xiaoling Sun

Date: . . . .

Abstract

Enterprise is an integrated parallel programming system that provides facilities to create, compile, execute, tune, and debug parallel programs on a network of workstations. Enterprise uses two models to completely specify a program. The first model presents an abstraction that relates the different elements of a program to assets in a business enterprise, giving an intuitive method of expressing the parallelism in the program. The second model modifies the semantics of sequential C so that the assets are called remotely. This last model is implemented by preprocessing the user's code to invoke the run–time system, which handles message passing and synchronization.

This thesis proposes that the run–time system can be designed and implemented using object–oriented techniques to improve upon specific aspects of the previous version. Most of these changes fix known limitations or errors in the previous system. The new system is more flexible, allowing the programming models to be changed easily. It can also dynamically incorporate new resources into an executing program. The preprocessing stage has been simplified and, finally, the communications system was abstracted out of the code. These changes, along with several other optimizations, have been implemented and the new and old systems were compared. The results show that the new system has met all of its goals, although some have yet to be validated.

Acknowledgements

First and foremost, I would like to thank my wife–to–be Trudy for her love, patience, and understanding.

Second, I thank my family and by future family–in–law for never once telling me to "get a haircut and get a job." Besides, I finally got the haircut...

Next, of course, I have to thank all of the members of the Enterprise team. First, I have to thank Jonathan Schaeffer and Duane Szafron for providing invaluable feedback during the course of this work. I appreciate the time and effort the expended on my behalf. Next, I am grateful to Randal Kornelson for providing me with my thesis topic, albeit indirectly. Thanks to Ian Parsons, who contributed to this project by providing me with his knowledge of the previous system and by using my system as a starting point for his work. To Diego Novillo, who was the first person to use and correct my code and who learned C++ alongside me. Next, to David R! Woloschuk, because we both suffered the same. To Nicholas Kazouris, who will continue with the work I've started. And finally, to Chris Morrow who, every once in a while, just plain ticked me off.

Of course, there are other people to thank: Mark Brockington for all the amusement; Jonathan Baldwin for the same reason; Bob Beck, Karim Makhani, and Rod Johnson who (possibly unknowingly) help me with my experiments; Gordon Atwood who helped us with a few things we weren't supposed to do; and Darse Billings for being such an easy target. To the rest of you, I didn't forget; I just have to stop somewhere. Thanks to all of you.

Contents

List of Figures

Chapter 1

Introduction

Enterprise is a parallel programming system that allows users to create, compile, execute, tune, and debug parallel programs on a network of workstations [26]. The overall goal of the project is to provide an easy–to–use environment for parallel programming, giving inexperienced parallel programmers the ability to use this software technology to increase the performance of their applications.

Enterprise provides two integrated models for the user. The first is a high–level abstraction relating a parallel program to a business enterprise. This abstraction, called the *meta–programming model*, gives the user an intuitive method of expressing the parallelism in the program. The meta–programming model was found to be a useful and appropriate level of abstraction for users [24].

Enterprise also provides a *programming model* that uses the familiar sequential C model. Using this model, the user is able to create parallel programs quickly, even using existing sequential code where possible. This feature greatly contributes to the usability of Enterprise.

Both models are tied together through the combined efforts of the graphical user interface, the compiler, and the run–time system. The user interface provides a graphical method for the user to specify the parallelization technique for the application, using the meta–programming model. This information is used by the compiler, which preprocesses the user's code so that it invokes the run–time system. The run–time system is responsible for launching an Enterprise application and processing communication, handling any required synchronization.

This thesis speculates that the run–time system of the Enterprise programming environment can be designed and written to satisfy the following goals:

1. To use an object–oriented design and implementation to produce a better system. If this goal is satisfied, a more flexible run–time system will result, which will allow the Enterprise meta–programming model to change. In addition, to use a new method for specifying the run–time behaviour of an Enterprise program. This method would use object–oriented techniques to make a flexible and extendible set of behaviours to accommodate any desired result. The new system would be implemented in C++ [27] and replace the C version implemented by Randal Kornelson.

2. To allow the run–time system to exhibit dynamic behaviour during its execution by creating and using new processes. The previous run–time system could not implement this functionality because a static process structure was enforced by the communications system. The new system should be able to dynamically include resources into an executing program.

3. To eliminate the code generated by the preprocessor. This goal will cut compilation times by shortening the preprocessing stage and reducing the amount of code to be compiled. Since this code is executed by the run–time system, it is considered in this thesis.

4. To create a small communications interface. By creating a small, well–defined interface, the run–time system will be able to implement its communication primitives using any available system that provides the necessary functionality. As well, if the implementation is properly encapsulated, then the run–time system should be independent of any specific implementation, which was a problem in the previous implementation of the run–time system.

The success of this thesis rests on achieving these goals while not causing performance losses in Enterprise applications. This last aspect is crucial; if all goals are met but the resulting implementation makes Enterprise unusable, then the project is a failure. Having little or no experience with C++, there is no way of knowing if the performance of Enterprise would suffer because of the run–time overhead of an object–oriented language. Although C++ is designed to minimize this overhead, it still exists and may need to be addressed [27].

1.1 Motivation

At best, parallel programming is a difficult task, combining the normal complexities of sequential programming with issues such as synchronization, nondeterministic program behaviour, and increased risk of errors from the hardware environment. However, increased demands for computing power have made this a viable solution for a large range of problems. Users can solve larger problems and expect results in less time than before.

Unfortunately, few of these users can afford the large supercomputers normally associated with high performance computing. Instead, they are looking at networks of workstations as a cost-effective method of providing supercomputer performance. Many organizations already have this equipment in place, and only need a method of harnessing it to increase performance.

So far, there have been two opposing philosophies on how to write parallel programs. These two factions typically represent the experienced users who are willing and able to deal with the complexities of parallel programming and the neophytes who are (justifiably) daunted by the task. The first group typically believes in using low–level code libraries to maximize performance while the other believes in the high–level abstractions provided by parallel programming environments to minimize the learning curve.

Code libraries such as PVM [28] provide the programmer with a set of procedures that allow processes on different machines to communicate via message passing. The programmer uses these to produce a parallel program. This system has the advantage of flexibility; the programmer can structure the application in any manner, giving the experienced programmer invaluable freedom. The disadvantage of this approach is that the procedures provided are low–level, forcing the programmer to explicitly handle communication, synchronization, and other issues in their code. As well, the programmer usually embeds the parallel structure of the application into the program. If this structure must be changed to improve performance, it can mean significant code changes. Overall, this approach leads to large applications with complex code, increasing the risk of programming errors.

Parallel programming environments such as Enterprise and HeNCE [3] present an abstract model of parallel programming to the programmer. These environments differ in the model provided, the available tool–set, and the user interface. The advantage of such systems is the abstraction. The programmer can concentrate on the parallel aspects of the program at a high level and need not be concerned with low–level details. The tool–set may also be helpful, although this aspect is dependent on the environment used. The disadvantages of this approach can also involve the programming model used. If the model is not intuitive, just writing a program that executes correctly can be difficult. Regardless, there will be a learning curve for new users. Also, the program structure used in the application must be supported by the model; some structures may be difficult or impossible to implement. Finally, the loss of flexibility in the programming model results in potential performance losses.

A common problem that appears regardless of the approach chosen is creating a correct parallel program. Debugging a parallel application creates a special challenge because of issues like nondeterministic execution. Errors may appear only on certain conditions that do not occur every time the application executes. Also, the effect of any probing by a debugger or the influence of adding extra statements to the code can change the execution pattern and prevent the error from appearing [13]. These problems are compounded by a lack of robust tools.

Once a correct program is written and debugged, the programmer must revisit the code for performance debugging, tuning the program until the desired level of performance is reached. This activity is usually done without the benefit of any tools or methodologies. Without these tools, it is easy to introduce new errors into a program, meaning the correctness must be verified after performance debugging is finished.

Enterprise was developed to remedy the lack of parallel programming tools. It is a parallel programming environment that provides a graphical interface to its facilities. The model is comprised of two abstractions, the meta–programming model and the programming model, that cover the expression of parallelism and the semantics of user code. Parallel code is inserted at compile–time, making the application code independent of the communication structure. This feature allows for quick changes to the program structure with minimal effort and, combined with the use of familiar C code, minimizes the risk of introducing new errors into the application. Enterprise also supports facilities for debugging user code, with the ability to run a controlled execution to closely duplicate a previous execution pattern [13].

The main force driving an Enterprise application is the run–time system, which is responsible for spawning the application on the desired set of workstations and ensuring the parallelization structure is properly enforced. Obviously, the run–time system must be capable of supporting any application structure allowed within the meta–programming model. However, there must be room for change to allow the model to expand and grow, based on the experiences and requirements of users. The run–time system must also be portable to support heterogeneity, which requires that code be as independent of computer and operating system architecture as possible. Finally, as with all large programs, the design of the software is important to facilitate maintenance.

It is for these reasons that the Enterprise run–time system was rewritten. The existing code was inflexible and failed to meet all of the above requirements. Changes to the model and the underlying process structure were becoming increasingly difficult, and there were some dependencies on earlier communication library choices. In addition, several earlier design decisions proved to be unsuitable for the system, creating performance problems that needed to be rectified.

The choice of an object–oriented language arose from the correspondence between the meta–programming model and object–oriented constructs. In addition, object–oriented and distributed software systems have considerable conceptual overlap, particularly the idea of a strong, well–defined interface. Finally, several compiled hybrid languages such as C++ have emerged. These languages do not have the high run–time overhead normally associated with object–oriented languages, making their use in systems programming more feasible. This has been demonstrated by the Mentat [10] and μC++ [4] systems.

1.2 Obtaining the Enterprise Distribution

The Enterprise system can be downloaded via `ftp` or the World Wide Web. The distribution includes the user manual, source code for the user interface, executables for supported architectures (SUN4, SGI, RS6000), and some example programs. The user interface requires VisualWorks 2.0 [1] [16]. The site is `ftp://ftp.cs.ualberta.ca/pub/Enterprise`.

1.3 Thesis Organization

This thesis consists of five chapters. The second chapter explains the programming models for Enterprise, including an example. Chapter three is a detailed discussion of the important aspects of the new design. Chapter four presents some results, justifying that the design goals have been met. The fifth chapter concludes the thesis, giving a summary and outlining several outstanding research items.

[1] VisualWorks is a trademark of ParcPlace Systems.

Chapter 2

An Introduction to Enterprise

As with other parallel programming environments such as HeNCE, Enterprise provides a high–level abstraction to simplify parallel programming. Unlike most models, Enterprise has two different programming models: the *programming model* to specify the semantics of user code and the *meta–programming model* to express the parallelism in the application.

This chapter provides an overview of the Enterprise programming models, including an example application. The details are not complete and do not delve into interface details; the interested reader is referred to the Enterprise user manual [14].

2.1 The Programming Model

The Enterprise *programming model* specifies the semantics of user code, which is written in C. In sequential code, a program consists of a hierarchical set of function calls. When a function is invoked, the caller is suspended until the completion of the called function. In an Enterprise program, some of these function calls are made in parallel, executing on a separate process on another processor that is specifically responsible for that function call. This section describes the semantics of parallel calls in Enterprise, which look almost identical to normal C invocations. This discussion includes calling another process and receiving the results.

```
future = parallel_call() ;
    .
    .
/* Work continues, with parallel_call() */
/* in parallel.  */
    .
    .
/* Resolve future on access.  */
/* Block if results unavailable.  */
x = future + 1 ;
```

Figure 2.1: Example of a future call.

Enterprise preserves as many of the semantics of sequential C as possible. Some of the semantics are violated for efficiency reasons. This goal was achieved using the *future* model [12]. The future model is based on tagged variables that represent the results of parallel function calls. The user invokes a function, which may be a normal sequential call or a parallel call that is transformed into a message to a remote process. The method of specifying which functions are parallel is up to each programming environment, and is done using the *meta–programming model* in Enterprise (Section 2.2). However, the current process continues executing, with the evaluation of the future taking place in parallel, until the tagged variable is accessed. At this time, the future must be *resolved*, meaning the process must now wait for the results of the parallel call before it can continue. If results are not available, the process blocks until they are ready. For an example, see Figure 2.1.

The future model also allows for a *purely asynchronous call*, which is a parallel call with no results to be returned and hence no generated futures. These calls execute concurrently with the calling process. A common example of this kind of call is a parallel procedure that takes a point and draws it on a window, which represents the output of the program. Since no results are returned to the calling process, it may be invoked without blocking, making it an ideal output process.

The Enterprise compiler is responsible for recognizing remote calls and future accesses. It also inserts code to create a message from the function arguments (mar-

```
/* Marshals the arguments and invokes asset.  */
_ENT_SendCall_parallel_call(&future) ;
    .
    .
/* Other user code.  */
    .
    .
/* Resolve future.  */
/* Block if results unavailable.  */
_ENT_Wait(&future) ;
x = future + 1 ;
```

Figure 2.2: Simplified post–processed code.

shaling data) and extracts the arguments from a message (unmarshaling data). This code is necessary because the run–time system does not know the argument lists of the parallel function calls in the application. Figure 2.2 shows simplified post–processed code from Figure 2.1. The `_ENT_SendCall_parallel_call()` procedure is generated by the compiler, and is not shown. The `_ENT_Wait()` call is implemented by the run–time system.

There is only one obvious break with sequential semantics in Enterprise, which is necessary because of a limitation in C. To marshal data referenced by a pointer, the size of the data must be provided by the user. This is only required for remote calls. As well, the normal semantics of pointer variables have been changed for reasons given in Section 2.3.

The greatest benefit of this programming model is its ease–of–use, since it preserves most of the semantics of C code. Many programmers are already able to program in C, so they will be able to start programming quickly. In addition, existing code can be easily incorporated into an Enterprise application.

2.2 The Meta–programming Model

The *meta–programming model* allows a programmer to specify the parallelism in an application. The meta–program is a hierarchical combination of *assets*, which relate the user program to a business enterprise. It is described by an *asset graph*, which is created in the graphical user interface.

Unlike systems such as HeNCE, this asset graph expresses both the connection pattern of the different processes and the parallelization technique the user wishes to apply. This expression is achieved by connecting assets within the constraints of a set of provided templates, which are assets themselves. The user cannot arbitrarily connect the processes, but can combine the templates to create larger, more complex program structures. Our templates reflect several common parallelization techniques traditionally used in coarse–grained parallel programs.

Another important aspect of the templates is that they preserve the flow of control in the program. In the user code, an asset call must also be a valid sequential procedure call within the constraints of the template. Thus, one may also think of the asset graph as a call graph for the application, which can actually be derived from analysis of the code [22].

The meta–programming model gives a high–level view of the parallelism, relieving the user of issues like message passing and synchronization which are attributes of the assets. Since the compiler uses the asset graph to insert the parallel code, the structure of the program can be modified without changing the code. The programmer can create new program structures quickly, making performance debugging significantly easier.

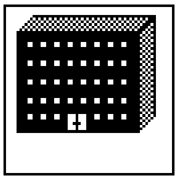
The only limitation of the meta–programming model is its hierarchical nature; the flow of control is always downward. This is characteristic of the execution of a sequential program, which Enterprise retains in its programming model. In addition, deadlock is prevented within an Enterprise application because no cycles can develop in the communication graph.

There are two different groups of assets: *composite assets* and *singular assets*.

Composite assets are the templates for parallelism, containing other assets (both composite and singular) and specifying the communication structure. They contain no code themselves. Singular assets are akin to processes executing a function call. They contain user code that is invoked, and can in turn call other assets and return results to the caller. The name of the executed procedure is the same as the asset name.

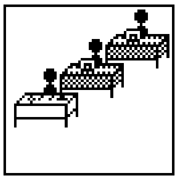
The following sections describe the different asset types and their uses. It begins with a description of the composite assets followed by the singular ones.

2.2.1 Enterprise Asset



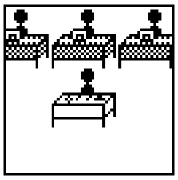
The *enterprise asset* is analogous to a business enterprise, and represents the entire user application. All Enterprise programs start with an enterprise asset, which initially contains a single individual asset, which is the entry point to the user's program.

2.2.2 Line



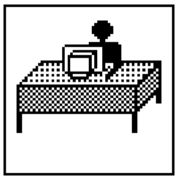
A *line* is analogous to an assembly line, usually referred to as a pipeline in the literature. It consists of an ordered set of assets that contains a receptionist and a fixed number of heterogeneous assets. Each asset takes the results of the previous asset and continues processing, finishing by giving the refined data to the next asset. Each call to the line waits for the receptionist to finish processing the previous call, not for the entire line to finish. Each asset can only call the next asset in the line, with the last asset able to call out of the line. Again, the assets in the line can be either composite or singular.

2.2.3 Department



A *department* is similar to a department in an organization. It contains a receptionist and a fixed number of heterogeneous assets. The receptionist directs requests to the appropriate asset, which can be either composite or singular. Each call to a department must wait until the receptionist finishes processing. Every asset within a department is able to call out to the next asset if it is contained within a line, but cannot call other assets within the same department.

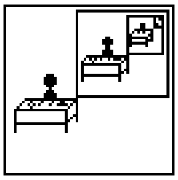
2.2.4 Individual



An *individual* is analogous to a person in an organization. It is equivalent to a process and contains a procedure whose name is the same as the asset. When invoked, this procedure is run to completion; subsequent calls to the asset must wait for the previous call to finish. An individual may call other assets subject to the constraints of its *owner asset*, the composite asset in which it is contained.

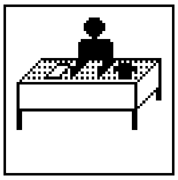
In creating an asset graph, it is important to note that an individual can be replaced with any composite or a division asset. This operation is called *coercion*, and allows assets to be combined to form complex graphs.

2.2.5 Division



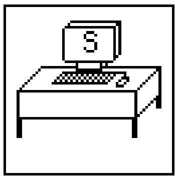
A *division* is a special instance of an individual that is allowed to make recursive calls. A division is specified using replication factors and an additional depth, specifying the point where recursive calls are made sequentially, the *leaves* of the division. Non–leaf nodes generate requests that are sent to the manager of the division, which sends them to a free division worker. In both cases, non–recursive asset calls will always be messages. A division may be coerced into an individual asset.

2.2.6 Receptionist



A *receptionist* is another special instance of an individual that acts as the entry point to composite assets. The exact behaviour of the receptionist is dependent on its owner asset. Because of its function, the receptionist cannot be replicated, although the owner asset may be. As well, a receptionist cannot be coerced to any other asset. It is important to note that the receptionist has the same name as its owner asset.

2.2.7 Service



A *service* is analogous to a business asset that can be accessed freely and is not consumed, such as a clock. All assets are allowed to call a service, but a service cannot call any assets. It can be used to provide expensive shared memory. A service cannot be coerced to any other asset.

2.3 Optimizations

The different asset types are not always sufficient to create a program that achieves the best performance. The user is given other optimizations that may be applied to help. They are optional because they can break C semantics, requiring the programmer to understand their effects and compensate for them in the code. These options are directional parameters, replication, unordered execution, and impatience.

Directional parameters allows the user to have greater control over how pointer data is handled in an asset call. Normal C semantics stipulate that pointer data is passed by reference. In an asset call, this entails sending the data to the remote asset and back again. For large amounts of data, this is clearly undesirable. Thus, when the user indicates the size of the data, the direction of the data can also be specified. The options are:

OUT: The data is not sent to the called asset, but is returned only.

IN: The data is sent to the called asset, but is not returned.

INOUT: The data is sent both ways, preserving C semantics.

The default direction for pointer data is `IN` for efficiency reasons. It is important to note that this does not preserve normal C semantics; `INOUT` does.

Replication allows the user to create multiple copies of the specified asset, each operating in parallel. The assets that may be replicated are the department, line, division, and individual. Note, however, that the enterprise asset cannot be replicated. This means that the outermost template, the asset that represents the structure of the entire program, cannot be replicated.

Replication is specified as a minimum and maximum value. The old run–time system uses the maximum value. In the future, using the new implementation, the number of replicas will be dynamic, varying between the minimum and maximum value as necessary. This enhancement will tailor a computation to the available resources and will ensure that only those actually used will be kept.

Unordered execution applies to the order in which reply messages from replicated assets are processed by the caller. The default is to match the future with the proper reply message from the generated request. However, if the order of replies is unimportant, the user can specify unordered semantics. This optimization can be useful if futures are being used in a transitive operation, such as addition. When results are accessed, the run–time system uses the first compatible future it finds. Two futures are considered compatible if:

1. they are results from different calls to the same asset.

2. they contain the same data (combination of return code and arguments passed by reference). The arguments that are passed by reference need to be checked to ensure that the results from the call

   ```
   parallelFunction(array1, INOUT(10), array2, IN(10)) ;
   ```

   are not confused with the results from the call

```
parallelFunction(array1, IN(10), array2, INOUT(10)) ;
```

3. the pass–by–reference arguments are the same size.

Impatience refers to the treatment of outstanding futures that have not been resolved when an asset finishes executing. The default is for the asset to resolve all futures before the asset returns, ensuring consistent data. If these futures are not important, the user can specify that the asset ignore them and return immediately. This optimization implements *eureka jumps*, particularly useful where a search has found the desired valued and wishes to stop immediately. This option is not available in the old run–time system; all futures must be resolved before the caller finishes. The new system provides a mechanism to implement different return strategies. Currently, the caller returns immediately and ignores reply messages from previous invocations, but the called assets must finish their work before they can be called again. The next step is to interrupt the called assets to stop them from processing unwanted work.

2.4 The CubeSquare Example

In this section, a simple Enterprise application program is shown to demonstrate the programming model. Our example is called `CubeSquare`, a simple program that sums the squares and cubes of integers 0 through 9. Please note that this is only a toy program; it does not do enough work to warrant a parallel implementation. However, it is a good example program to demonstrate the programming model.

The asset graph for `CubeSquare` is given in Figure 2.3. The code for the receptionist, `CubeSquare`, is shown in Figure 2.4. `Cube` and `Square` simply return the cube and square of the argument, and are not shown.

The asset graph consists of a department with two components. If we examine the code for `CubeSquare`, we see that it invokes both `Cube` and `Square`, which cannot invoke each other. Thus, the asset graph mirrors the calling patterns of the program. In addition, only functions and procedures that are specified in the asset graph are made into parallel calls; `printf()` is not part of the graph so it is a sequential call.

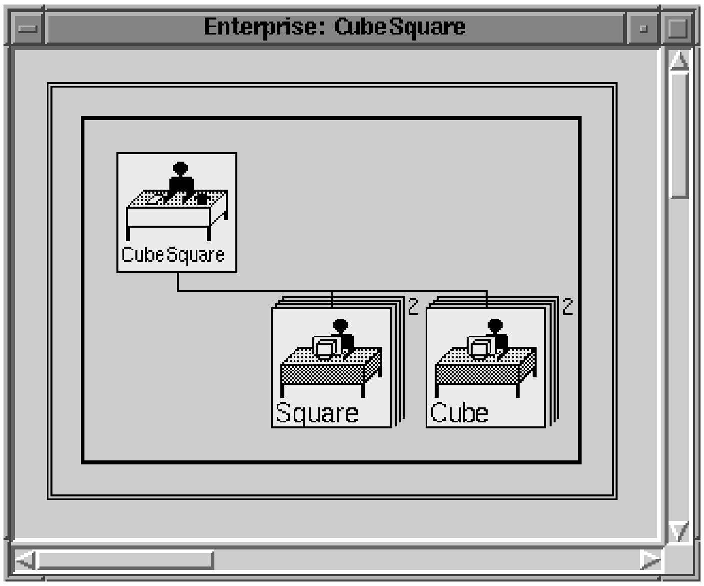


Figure 2.3: The asset graph for CubeSquare.

```
CubeSquare(int argc, char **argv)
{
    int i, sq, cu ;
    int a[10], b[10] ;

    for(i = 0;i < 10;++i) {
        a[i] = Square(i) ;
        b[i] = Cube(i) ;
    } /* for */

    sq = cu = 0 ;
    for(i = 0;i < 10;++i) {
        sq += a[i] ;
        cu += b[i] ;
    } /* for */
    printf(``Sum of squares %d\n'', sq) ;
    printf(``Sum of cubes %d\n'', cu) ;
} /* CubeSquare */
```

Figure 2.4: The code for the CubeSquare asset.

```
a[0] = 1
b[0] = 8
a[1] = 81
b[1] = 0
...
```

Figure 2.5: A partial execution of CubeSquare with unordered semantics.

In addition, the `Cube` and `Square` assets have been replicated twice, as shown in the graph, and have been set as unordered assets (not shown). By creating multiple instances of an asset, multiple invocations of that asset can be done in parallel, hopefully speeding up the program. Also, since the program sums up numbers and addition is transitive, the two assets are set to unordered. Thus, `CubeSquare` can process the replies in the order they arrive rather than enforcing the order.

The code for `CubeSquare` gives a concrete example of the programming model, which uses standard sequential C. This example also shows how the futures model can be used to prevent unnecessary blocking. The results from calls to `Cube` and `Square` are stored in arrays rather than directly summing them up. If a single sum variable is used, successive calls to the assets would be blocked until the previous one returned since the sum variable would be a future for each call. Since our results go to different memory locations, each is independent and can proceed in parallel.

The second loop is used to sum the results by resolving each of the calls. If the assets were ordered, the results are guaranteed to be processed in order, so that a[i] would be the square of i and b[i] would be the cube of i. Unordered calls remove this constraint and can result in a faster program. An unordered execution can result in the partial execution shown in Figure 2.5, where a[i] is now the square of some other number j and b[i] is the cube of another number k. Note that unordered semantics apply to each asset, so that even the pairs of invocations may not necessarily match.

This example demonstrates how the meta–programming model and programming model are used in Enterprise to produce programs. It provides a concrete example of how the asset graph reflects the program structure and how it is used to specify the run–time behaviour of function calls. In addition, the programming model shows how the semantics of sequential C have been changed to use the futures model of computation.

2.5 Summary

This chapter introduced the Enterprise parallel programming environment. The two programming models, the programming model and the meta–programming model, were discussed. Other optimizations on top of the programming models were also introduced. Finally, an example was provided to demonstrate how the models work.

Chapter 3

The Design of the New Run–time System for Enterprise

An Enterprise application consists of three layers: the user application code, the run–time system, and the communications manager (Figure 3.1). The application code is the sequential program written by the user. The compiler inserts additional code into the application, including calls to the run–time layer, based on the asset graph. The run–time system launches the processes for the application and performs all communication via message passing. The protocols for these messages are implemented in the communications manager. The communications manager currently uses the PVM message passing library system [28], but alternative implementations can be provided, such as NMP [21].

This chapter elaborates on the responsibilities of the run–time system and discusses related work on run–time systems. Next, the motivation for the re–implementation is presented and the design of the new version of the Enterprise run–time system is explained in detail. The main design details covered are the inheritance hierarchy of the new system, the internal representation of the asset graph, the launching of an application program, the process information stored in the graph, and the execution of a program. Finally, some of the problems encountered during the design phase and some other interesting concepts used during implementation are

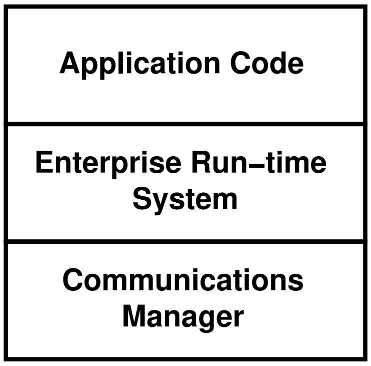


Figure 3.1: The layers of an Enterprise program.

discussed.

3.1 Responsibilities of the Run–time System

The Enterprise run–time system is responsible for the proper execution of an application program. The system is a library that provides general–purpose parallel code that, coupled with the compiler–inserted code, executes a parallel application within the bounds of the programming model. Specifically, the run–time system has to:

1. implement compiler–inserted function calls. The first duty of the run–time system is to provide the implementation for the interface between itself and the post–compiled code (the first two layers in Figure 3.1).

2. launch the application. The run–time system has to create processes for the assets in an application. This launch procedure is done in a distributed fashion, with each process responsible for launching processes for the assets it can invoke. Each process determines which processes it needs to create based on the function of the process and the asset graph, which is a part of the run–time system.

3. process messages. The system is responsible for all message processing in an application. This processing includes sending requests, processing requests, generating reply messages, and any necessary synchronization. In addition, the

run–time system uses the asset graph to make consistency checks, verifying that messages are only sent between communicating processes. This item will be especially important as Enterprise becomes more dynamic, with processes changing the asset they are executing and spawning additional workers at run–time.

4. manage futures. The run–time system has to keep track of futures to properly process reply messages and synchronize when the data is accessed.

5. gather run–time information for program animation and replay. These *events* are logged with the Enterprise process where they are written out to disk.

6. shut down the application. Once the application has finished executing, the processes need to be removed so they don't use system resources after a program finishes.

3.2 Related Work

Since the result of this thesis is an implementation of a run–time system created specifically for Enterprise, there is little previous work that is directly relevant. There exist other projects that attempt to deal with the different problems of a general run–time system for parallel programming. Several of these projects are examined, with an emphasis on the parallel model and different features provided by the system. If required, the language or programming system the run–time system was designed for is also described. The discussion of the different features does not generally include those that are common to all run–time systems unless they are particularly interesting. Features common to all of these system include a name space to locate objects, communication, delivery of remote requests, and processor scheduling.

3.2.1 Parallel Virtual Machine

Parallel Virtual Machine (PVM) [28] is a library system that provides the programmer with a set of primitives that can be used to build a parallel program. PVM is not useful as a compiler target because of the difficulty in pairing up the calls to send and receive in the different processes in an application. It provides facilities for process creation, communication, and synchronization. The programmer uses these primitives to build arbitrary programs without the benefit of any abstraction or high–level model.

The main benefit of PVM for other run–time systems is that it can be used to implement the communication protocols. Since PVM has no model of computation itself, the higher–level system can provide all the abstraction with little restriction.

3.2.2 Nexus

Nexus is a general–purpose run–time system that is intended as a compiler target rather than a user–level library [8]. It is used in the implementation of the Compositional C++ and FORTRAN M programming languages. Nexus provides a task–parallel computation model where multiple unrelated threads can execute on a set of *contexts*, which are the different executables being used in a program. Each context may have multiple threads executing in it and is mapped to a *node* or processor for its lifetime. Multiple contexts can be mapped to a given node.

Nexus also provides global pointers that can point to an address within a context. These pointers are valid in other contexts and are used by Nexus itself to send *remote service requests* between threads in different contexts. The thread generating the request does not block and receives no acknowledgement or reply. Replies are generated at the called thread by making another remote service request to the caller.

The most distinguishing feature of Nexus is the ability to use multiple communications protocols. Nexus provides one interface for communications but several different protocol modules that can be used. The decision for the most efficient protocol is made at run–time using a list of communication protocols supported by a remote context. This list is attached to the global pointer to the remote context, which is obtained as part of generating a remote service request.

3.2.3 Concert

The Concert run–time system tries to improve the efficiency of concurrent object–oriented programming languages [17]. It attempts to distinguish between the different costs of the various cases for each operation in the system. To reduce the cost of the system, Concert provides different implementations of run–time operations that are optimized for a specific case and chooses the optimal version of an operation at run–time. The most common difference between these cases is local versus remote operations, but there are other distinct cases for some operations.

Concert also uses *speculative compilation techniques* to try to bypass run–time checks of object properties, *inlining* the call to the cheapest run–time operation based on compiler–generated assertions. If the assertion becomes false, the run–time system is responsible for replacing the inlined operation with a more general (and usually expensive) version. Since replacing an invalid assertion can be costly, speculative compilation must be used with care.

3.2.4 Mentat

The Mentat run–time system [11] is the compiler target for the Mentat programming language [10]. The programming language is an extension of C++, with additional keywords to indicate that instances of certain classes (*Mentat objects*) are to be executed in parallel. Mentat provides a dataflow programming model, where a program is represented by *nodes* and *directed arcs*. A node has user code associated with it, with corresponds to a user function. Outgoing arcs from a node indicate data is passed from that node to another. Incoming arcs indicate data that is to be received by a node. A node executes when information is available at all of its incoming arcs.

The programming model used within the nodes is based on the *futures model*, similar to Enterprise. Unlike Enterprise, a future can be passed between Mentat objects without being resolved. As long as the future is used as an argument to another Mentat object, it does not need to be resolved. The future only needs to be resolved when it is used in a strict expression. Mentat uses compiler technology to replace invocations of Mentat objects by calls into their run–time system and resolve futures when necessary.

The implementation of the Mentat run–time system uses a layered approach to isolate system–dependent code. This approach improves portability since a small portion of the system will require modification for different architectures.

The main limitation identified by Grimshaw et al. was in the scheduling algorithm. Each machine uses its local load to determine if it will accept a new instance of a Mentat object for execution. If the load is above a specified threshold, the Mentat object is transferred to another host. After a specified number of transfers, the receiving host must execute the new object regardless of its load. This algorithm fails to take into account the weight of the new object (I/O bound versus computationally intense) and whether the object is in the critical path for the application. If a new object is a heavy one and on the critical path, additional effort should be expended to try to place it on an idle node. For non–critical or light objects this effort can outweigh the potential benefits, so a less expensive scheduling algorithm can be used.

3.2.5 Chares

The Chare system provides a relatively low–level system that abstracts out resource allocation and provides a machine–independent method of specifying the parallelism in a program [7]. The programmer is required to create chares and specify the communication between them. For this reason, Chares is a user–level system and not a compiler target.

Chares consist of a name, local variables, and a set of entry points. Each entry point is a function name with a single argument, a pointer to a user–defined message type. Each entry point has a block of C code associated with it, which processes the supplied message. Variables local to the chare can be accessed within this block. A set of C functions provide facilities for creating and communicating with other chares.

There are two different types of chares in the system: user chares and branch office chares (BOCs). A BOC consists of a manager and branch chare, which execute on each processor. There are separate functions to communicate with BOCs, which all have the same chare ID regardless of the machine they are executing on. Among their responsibilities is dynamic load balancing. When a new chare is to be scheduled, the BOC responsible for its creation finds its least–loaded neighbour machine. To help this decision, the BOCs periodically exchange load information. This exchange can also be used by the BOCs to perform load balancing when a neighbour's load falls below or rises above a threshold.

Chares also provides a shared memory using abstract data types that also specify access patterns. This shared memory is used to make communication more efficient. Data only needs to be packed into a message if it is being transmitted between processors. The packing and unpacking routines have to be specified by the programmer and are invoked by the run–time system. If the two communicating chares share an address space, this packing is unnecessary.

3.2.6 Paralex

Paralex is another example of a dataflow system [2]. It provides a graphical user interface for the user to create a set of nodes and arcs. Inside of each node in a Paralex program executes a sequential C function. Stubs functions are generated by a precompiler to create a parallel program.

Paralex achieves both fault tolerance and load balancing using ISIS process groups. The programmer can specify the number of processes executing each node. One of these processes is elected as the *co–ordinator*, while the others are inactive backups. For fault tolerant nodes, if the co–ordinator stops executing then another is elected from the backups. Load balancing is implemented by forcing an election if the load on the co–ordinator's processor get too high. Since each process in the group receives all messages, this change is transparent to the rest of the program.

Paralex also supports distributed debugging using a trace file. The execution is controlled using a controller process, which has an outgoing arc to every node in the application. Since a node can only fire when data is on all incoming arcs, the controller can be used to precisely control the application.

3.2.7 Chores

Chores is an object–oriented system that was written to be used by programmers or as a compiler target [5]. It was targeted for uniform memory access shared memory machines, although the authors believe the system generalizes. It attempts to handle granularity, scheduling, and synchronization at run–time rather than leaving these issues to the programmer or compiler. Chores express both functional and data parallelism (specifically, loop–based parallelism) using the same constructs.

The main abstraction is the *chore*, a structure that consists of several items of work or *atoms* that are executed by threads. As an example, a chore might be multiplying two matrices together. The atoms inside this chore would be computing the value of each individual element in the resulting matrix. A chore is specified with the chore body (the function that is executed by the chore), the arguments of the body, an optional set of precedence constraints, and an optional list of ranges for some of the arguments.

There are several different types of chores: indexed chores, dynamic chores, and sequential chores. These chores differ in the data specified when the chore is created and how atoms are added during execution. An indexed chores specifies all of its atoms at compile–time by specifying ranges for one or more of its arguments. No additional atoms can be added during execution. Dynamic chores do not specify any ranges, but can add additional atoms as the chore body executes. Both of these chores implement data parallelism. Functional parallelism is achieved using the sequential chore, which has only one atom. A sequential chore does not specify ranges for any of its arguments or add atoms in the body. These chores can be used to create threads executing in different chore bodies simultaneously.

Indexed chores can be subdivided into *independent*, *precedence–constrained*, and *initiation ordered*. The atoms of independent chores can be executed in any order. However, some algorithms have data dependencies that must be satisfied. The atoms of these kinds of chores are either precedence–constrained or initiation ordered. In indexed chores, each execution of the chore body will cause other atoms to be freed for possible execution. The difference between precedence–constrained and initiation–ordered chores is the order of execution of the freed atoms. In a precedence–ordered chore, the new atoms may be executed in any order. This is equivalent to a dynamic chore that adds new atoms as it executes. An initiation ordered chore always executes its atoms in the order defined by the supplied ranges.

To reduce the cost of communication, the atoms of a chore are sent to requesting threads in blocks rather than individually. A chores can refine its granularity of a thread by changing the size of the block a thread receives. Most of the other run–time operations in the Chores system are specifically optimized for the common case, with more expensive alternatives for other cases. When there is no common case for an operation (atom addition, for example), the operation is optimized to ensure efficiency.

3.3 Motivation for Re–implementation

The previous run–time system was written circa 1992 to rectify performance problems uncovered by Parsons [24]. Many of these problems concerned the use of ISIS [15] as the communication system. This was replaced by a communication library called NMP, developed at the University of Alberta [21].

The largest problem with this version of the run–time system was the ad–hoc way it was maintained. New code was written to meet the new requirements as they emerged, with little concern for keeping a consistent and coherent design. The system became large blocks of complex code that were difficult to modify. Only the few remaining people who have written the code were able to maintain it effectively. This problem was a concern when several design decisions regarding the process structure of Enterprise applications were re–evaluated. It became clear that the older system was heavily tied to these earlier decisions, exacerbating the problem of modifying the code.

This maintenance policy also introduced a dependency on NMP-based constructs, such as the use of its process identifiers to index the processes and the connection file (a file used by NMP to identify all processes for an application and a matrix specifying process connections). The connection file forced a static process structure for Enterprise applications, which became a requirement of the run–time system. When PVM was substituted as the communications system, many of these anachronisms were still necessary because they were an integral part of the system. Since NMP did not support dynamic process creation[1], the run–time system was unable to use this feature. The re–write of the code attempts to remove such inflexibilities by encapsulating the communications system into a class with a generic interface, limiting the scope of any potential dependencies. The new system also provides the ability to spawn new workers at run–time and incorporate them into the computation. The minimum/maximum replication factors specify the limits on the number of replicas, and the available system resources will determine the actual number.

Another goal in the re–implementation was to reduce the amount of *stubs code*, the code inserted into the user code at compile–time to parallelize the application. The older version of the run–time system generated a large amount of stubs code, adding about 37 kilobytes of code to a typical asset source file. This code was initially intended to marshal and unmarshal the arguments of a remote call into a message, but this role expanded to include separating and interpreting the header information in the message and generating replies. In the new version of the code, many of these functions were successfully moved into the run–time library, creating a small, clean interface between the two components and reducing the amount of generated code.

Functions for parsing and manipulating the graph file were replicated in various

[1]Subsequent releases of NMP will include this functionality.

Enterprise utilities, such as the compiler and run–time system. One result of the re–implementation is a library that specifically deals with the graph, allowing it to be read and manipulated. The library reduces the amount of redundant code in the Enterprise system, which allows the graph file format to be changed easily. This feature is particularly important as the format was changed during this research and future changes are planned.

Finally, using an object–oriented language in the re–implementation clarified the design of the run–time system. The previous implementation used C structures to represent the asset graph but contained a full set of attributes, even though the specific asset represented may not need them. Using a superset of attributes made the behaviour of a particular asset difficult to express cleanly, with a variety of flags used to specify the asset type, its position within a composite asset, and its function within the graph. The new C++ code mirrors the actual structure of the meta–program of an Enterprise program. By representing each asset type as a class, the behaviour of each is encapsulated cleanly. The use of multiple inheritance allows a more precise specification of the attributes of the different assets. Both of these elements will make adding new asset types a straightforward process, only requiring that certain methods be written to express the desired behaviour. If new assets can be subclassed from existing ones, the effort is further reduced.

As well, the use of *behaviour classes* allows the run–time characteristics of a process to be expressed cleanly. The earlier implementation based run–time decisions on a set of flags included in each message. Now, a process creates an instance of a behaviour class, which models the responsibilities of the process and automatically processes messages correctly. Finally, the new system uses an instance of a *dispatcher class* to receive messages and forward them to the behaviour. This additional indirection makes it possible to change the instance of the behaviour at run–time, allowing a more dynamic system to be created.

To summarize, the previous implementation of the Enterprise run–time system was static, inflexible and severely limited by early design decisions. The new design remedies these problems, creating a cleaner design and better code. In addition, the new system allows an Enterprise program to be run over a wide–area network and supports dynamic process addition, two features that were not possible before.

3.4 The Design of the Object–Oriented Run–Time System

This section gives high–level details of the design of the key aspects of the new run–time system. The topics of this section are the class hierarchy for the new run–time system, which is compared to that used for the user interface; the internal representation of the asset graph and how it is be manipulated; the launching of an Enterprise application, including the process structure; details on how the process information is stored and used; and finally the execution of a program.

3.4.1 Class Hierarchy

In this section, the asset class hierarchy designed for the new run–time system (Figure 3.2) is compared to that implemented by Lobe for the graphical user interface[20, 19] (Figure 3.3). The two hierarchies attempt to represent the same class structure in an object–oriented way, although each is used by a different component of Enterprise. By comparing the two representations, some of the benefits of the new design are clearer. Since the old version of the run–time system was not written in an object–based manner, there is no good basis for comparing its design to the new class hierarchy.

For clarity, the hierarchy for the interface has been factored to remove any classes specific to the interface. A black square in the upper left corner of a class indicates an abstract superclass.

The leaves of the hierarchy diagram correspond directly to their counterparts in an application program, which are described in the meta–programming model

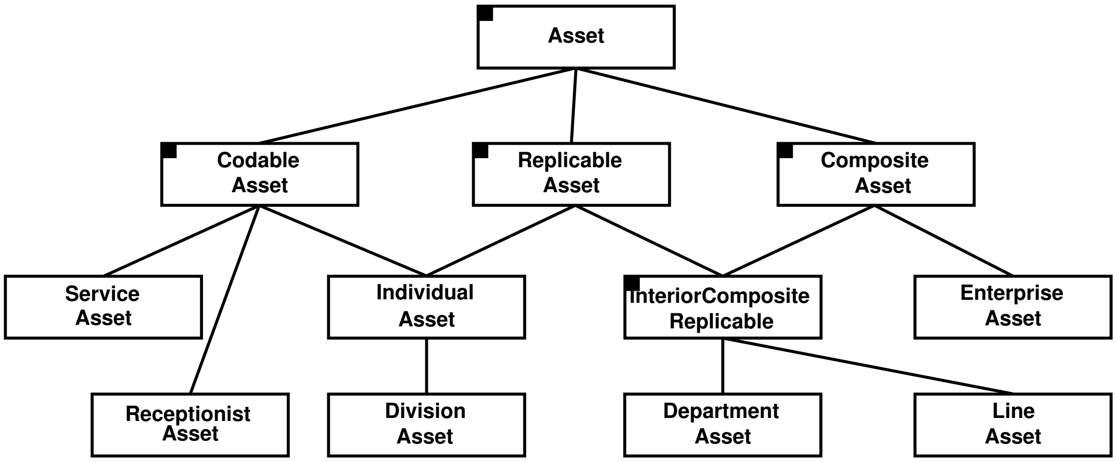


Figure 3.2: The class hierarchy for the Enterprise run–time system.

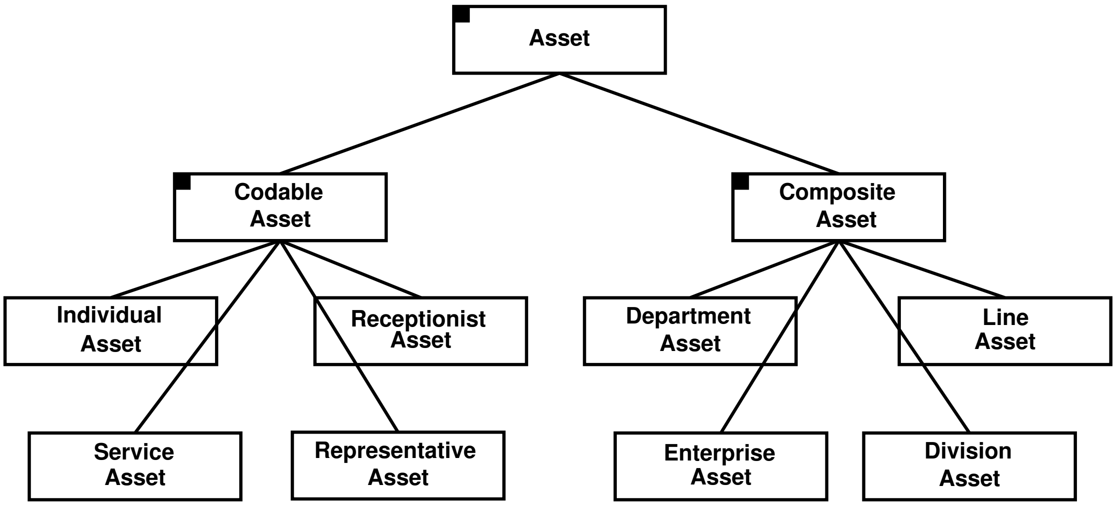


Figure 3.3: The class hierarchy for the user interface.

(Section 2.2). However, the abstract superclasses, which help describe the behaviour of their subclasses, are:

Asset This is the base class for all asset types. It contains information relevant to all asset types, such as the asset name, a pointer to the owner asset, and the process information for an asset.

CodableAsset This class represents any asset that has user code associated with it. Instance data in this class are the compile flags, including debug and optimize.

ReplicableAsset This class represents any asset that can be replicated. Instance data in this class are the minimum/maximum replication factors and the ordered flag.

CompositeAsset This class represents any asset that is composite. Instance data in this class are the receptionist for the class and an array of *components*, the assets that make up the composite asset.

InteriorCompositeReplicable This class allows the department and line assets to factor common behaviour. For instance, parsing a line or department reads in the replication factors, stored in ReplicableAsset, and initializes the components, stored in CompositeAsset. Without this class, this method could not be factored since instance data from both immediate superclasses is accessed.

The first difference in the two hierarchies is that the interface uses single inheritance while the new design uses multiple inheritance. This difference results from the languages chosen for implementation; the user interface was written in Smalltalk[2] [16] whereas the run–time system was written in C++. Unfortunately, the interface hierarchy is not able to benefit from the run–time system implementation the graph file because the code is not compatible.

Single inheritance, the only model supported in Smalltalk, was identified as a deficiency in the implementation of the user interface. Problems arose when trying to

[2]Smalltalk is a trademark of ParcPlace Systems.

factor responsibilities from the design, which used multiple inheritance, into a singly-inherited tree. The result was that several of the subclasses were forced to override methods from their superclasses to prevent incorrect operations from being applied. As an example, the Asset class supports a `replicate` method, which is inherited by all other asset types. However, the EnterpriseAsset cannot be replicated, so it overrides the method to make it a run-time error. This violates one of the ideals of object-oriented design; a subclass should respond to all messages of its superclasses.

Using the multiple inheritance mechanism within C++, the new design avoids this pitfall. An abstract class ReplicableAsset was created and appropriate assets inherit from it and other applicable superclasses. Since the responsibilities have been factored down, it is possible to avoid making the EnterpriseAsset replicatable by not having it inherit from the ReplicableAsset class, as shown in Figure 3.2. This property has the benefit of increased encapsulation and, more importantly, better reflects the behaviour of the classes. It also clarifies the design, making it possible to understand the behaviour of an asset type simply by examining its superclasses. This was not possible with the original hierarchy because a class may have overridden some behaviour, which is information not included in the hierarchy diagram.

Finally, the division has been changed to a singular asset, rather than the composite asset it was before. The representative asset, the component of the former division, is now unnecessary and has been removed. This is the result of a new model for the division asset, which is further discussed in Section 4.4.

3.4.2 The Asset Graph

One of the structures inside the run–time system is the asset graph, which describes the structure of the user program. The graph is integral to the proper operation of the run–time system; it is required to properly launch the application and correctly process messages. It is also used by the compiler to verify that asset calls are made within the constraints specified by the asset graph.

This section discusses the graph in detail. It explains the structure of the graph,

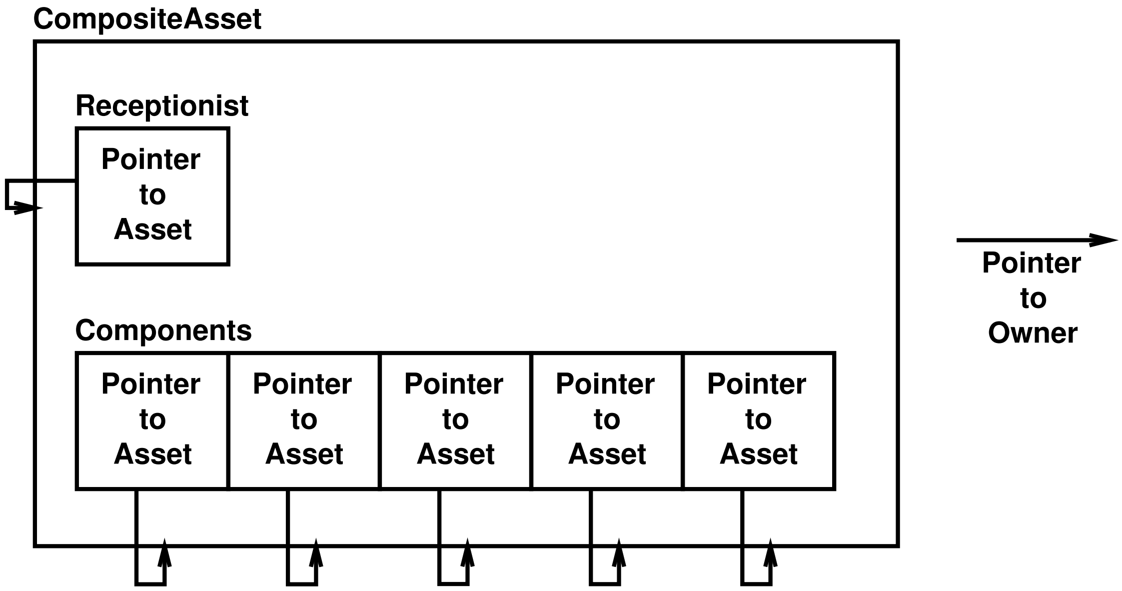


Figure 3.4: The structure of a composite asset.

how it is created, and how it is traversed to retrieve structural information about the user program.

Structure of the Graph

Internally, the graph has a simple structure. It consists of singular and composite assets, as described in Section 2.2. A singular asset simply contains all necessary information about itself. Composite assets are a collection of other assets, including the receptionist. The structure of a composite asset is shown in Figure 3.4.

To provide a complex parallel communication structure, where several types of assets can be combined, it is possible for a component in a composite asset to be a composite asset itself. This structure is shown in Figure 3.5. The fourth component of $CompositeAssetA$ is $CompositeAssetB$, which could correspond to a department inside of a line, for example. By nesting composite assets this way, any graph that can be entered by the user can be represented. Note that, as with graphs resulting from the meta–programming model, this representation only allows acyclic graphs.

The graph is rooted at an instance of the EnterpriseAsset class, which represents

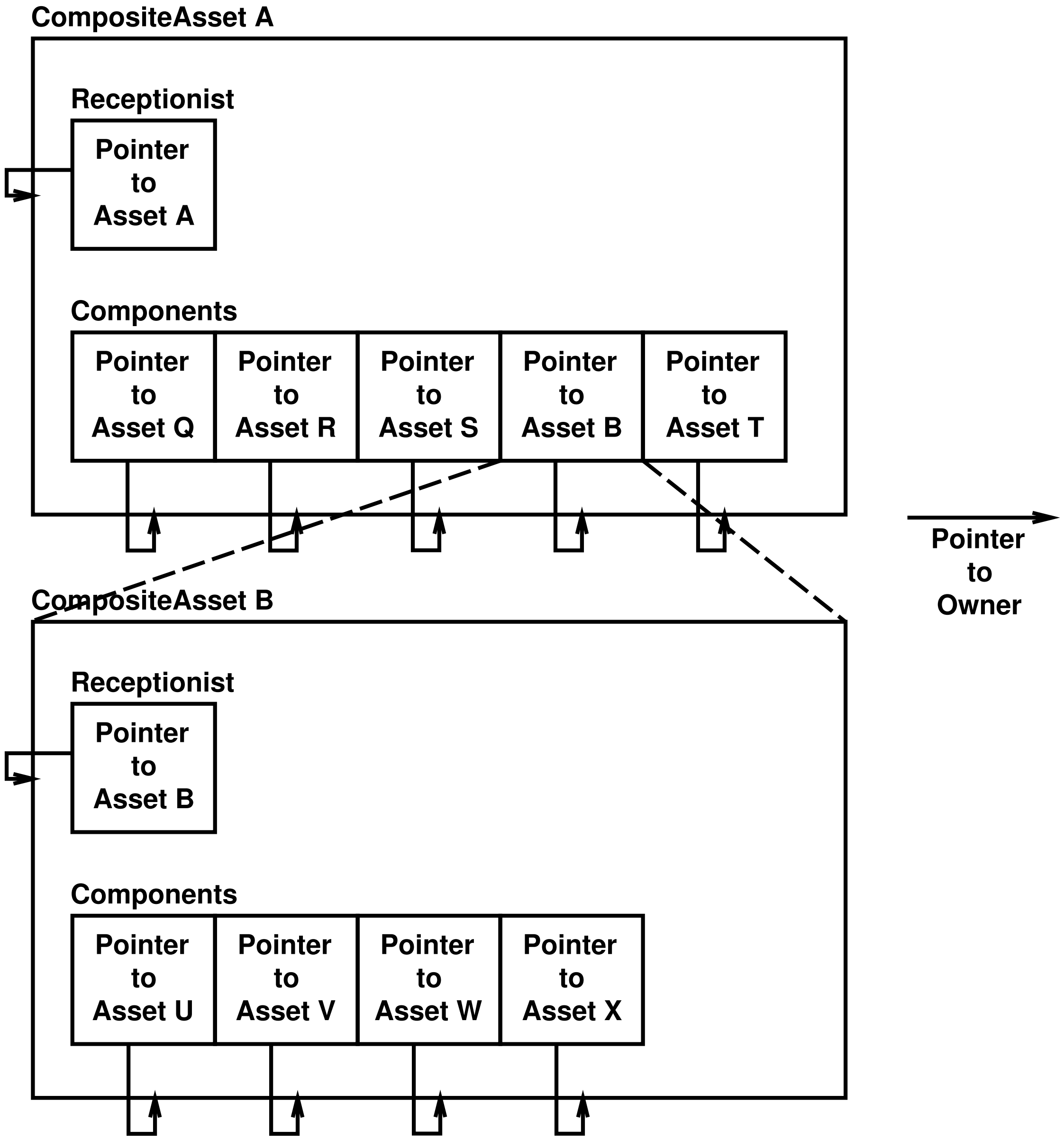


Figure 3.5: Nesting of composite assets.

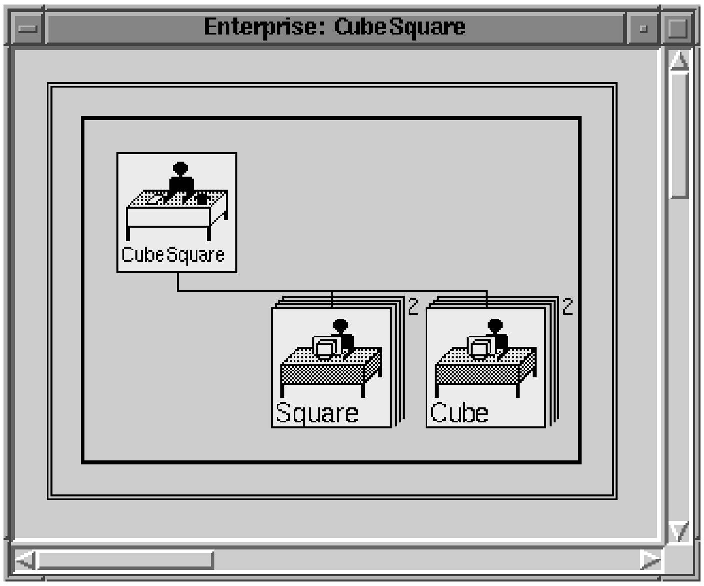


Figure 3.6: An example of an asset graph.

an entire user application program. The EnterpriseAsset class overloads the receptionist and components of the composite asset. The receptionist contains the user program, which can only consist of one asset. The components of the EnterpriseAsset class are the services, if any are defined. Thus, the receptionist and components are defined as pointers to generic assets, not to any specific type of asset. Otherwise, the receptionist could have been specified as an instance of ReceptionistAsset.

Parsing of the Graph

The graph file, the output of the meta–programming model, is a textual representation of the graph. The graph file for the asset graph in Figure 3.6 is shown in Figure 3.7. This graph consists of an unreplicated department asset that has two components. Each component is an individual asset that has been replicated twice. The correspondence between the two representations of the asset graph should be easy to discern.

In general, each asset is responsible for parsing itself from the graph file. A composite asset is responsible for parsing itself and starting to parse its components

```
CubeSquare department 2 1 1 ORDERED DEBUG PATIENT NOPTIMIZE
CFLAGS -DDEBUG
EXTERNAL
INCLUDE sawnlk.cs.ualberta.ca
EXCLUDE sass-lake.cs.ualberta.ca
Square individual 2 2 UNORDERED DEBUG PATIENT NOPTIMIZE
CFLAGS -I/usr/sundance1/grad/stevem/include
EXTERNAL
INCLUDE sundance.cs.ualberta.ca sunset.cs.ualberta.ca
EXCLUDE sawnlk.cs.ualberta.ca
Cube individual 2 2 UNORDERED DEBUG PATIENT NOPTIMIZE
CFLAGS -Wall -I/usr/sundance1/grad/stevem/include
EXTERNAL
INCLUDE
EXCLUDE sawnlk.cs.ualberta.ca sundance.cs.ualberta.ca
```

Figure 3.7: Example of a graph file.

(they will finish parsing themselves). However, the asset cannot be instantiated until the type is discerned, since an instance of the abstract class Asset cannot be created and later cast to the appropriate concrete subclass. Instead, another asset must create the new asset based on the first two tokens of the graph file entry (the name and type of the new asset). Once the new asset is created, it is capable of parsing the remainder of its graph file entry. For this to work properly, there must be an initial asset to initiate the process.

As noted earlier, the Enterprise asset is the root of the graph, making it an ideal candidate for this function. Since it is not an explicit part of the user graph, it does not need to parse any part of the graph file and is easily created. This process simply takes the first two tokens from the graph file and creates the first asset. Using the above example, the two tokens `CubeSquare` and `department` are taken and a department asset is instantiated.

Once the initial asset has been created, the Enterprise asset sends it a message telling it to parse itself. The department parses its entry, which indicates the department has two components and has a minimum and maximum replication of one (the "2 1 1" following the `department` keyword). The remainder of the first line is a sequence of flags indicating if the asset is ordered, compiled with the debug option, patient, and compiled with the optimize flag. The remaining four lines are the compilation flags, a reserved field for parallel I/O, and lists of machines that any process executing this asset should/should not run on. After the department is finished, it begins parsing its components. This process is identical to the parsing of the initial department by the enterprise asset, and can be nested arbitrarily to accommodate any graph that can be entered in the user interface.

The receptionist asset is handled slightly differently than above. It is not explicitly parsed, but rather is parsed in step with its owner. The information needed from the graph file entry for the owner and the receptionist is almost disjoint. The owner uses the ordered flags and the replication factors. The receptionist uses the debug, optimize, and patient flags. Both assets use the last four lines of the graph file entry for the composite asset. When parsing a composite asset, its receptionist asset is instantiated and the information is given to the appropriate asset as it is parsed.

Traversal of the Graph

Once the graph has been created, it needs to be traversed it to find relationships between the assets. These relationships are used by the compiler to ensure the user's code matches the asset graph and by the run–time system to verify messages during execution as a consistency check. The run–time system itself does not traverse the graph; it invokes methods within the graph classes to derive the relationship information. There are two kinds of traversals that are used: *depth–first* and *owner traversals*. Please note that in this section, a message is defined as a message sent to an instance of an object in C++, and not a message between processes.

Given the graph architecture, depth–first traversals are simple. Methods are defined such that a singular asset performs the operation on its instance data and a composite asset forwards the message to its receptionist and components. As an example, consider applying a method that prints the names of all assets to the asset graph shown in Figure 3.5, where only the one component of *CompositeAssetA* is composite. The operation will proceed as follows. *CompositeAssetA* will receive the message. In response, it sends the same message to its receptionist. The receptionist, being a singular asset, applies the operation to its instance data and prints its name. Next, *CompositeAssetA* iterates over its components, sending each one the same message. The first three components accept the message and print their names, exactly like the receptionist. When the fourth component, *CompositeAssetB*, receives the message, it invokes the composite asset definition of the method and sends the message to its receptionist and components, causing each asset to print its name. This process will continue again if any component of *CompositeAssetB* is composite. Finally, after *CompositeAssetB* has finished processing the message, it is sent to the last component of *CompositeAssetA* and its name is printed. The final output, which does not print the names of the composite assets, is

A, Q, R, S, B, U, V, W, X, T

Using this definition for the operation, it is clear that this traversal is depth–first.

Before owner traversals are described, there is some terminology to define. Every asset in the graph has an *owner asset*, which is the composite asset in which it is contained. The owner asset is indicated by the arrows in Figures 3.4 and 3.5. Note that the owner of *CompositeAssetB* is *CompositeAssetA*, as indicated by the arrow from that component in the first asset. In addition, each asset has an *entry point* and *leaves*. For a singular asset, the entry point and leaves are the asset itself. For composite assets, the entry point is the receptionist. The leaves of a department consist of the receptionist and the leaves of all components, and the leaves of a line are the leaves of the last component.

Owner traversals are used to determine communication relationships between the assets in a graph. There are two relations that are important for the run–time system and the compiler. Given an asset, it is necessary to know which other assets are allowed to send requests to it and to which other assets it can send requests. Each of these problems is examined separately.

The first problem requires a method of upward traversal in the asset graph since requests can only flow down in an Enterprise program. Upward traversal is provided by giving each asset a pointer to its owner. As an example, take Figure 3.5 and find all the callers of the receptionist of $CompositeAssetB$. The callers of an asset are defined to be the leaves of the previous asset. To find the previous asset, the owner pointer is followed from the receptionist to $CompositeAssetB$. Here, the system notes that the receptionist is the requesting asset. $CompositeAssetB$ needs to find who can invoke its entry point or, more abstractly, who can invoke it. It requests this information of its owner in the same manner as the receptionist. $CompositeAssetA$ finds that the requesting asset is no longer the receptionist, so the traversal stops. The behaviour at this point depends on the type of $CompositeAssetA$. For a line, given component i, the previous asset is component $i-1$ so its leaves are returned. For Figure 3.5, the result is the leaves of $AssetS$. The receptionist is returned if the first component is the requesting asset. For a department, the only asset that can invoke a component is the receptionist, which is returned. For Figure 3.5, the result is the receptionist $AssetA$.

Finding all other assets that a given asset can send messages to is similar to the problem just discussed. The difference is that the graph is traversed downward to the next asset in the graph. An asset may send messages to the entry point of the next component. Since the components of a composite asset do not have knowledge of one another, this responsibility must be placed in the owner. However, a problem arises if the requesting asset is a leaf of its owner. The owner does not contain the next asset. If a next asset exists, it must be in the next owner asset, so the system traverses up one more level. This is continued until the requesting component is not a leaf or the root of the graph is reached.

As an example, consider Figure 3.5 and find all assets that may expect messages from $AssetX$. The first step is to follow the owner pointer to $CompositeAssetB$, where the system finds that the asset making the request is a leaf of the composite asset. Thus, the owner pointer is followed to $CompositeAssetA$, where a non–leaf component is now making the request. Again, the behaviour here depends on the type of $CompositeAssetA$. For a line, given component i, the entry point of component $i+1$ is returned[3]. For the receptionist of a line, the entry point of the first component is returned. In Figure 3.5, the result is the entry point of $AssetT$. For the receptionist of a department, the entry point of each component is returned. In addition, if the department is part of a line, the entry point of the next asset in the line is also returned. Thus, the receptionist $AssetB$ in Figure 3.5 would be able to send messages to the components of the department ($AssetU$, V, W, and X) and $AssetT$ if $CompositeAssetA$ is a line. If the requesting asset is a component in a department, the upward traversal continues since all components are leaves.

3.4.3 Launching an Enterprise Program

The launching procedure for Enterprise programs creates or *spawns* processes for the assets in the program. Each process spawns others for subsequent assets in the graph in parallel with the other processes in the application. This is different than the centralized approach where a single process spawns all the processes for the application. Not all assets need to create new processes when launched; the behaviour of a launched asset is a function of its owner and a function of the process itself. Since this procedure is now done in a distributed fashion, it relies on the graph to know which processes need to be spawned. It is important to note that in this section, a message is defined as inter–process communication, and not the object–oriented messages used previously.

This section is divided into two subsections. The discussion includes an overview of the launching procedure, which includes some notes on the process structure of an Enterprise program, and the details of the procedure.

[3]This component will always exist since the traversal up the graph continues if the asset is a leaf.

Overview of Launching

This subsection elaborates on the problem of launching an Enterprise program, concentrating on the actual process structure and terminology used in later sections.

An example of a complete Enterprise process structure is shown in Figure 3.8, which shows the processes created for the `CubeSquare` example (Figure 3.6). The process structure is derived from the asset graph, with one process for each instance of an asset and other processes used internally by Enterprise. There are eight processes for `CubeSquare`: the Enterprise process, the receptionist for the department `CubeSquare`, manager processes for `Cube` and `Square`, and two individual workers for both the `Cube` and `Square` assets.

The solid lines indicate the connections between the processes, showing how messages flow in this example. A process A is connected to process B if the asset represented by A can invoke the asset represented by B. The exception to this definition is that all processes are connected to the Enterprise process, as indicated by the dotted lines. This definition requires that A must either know the communication ID of B or have an open connection to B. It is important to note that this definition does not require that B be connected to A. A process may have the communication IDs of other processes but not be connected to those processes.

The first item to note is the presence of processes that are not an explicit part of the asset graph. These processes are created and used internally to help manage the application program and are included without the knowledge of the user. The first such process is the Enterprise process. It handles all system messages in the program, such as the shutdown message. It is also responsible for event logging [20] and executing a program using controlled replay [13]. As a result, all other processes are connected to the Enterprise process, indicated by the dashed lines.

The second process to consider is the manager process. A manager is used with a replicated asset to control the workers using the *administrator* concept found in distributed computing [9]. The administrator controls the replicated workers by queuing up messages until workers become available. The administrator is typically a separate

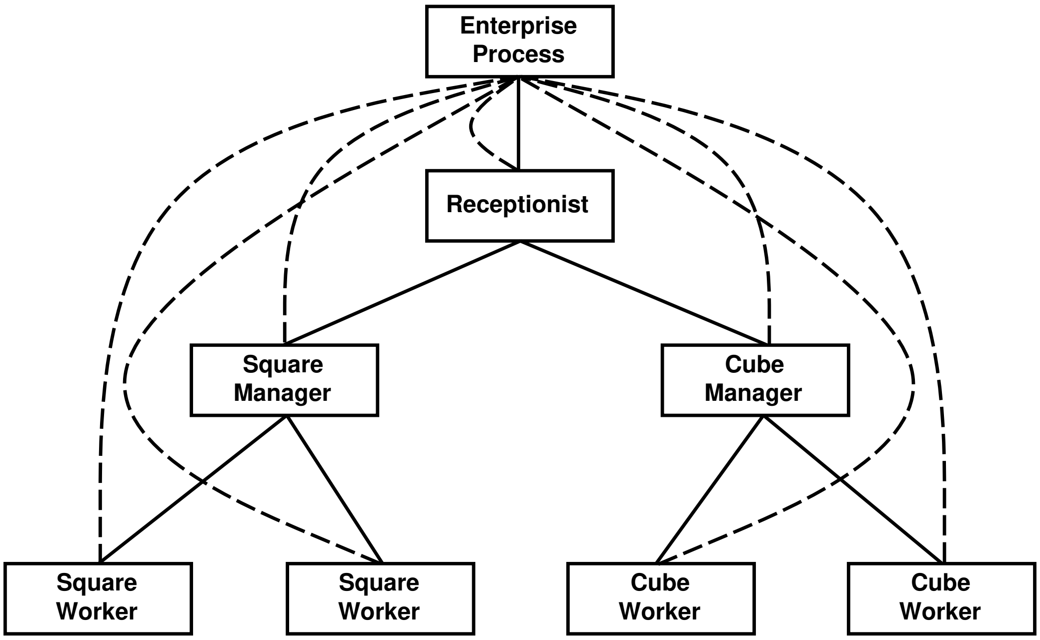


Figure 3.8: A process diagram for an Enterprise program.

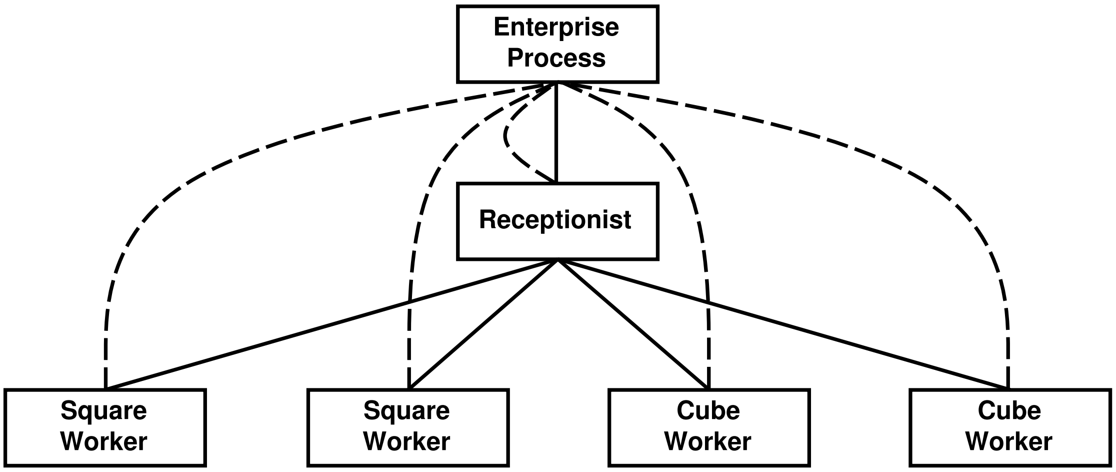


Figure 3.9: The real process diagram for the `CubeSquare` program.

process, providing a single entry point to the workers. In Enterprise, an optimization has been applied to manager processes. If only a single process can invoke a replicated asset, then the manager process is collapsed into that process. Note that more than one manager can be collapsed into a process. In the `CubeSquare` application, the managers for the `Cube` and `Square` assets are collapsed into the `Receptionist` process, resulting in the process diagram in Figure 3.9. This optimization saves a process and two network messages (the request to the manager and the reply back from it). Now, though, the process that contains the manager must multiplex its own responsibilities with those of the manager, which grows with each one collapsed. If more than one process may invoke the replicated asset, the manager must be a separate process.

Before the discussion continues, the terms *parent process*, *next asset*, *root task*, and *user task* need to be defined. The parent of a process is the process that was responsible for spawning it. The parent is important for replicated assets as it will always be their manager. The *parent asset* is the asset represented by the parent process. Unless otherwise specified, the *next asset* is the asset that follows the current one if it is in a line. If there is no line, the next asset does not exist.

The *root task* represents the root of the process structure, or the Enterprise process. This is the only process directly spawned by the Enterprise system; the remainder are spawned in a distributed fashion by other Enterprise processes. A *user task* is any other process in an Enterprise program. The difference between the root and user tasks is the information available to them during execution. The root task, by definition, has access to the Enterprise environment, including the graph file on disk. This distinction is important because we cannot always assume a homogeneous file space, so a user task cannot assume it has access to the graph file from disk. Instead, it receives the graph file and the name of the asset it represents as part of its initialization messages. The user task parses the received graph file and uses the name to initialize and launch the asset.

To reuse the parsing methods for both tasks, two stream classes were implemented specifically for the run–time system, one using files and the other using a memory stream. By using an abstract base class, the parsing methods were reused for both tasks.

The launching of an application is done in one pass down the asset graph. However, any given process in an Enterprise program requires the communication IDs of any processes it may invoke. This is accomplished by having each asset spawn processes for assets it may invoke before spawning processes for its internal assets (such as replicated workers or components of composite assets). The spawning asset gets the communication ID as a result of spawning the new process, and the internal assets get all known communication IDs from their parent. A process does not require the communication IDs of processes that may invoke it; messages contain sufficient information to verify the calling process is making a legal request.

The Launching Procedure in Detail

The procedure for launching an Enterprise program starts with the Enterprise process or root task, which opens a file stream and parses the graph file by instantiating the Enterprise asset. The Enterprise asset is launched, which spawns processes (user tasks) first for the services and then for the entry point of the receptionist (the user program). The spawn method returns the communication ID of the created process, which is stored in the asset graph as discussed in Section 3.4.4.

When a user task starts, it first waits for its initialization messages, which contain the graph file, all known communication IDs for assets, the name of the asset the task represents, and other information needed for execution. The graph file is parsed and the represented asset is launched. This procedure spawns the next asset or assets in the graph. The exact behaviour depends on the asset type of the owner asset, the number of leaves, replication factors, and the function of the process. These factors determine which processes need to be spawned, given the constraint that each asset must have the communication IDs of any assets it may invoke. An asset may also know communication IDs of other assets; this extraneous information will simply never be used by the asset.

The asset type of the owner gives us the general flow of control for launching processes. This discussion ignores problems raised by the leaves, replication factors, and the function of the process, which are treated later. In a department, the receptionist is responsible for spawning all of its components and the next component after the department if contained in a line. In a line, each component is responsible for launching the next component in the line (the receptionist spawns the first component, which spawns the second, etc.).

However, if a line is embedded within another line, the leaf of the inner line is responsible for launching the next component of the outer line, not the receptionist. This allows the launch procedure to correctly establish the parent processes in the event that the next component in the outer line is replicated and its manager is collapsed into the leaf.

Consider Figure 3.10, which contains a department within a line. The receptionist of the line, $R1$, is responsible for launching the department, $R2$ (specifically, the receptionist). Receptionist $R2$ is responsible for launching its components, $AssetA$ and $AssetB$, since it is contained in a department. In addition, the department is responsible for launching the next asset in the line that contains it, asset $AssetC$. This algorithm resulted from attempting to handle this scenario. Subsequent processes need a consistent, single point from which they can be spawned. Since a department can have multiple exit points but only one entry point (the receptionist), it was chosen to perform the launch.

In this example, we can also see how multiple leaves affect the launch procedure. All of the components of the department can call $AssetC$. In order for the communication ID of $AssetC$ to be known in one pass, its process must be spawned before $AssetA$ and $AssetB$. In this example, $R2$ spawns $AssetC$ and then its components. The same problem arises in Figure 3.11, where $AssetD$ in the line is replicated. Note that $AssetD$ need not be a singular asset. The solution is analogous to the one for Figure 3.10; the manager of $AssetD$ (the receptionist $R3$) must spawn $AssetE$ be-

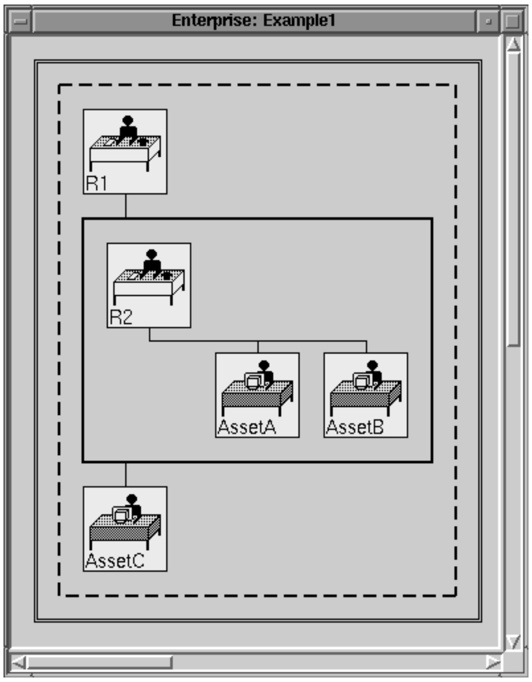


Figure 3.10: A department within a line.

fore spawning the replicas of $AssetD$. This is the first case where the function of a process comes into play; the process must realize it is the manager of a replicated asset and not a replica. The manager needs to spawn the next asset, but the replicas do not spawn any subsequent assets (except for internal ones) to prevent incorrect replication.

Note that the case where $AssetE$ is also replicated is the same; since there are multiple processes that can invoke it, it will need an explicit manager and hence can be thought of as a single process from the perspective of the spawning asset. This single process becomes a manager and spawns the worker processes.

An alternative implementation for the problem in Figure 3.11 would be to create a logical manager for $AssetD$. This process would spawn $AssetE$ followed by the remaining workers for $AssetD$. At the end of this procedure, the logical manager would then become a worker. This solution appears to be simpler because $R3$ does not need to be aware that it is the manager of $AssetD$. However, this implementation requires extra communication between the logical manager the the actual manager to

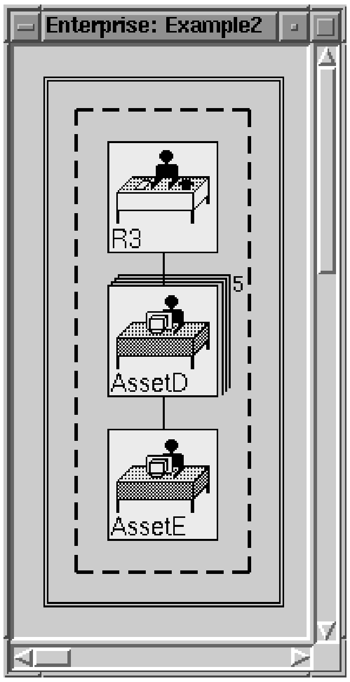


Figure 3.11: A line with a replicated component.

deliver communication IDs of the workers, which still requires the actual manager be aware that this information is forthcoming. Thus, there is no benefit to this solution.

It is clear from the above description that collapsed managers make the launching mechanism more complicated. A process has to look ahead in the graph before launching another asset to determine exactly what assets need to be launched. During the initial phases of this project, all managers were explicit, which simplified the task of launching a program since each asset could be thought of as a single process. If the asset was replicated, that process simply became a manager and spawned all necessary replicas. Although explicit managers result in a simpler process launch, the potential performance benefits from collapsed managers may make the additional complexity worthwhile.

3.4.4 Process Information in the Graph

Throughout the above discussion, the details of storing communication IDs in the graph was avoided. This section deals with this important aspect of the asset graph.

Inside each instance of an asset is an instance of a class that holds all associated process information for the asset. This process information includes a list of communication IDs for that asset, called the *communication list*. Internally, messages are sent to the desired asset via a method invocation. The asset passes the message to the process information in the same way. The process instance then decides what needs to be done with the message. This method of communication requires that each asset have the communication IDs of any assets it may invoke, which is assured in the launching procedure.

The process instance decides how the message is dealt with based on the number of communication IDs in the communication list. If there is only one ID, the message is sent directly to that process. If there are multiple IDs, then the process instance acts as a manager for the asset, queuing up the request until workers become available. Placing this functionality in the process instance allows the manager of an asset to be collapsed into another asset rather than always be a separate process. However, if the manager is collapsed, a method is needed to process messages while user code in the actual asset is executing. Otherwise, the workers can be starved, unable to get further work since the manager cannot process replies and give out new work. A timer was used to interrupt user code and check for messages, multiplexing the manager with the executing asset.

3.4.5 Executing an Enterprise Program

The final aspect of the design discussed is the execution of an Enterprise program, specifically the *behaviour classes* and their use at run–time. To understand the behaviour classes, the dispatcher classes need to be examined first. Combined, these two sets of classes comprise the run–time portion of the system.

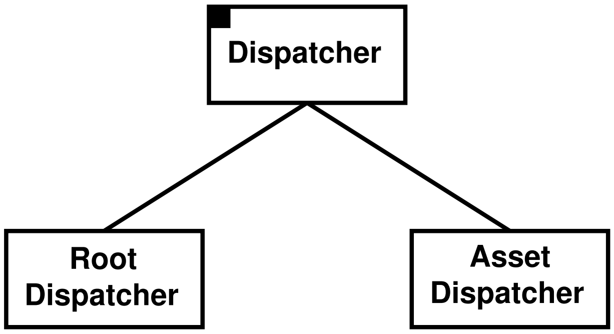


Figure 3.12: The dispatcher class hierarchy.

Finally, the procedure for shutting down an Enterprise program is given. This procedure is a synchronized way of ensuring that processes only terminate when there is no outstanding work.

Dispatcher Classes

The dispatcher classes (shown in Figure 3.12) are mainly responsible for receiving incoming messages and delivering them to the behaviour, which is an instance variable of the dispatcher. The behaviour is actually created by the asset, as the function of the process is required to know which behaviour is correct.

The difference between the two concrete classes is that the `AssetDispatcher` has message buffers that can be used to cache messages for later processing. The most common case where the buffers are necessary is to buffer an incoming request message if an individual asset is currently processing another message. An individual asset executes each request to completion, so the other message cannot be processed immediately. The `RootDispatcher` does not need to buffer messages, as the root process can process them as they arrive.

The dispatcher classes hold an instance of the behaviour so that it may be changed at run–time by simply instantiating a new one. The common interface to all the behaviours makes this change transparent; the only difference will be in the processing of subsequent messages.

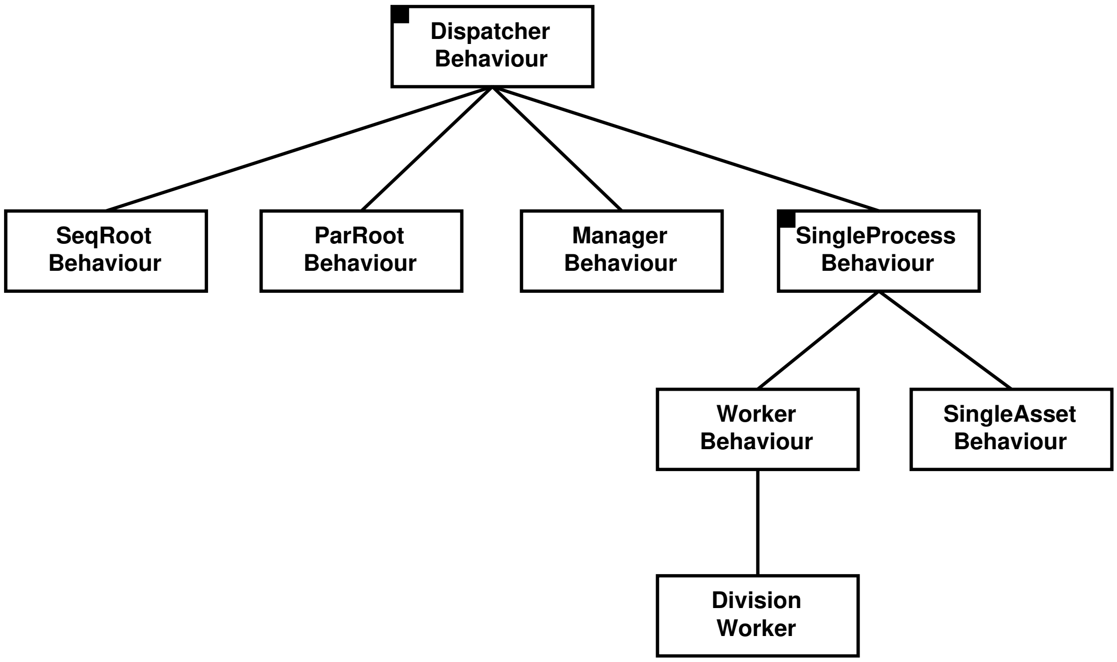


Figure 3.13: The behaviour class hierarchy.

Behaviour Classes

The behavior classes (shown in Figure 3.13) are responsible for processing incoming messages. By using a common abstract interface in the `DispatcherBehaviour` class, the processing can be refined in the subclasses until the desired run–time behaviour of a process is expressed.

Each concrete class in the hierarchy has a specific purpose. These are:

SeqRootBehaviour This behaviour is used to execute an Enterprise application sequentially rather than in parallel. This feature can be selected from the interface to debug the program before it is run in parallel. It simply invokes the user code with a flag to bypass the parallel code inserted by the Enterprise precompiler.

ParRootBehaviour This behaviour indicates the current process is the root process of an Enterprise program that is being run in parallel. It sends the the command line arguments to the first process and then continues to process messages.

ManagerBehaviour External managers use this behaviour to process messages and manage its workers. Since the process instance inside the asset does all of this management (see Section 3.4.4), this behaviour mainly consists of sending the messages to the managed asset. If the manager is collapsed into the calling asset, this behaviour is not used; the proper behaviour for the process is used in conjunction with the timer facility described in Section 3.4.4.

SingleAssetBehaviour Unreplicated assets use this behaviour to process their messages. This class only generates reply messages to requests if there is data to return.

WorkerBehaviour Workers in a replicated asset (except for divisions) use this behaviour, which always generates a reply message. A reply is always needed to inform the manager of the worker that it is available for more work. In addition, this class sends an initial message to its manager process indicating its availability.

DivisionWorker Workers in a division asset use this behaviour. It behaves like its superclass `WorkerBehaviour` except that an extra depth field is used to determine if subsequent recursive calls are parallel or sequential. This depth field is one used when specifying a division asset in Section 2.2.5.

The behaviour classes also help make the Enterprise system more flexible. Behaviours can be added or modified without affecting any other existing code, minimizing the side effect of these changes.

Shutting Down an Enterprise Program

Finally, after an Enterprise program has finished executing, it should be shut down in an orderly fashion to prevent *orphan processes* from being left over. Orphans are processes left running after a parallel program is shut down, taking up resources by running part of a program that no longer exists. The user will have to manually kill these processes if they have been abandoned by the run–time system.

The first step is determining when an Enterprise program is finished. This turns out to be simple; it is finished when the Enterprise process receives a reply from the first asset. This event is equivalent to the mainline procedure returning and terminating a sequential program. However, if any asset in the program generated work but not futures (by using an asynchronous call), there may still be work being processed by the rest of the program that needs to be finished before the program is shut down. The program `Mandelbrot`, included in the Enterprise distribution, is an example of such an application. The results are not returned data, but rather an output display of the Mandelbrot set.

A program is shut down by a synchronized traversal of the process structure. Each process keeps a list of the assets that may invoke it, with each asset checked off when a `die` message is received from a process representing that asset. When a process has received die messages from each asset in its list, it continues to execute as long as it has outstanding work to finish. When this work is exhausted, the process sends a die message to all processes it may invoke, waits for acknowledgements of these messages, and finally acknowledges all of its received die messages and shuts down.

Replicated assets are handled slightly differently than above. The manager sends a die message to any assets its workers may invoke that also includes the number of workers it has at that time. This message is acknowledged by the receiving assets. The manager can then shut down its workers, each of which must send the die message to the assets they may call. An asset that may be invoked by a replicated asset can only die when all workers have sent it a die message.

For a concrete example, consider shutting down the `CubeSquare` application, with the process structure in Figure 3.8. The Enterprise process receives a reply from the receptionist for the department, which causes the Enterprise process to send a die message to the receptionist and await an acknowledgement.

The receptionist receives the die message and notes that this is the only one it is expecting, which flags the process as dying. The process checks message and future queues, continuing execution if any outstanding work remains undone. These checks also include any queued work in collapsed managers. Once the receptionist has determined it has no further work, it proceeds to shut down by sending die messages to the managers for `Cube` and `Square` and awaiting acknowledgement.

In this example, however, the managers are actually collapsed into the receptionist so the messages are not actually sent. Instead, the workers are shut down by the receptionist, who appears as the manager of `Cube` and `Square`.

The workers behave exactly as described above. The acknowledgements traverse back up the process structure, causing processes to exit along the way. At the last step, the Enterprise process receives its acknowledgement and dies, terminating the application.

The most important aspect of this procedure is that it works even when there are asynchronous asset calls in the program. If all assets return results, this procedure is not necessary, as the Enterprise process only receives a reply after all work is finished. Otherwise, the state of each process must be determined and shut down must be delayed until all outstanding work is finished.

3.5 Problems in the Design Process

During the design of the new run–time system, several conflicts between similar terminology were encountered by the design group, which consisted of Paul Iglinski, Randal Kornelson, Kent McPhee, Diego Novillo, Ian Parsons, Jonathan Schaeffer and Duane Szafron. These conflicts hampered the design effort until they were finally resolved. In addition, some of the steps of object–oriented design were difficult because of the nature of the system and these conflicts. These problems are mentioned in this section.

First, the term *asset* has been overloaded to mean both an instance of the C++ class implementing the behaviour of an asset and the process that is executing the code for that asset. This overlap in terminology has even affected this thesis. However, the conflicting definitions caused even greater problems when the following conflicts arose.

The next problem resulted from the above conflict and overlapping terminology in distributed systems and object–oriented systems. The term *message* means different things to each programming paradigm, as shown earlier in this thesis. An object–oriented message refers to invoking a method on a particular instance of an object, a strictly internal message used to execute some code. A distributed message refers to a message sent between processes on different machines. Unfortunately, because of the high degree of correspondence between assets and processes, this overlap causes even more confusion as the responsibilities of the asset code and the asset process becomes unclear.

The next conflict resulted in confusion between an actual process and the model of the process stored in the asset graph (from Section 3.4.4). The process model gives assets a conceptual understanding of the processes in an Enterprise program, but only so that messages may be passed and processes may be spawned. This limited model and the aspects of a full process were perpetually confused by the design team, who found it difficult to discern the difference at times. The conflicting meanings for a message exacerbated this difficulty.

Also, the design process was iterative, with several decisions constantly changing as new ideas arose. The two best examples of this problem are the collapsing of managers and the new division specification. As these concepts evolved, the differences between the new and old specifications were confused until a clear direction for the design was found.

Because of the above problems, the design team found it difficult to execute scenarios to verify the correctness of the design. This situation hampered the process until the confusion was finally settled when the design team clarified the terminology.

3.6 Other Interesting Concepts

One implementation problem that arose early on was how to send an instance of an object from one process to another. This problem appears with the communication ID lists in the process model and messages, which are implemented as classes within the run–time library. This problem was solved by providing methods for these classes to flatten their instance data into streams and vice versa. The technique was straight–forward; instance data was flattened into a binary buffer and was sent to another process. At the other end, the stream was given to an existing instance of the class, which applied the reverse transform from binary data to fill in the complete object instance.

This approach lets these classes be used normally, giving all the benefits of programming with a C++ class, while simplifying the task of sending this data to other processes. Particularly important is the ability to add new data to the message easily, only requiring the programmer to add an interface to the new data and modify the marshaling/unmarshaling routines. As well, any arbitrary data can be added to the message class as long as it can be placed into and read from a binary stream, including instances of new classes.

Currently, this technique is simplified by the absence of subclasses on those classes that are sent this way. It could be adapted to handle subclasses if type information is included in the message, which would require that the message be received, the proper subclass instantiated, and the binary stream unmarshaled.

3.7 Current State of the Implementation

The new run–time system has been coded using the above design. The size of the system is about 14,000 lines of documented C++ code, and has been tested on a network of SUN SPARC workstations.

Diego Novillo also tested some of this code, by using a library that parsed and manipulated the graph inside the new Enterprise compiler. This code, which has since been added to, was also tested on IBM RS6000s.

3.8 Summary

This chapter motivated the re–implementation of the Enterprise run–time system. In addition, the key concepts in the new design were explained in high–level detail. Finally, some of the problems and other interesting concepts used in the new design were covered.

Chapter 4

Results From the Re–Implemented Run–time System

One of the questions that could not be answered during design was just how many of the performance and design goals could be met during implementation. The design goals included using an object–oriented design and implementation, reducing of stubs code inserted into the program, making the code independent of the communications software, and adding dynamic behaviour to the run–time system, all while not dramatically impacting the performance of Enterprise and keeping the code clean and expandable. This chapter gives results obtained from the new implementation, addressing all of the above concerns. Other issues and features that arose during the course of the project are also discussed here. Although these features are not among the design goals set at the beginning of this thesis, they are noteworthy and further demonstrate the capabilities of the new system. These results will show that the design goals have been met, justifying both the new design and implementation.

```
int Cube(int i)
{
    return(i * i * i) ;
}
```

Figure 4.1: The code for asset `Cube`.

4.1 Stubs Code

Stubs code is inserted by the Enterprise precompiler to marshal and unmarshal messages. In the case of unmarshaling messages, the stubs code also invokes the user code and marshals the reply. This step is necessary because the run–time system has no good way of knowing what the arguments and return type of a function are, and hence cannot automatically perform this function. Since the run–time system does not know the arguments to an function, this data is treated as a binary stream that is passed around without any interpretation of the contents.

In the previous implementation of Enterprise, the stubs code was much more complex, including functions such as message authentication, message header processing, event logging, future queuing, and reply generation. The stubs code also handled most of the run–time behaviour based on a set of flags in the message header, which required considerable code. The new implementation only uses stubs code to marshal and unmarshal the arguments, with the rest of the code in the run–time library. Thus, the interface between the application and run–time code layers (Figure 3.1) is much smaller and cleaner, as is the stubs code itself.

To give actual numbers, consider the code for the asset `Cube`, shown in Figure 4.1, which is one of the assets taken from the example in Figure 3.6. Previously, the size of the post–processed code for this simple example was 1150 lines. This number is reduced to 108 lines.

Most of the reduction can be attributed to the removal of some system header files that were included in the old post–processed code. If the header file information is not counted, the size of the old code is 252 lines versus 61 for the new version.

Besides encapsulation, the biggest benefit of reduced stubs code was faster compilation, as the compiler had much less code to generate and subsequently compile. Not including the linking stage, the compilation times for CubeSquare dropped by 60%. However, linking time increased to remove some of this advantage, with an overall improvement of about 28%. The program was compiled without optimization and debug information.

4.2 Communication System Independence

Part of the rationale for the re–implementation was the dependence of the old version of Enterprise on the NMP communications package. Even when PVM eventually replaced NMP as the communications package, only features supported in NMP could be used, preventing the use of dynamic process creation.

To prevent the communication system from becoming integral to the run–time system, it has been encapsulated into its own class with an abstraction type provided for the process IDs. This communication manager has a simple, fixed interface and may be implemented using any library package provided the following functionality is provided or can be implemented:

Spawning processes The implementation needs to be able to spawn the processes for the application. To take advantage of the distributed launching procedure and dynamic process addition, any process should be able to spawn another. With more effort, a library that only provides a static process structure can also be used.

Send/receive messages This is just basic functionality for any communication library. However, the Enterprise run–time system needs to be able to receive a message from either a specific process or from any process.

Message Ordering The run–time system is written in such a way that messages between two processes must arrive in order in which they were sent. No other ordering constraint for messages is required.

Polling for messages When managers are collapsed onto calling assets, the run–time system uses a timer to interrupt the user's code and check for the presence of messages. This check is a non–blocking poll of the incoming message queue, which must be provided in the library.

Compatibility with timer The poll for messages in a collapsed manager is done using a timer and the `signal` facility. The communication manager must accommodate the use of a timer, either using the `alarm` currently used or through some other method.

Parent process Every process in an Enterprise program has the concept of its parent process. This information must be available to every process.

The run–time system will function properly if the communications manager satisfies these constraints. The requirements are straight–forward and are provided in most communications software packages. Other operations, such as process group notification, would be useful but are not part of the specification as they cannot be easily provided in all communications packages.

The best way to verify that the communication system has been properly encapsulated is to re–implement it with another communication package. Diego Novillo is currently working on another communication system that uses a distributed shared memory library. The success of this work will verify the run–time system is independent of its communication system, validating this design objective.

4.3 Dynamic Behaviour

Another objective of the new system was to provide for more dynamic behaviour in an Enterprise application, a feature the old version of the run–time system could not support because of static NMP process structures. This research is forthcoming, and the new system is expected to support its implementation. In this section, the ability of the new implementation to support dynamic behaviour is demonstrated.

The principle dynamic construct in the Enterprise system is replicated assets. These assets are specified using a minimum and maximum replication factor with the intention that the actual number of replicas vary with system resources and application requirements at run–time. The new implementation, like the old one, statically spawns the maximum number of replicas. However, the new system is able to dynamically incorporate new workers into the computation. This feature can be used in future versions of Enterprise to implement dynamic replicated assets as described above.

One side effect of the static nature of the old run–time system appeared when specifying the replication factors for an asset in the graph. If the replication factor was changed from one to many or from many to one, the program had to be recompiled to regenerate the NMP file and correctly spawn or omit manager processes. The new run–time system defines the communication manager to include dynamic process addition and the launch procedure uses the graph at run–time to determine what processes are necessary, independent of any other static structure. Combined, these two traits allow the replication factors of an asset to be changed arbitrarily without requiring the application be recompiled.

There are two issues about adding dynamic workers that need to be addressed. The first issue is spawning a new process if it does not already exist. This is solved by definition; the communication manager must supply the ability to do this.

The second issue is integrating the new process into the existing program, which is subdivided into making sure the new process can receive messages from its manager and can send messages to other assets it needs to call. Both of these are addressed by spawning the new process from its manager. The new process will have its manager as its parent, as required in Section 3.4.3. Next, the new process will need to inform its manager it is available for work, and the manager must recognize the new worker. The process instance in each asset, which contains all the process information, is written to add workers as the initial messages come in rather than making this a barrier to be reached before execution begins. When the process receives a message from an instance of an asset it manages, it finds that worker in its internal worker queue and either marks it as free or sends it more work. However, if the worker is not in the queue, then it is added. Since this can be done at any time, adding and integrating a new worker is possible. If the worker is removed from the worker queue, it is effectively removed from the system and can be destroyed.

Finally, the manager of a replicated asset will know the communication IDs of any other assets that may be invoked by the workers. This trait is assured again by the launching procedure. During launch, the manager gives this information to the new worker, which will now be able to send messages to any asset it may invoke.

The above discussion assumed that the manager process already exists. While it is possible to always ensure a manager exists for any asset that may be replicated, it is inefficient for an asset that starts with only one worker, adding two extra messages when calling the asset. The dispatcher and behaviour classes from Section 3.4.5 can be used to handle this case, by *promoting* a process to a manager and creating new workers. In the reverse case, a manager process can be *demoted* when it has no more workers.

The scenario starts with an asset that decides more workers are necessary. This decision could be based on the size of the queue holding pending request messages. Between requests, the process for that asset decides to promote itself, making itself a manager and spawning additional worker processes. This operation is done in the dispatcher, which simply deallocates the previous behaviour and instantiates a new manager behaviour for the process. The queue holding outstanding requests is flushed by sending those messages to the managed asset inside the graph, where the process information will buffer them until workers register themselves with the new manager. Promoting the existing process is preferred to creating a new manager because other assets that invoke this one already have its communication ID, so the change is invisible from their perspective. If a new manager were created, it would have to be grafted into the process structure, which requires a protocol for informing the callers of the new process.

The reverse operation can be performed if the number of replicas drops from many to one. When the last worker is removed, the dispatcher can demote itself by changing its behaviour to an unreplicated asset. Any messages queued up in the process structure can be delivered back to the the new behaviour and executed locally.

If the ideas of promotion and demotion are carried a little further, a process with proper graph information can even change the asset it represents and execute different user code. This feature would allow a pool of workers to be created and simply assigned an asset when requests for additional replicas are made, rather than having the manager spawn a new process[1]. This is made possible because all of the user code in an Enterprise application is linked into one executable, so all the code is available to all processes. It is also made possible because the graph and the run–time behaviours are almost orthogonal; the only dependence is in the data contained inside the process instance. This dependence can be resolved by overwriting the process instances using data from either the manager or from another process with complete knowledge of the process structure of the application. Once the process has the proper process information, it should be easy to graft it into the existing program.

The `CubeSquare` program was used to verify that additional workers could be added to a replicated asset at run–time. This program is ideal for this test as the program works the same, independent of the number of workers available. In addition, because of the regular structure of the program, it was easy to modify the run–time system to simply create a new worker every five requests up to a total of six workers. Since promotion has not been implemented, the program started with two workers already instantiated. It should be made clear that the addition of new workers in this test is completely arbitrary and only meant to show that the run–time system will support this operation. As expected, the new workers were integrated into the computation, which was shown by output from the new workers.

[1]The parent process would have to be modified for the worker to recognize its manager.

Clearly, the new run–time system is capable of dynamic behaviour. New processes can be incorporated into an executing application, which allows Enterprise programs to make better use of available resources. This section also validates the dispatcher and behaviour classes, which provides further avenues for dynamic process behaviour via the promotion and demotion mechanisms.

4.4 New Division Model

The definition for the division asset is a change from the previous meta–programming model, where divisions were represented as composite assets with a receptionist and representative assets. The user specified a division by creating a tree, nesting them to determine the depth and replicating them to specify the breadth, as shown with the binary tree of depth three in Figure 4.2. The new division is represented as a replicated singular asset, as shown in Figure 4.3. In this figure, the division is actually the first component in a line, which is necessary because the first component of an Enterprise program cannot be replicated.

This new representation has several advantages over the old one. It makes for a cleaner asset graph since the user does not have to specify an entire tree structure in the interface. The implementation is much simpler, as the structure is identical to a replicated worker, which has already been implemented.

The previous division specification provided a static process tree for this asset, leaving idle processes if a branch is not used. The new implementation will allow these processes to be assigned portions of other outstanding requests. This will increase the usage of the available processes and resources.

Finally, the new implementation of the division asset allows for greater flexibility in division nodes. Because a division was constructed of singular assets, the computation in a division node could not easily be subdivided into further parallel calls. A method to do this is to create a line of two identical divisions with the first invoking the other. This new division makes it possible to have a composite asset as a division node, a

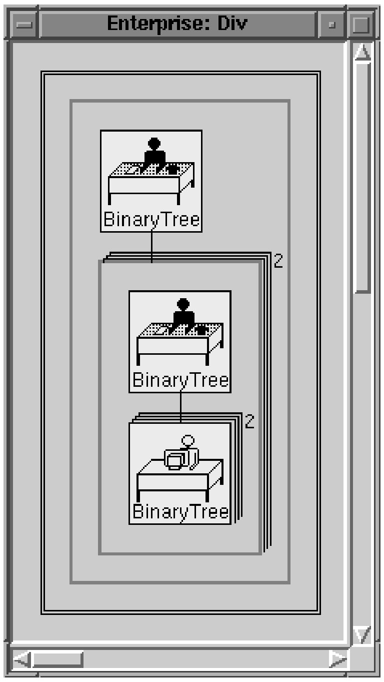


Figure 4.2: A binary tree in the old division representation.

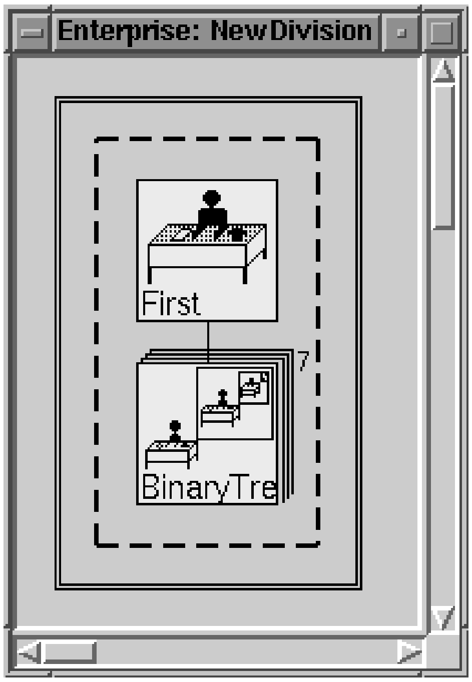


Figure 4.3: A division with 7 workers in the new representation.

more elegant solution to the problem.

The new implementation of the division asset also validates the use of an object–oriented language for the implementation of the run–time system. At early stages of the project, the old division model was implemented so that graph files generated by the old system could be properly parsed. When the new division model emerged, however, it was easily inserted into the system as a subclass of the individual asset. Additional changes to the meta–programming model can be made this way.

This work was done as the first step of additional research in parallel recursive program structures in Enterprise, and as such was not fully tested and evaluated. The remaining work is being done by Nicholas Kazouris as part of his Master's thesis.

Preliminary work with the new divisions shows that the new model is functional but not robust. A parallel quicksort was created and successfully executed using the graph from Figure 4.3.

Unfortunately, the implementation leaves the possibility of deadlock if the number of worker processes is not enough for the full tree. As an example, consider a parallel quicksort implemented as a division of depth three but with only 3 processes, enough for a full binary tree of depth two. The first division worker partitions the elements and makes two recursive calls, which are done in parallel. The remaining two workers receive these requests, partition the data, and make two more parallel recursive calls each. There are no remaining workers to accept these requests, so the manager of the division buffers the requests up until a worker becomes free. However, none of the workers will ever terminate since they have not received their results. This scenario is only one way a division asset can deadlock; there are other ways to achieve deadlock that involve a division with multiple requests pending. Nicholas Kazouris is investigating better scheduling algorithms for parallel recursive programs in Enterprise.

4.5 Flow Control with Collapsed Managers

A problem that has persisted in Enterprise is the lack of flow control between processes. Programs that generate a large amount of message traffic can physically overpower the system resources. However, the new system should be able to implement some form of flow control to issue the messages at a rate that can be handled. This issue was considered during the implementation phase of this work, but is not an issue or goal for this thesis. This section describes one form of flow control that was implemented; there are other scenarios that are left for later discussion.

The tradeoff for flow control is memory versus network contention. Messages that are held back for later delivery must be buffered in memory, increasing the memory requirements for Enterprise processes. If these requirements become large, the program must either use machines with sufficient space or must implement flow control itself by splitting the computation into several parts and synchronizing between them to clear the buffered messages.

Collapsed managers are an optimization that eliminates the messages between the caller of a replicated asset and a manager. The obvious benefit of this work is reduced communication time. A less obvious benefit is that the collapsed manager also provides flow control between the manager and its caller. This form of flow control reduces network contention by eliminating message traffic between processes, most visible in programs that initially generate large amounts of work. For these programs, reducing contention may be more valuable than reducing the amount of messages.

To demonstrate how collapsed managers can improve flow control, a block matrix multiply program written by Ian Parsons was used [24]. The asset graph for this program is given in Figure 4.4. The receptionist `Fblock` splits the input matrices into submatrices and sends pairs of the submatrices to `multiplyMATRIX` for multiplication. The results are sent to `collector` where they are merged. The number of messages in the system grows with the number of submatrices, which can strain the communications system by flooding it with messages. This problem occurs at

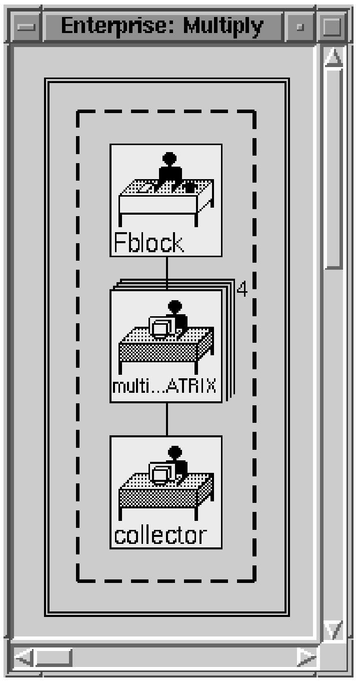


Figure 4.4: The graph for the block matrix multiply program.

the beginning of the program, where the first asset sends work to the manager of the replicated worker. This stage executes much faster than the messages can be received, clogging the network and limiting the size of problems that can be run efficiently.

Again, in this example the manager for the replicated asset `multiplyMATRIX` is collapsed into the receptionist. Request messages to the replicated asset will be queued in this process and given to available workers rather than flooding the network. This reduction in the number of messages should reduce contention and allowing the program to work much faster.

For an initial test, multiplying two 100x100 matrices by splitting each matrix into 16 submatrices was executed on five processors. The statistics were gathered by the program itself, and so do not include times for program launch and shutdown. Comparing the old versus the new system resulted in a 63% decrease in execution time. The most notable difference in the times is in the receptionist and collector asset. The time for the receptionist to execute dropped by 35% and the collector dropped by 64%. The reason for the drop in time for the entire application is reduced network traffic, which also reduces the network contention that has plagued this example program. Within the receptionist, the improvement results from messages being stored in memory rather than being sent across the network. The receptionist finishes executing its code and spends the remainder of its time acting as the manager for multiplication, which is not counted in the execution time of the receptionist.

A larger test was performed using 700x700 matrices, each split into 49 submatrices, on five processors. The wall–clock time was better by 41%. The execution time for the receptionist dropped by about 35% again, and the collector decreased by 41%.

The reason for the decrease in total improvement can be explained using Amdahl's Law [1]. In the 100x100 matrix example, the proportion of communication time to processing time is large, so reducing the communication time results in a dramatic improvement. In the 700x700 matrix case, this proportion is smaller, so the reduction in communication time has a smaller overall effect.

Collapsed managers provide two benefits to an Enterprise application. First, they eliminate the messages between the callers or a replicated asset and a manager. Second, the reduction in messages provides another form of flow control that can further reduce network contention and provide better performance. It should be noted that these results are based on the execution of the block matrix multiply program; no measurements to determine the actual reduction of messages were made.

4.6 Heterogeneous File Space

One of the disadvantages of the previous run–time system was its dependence on NMP as its communications system. This dependence forced a static process structure because of the NMP connection file. However, the use of this file imposed another limitation on the use of Enterprise.

Each process started by NMP must be able to read the connection file so it can determine which process it is running and which others it may communicate with, forcing the file to be visible on each processor the application is using. Thus, a homogeneous file space was needed for an Enterprise application to work correctly.

This restriction has been removed in the new implementation. Remote processes are no longer dependent on any files on their processors. Currently, the only files that need to be present are the executable and a directory structure for output files. This last restriction can be relaxed by making the placement of these files more flexible, placing them in either the proper directory or simply leaving them in the home directory of the user.

This feature has been tested by executing the `CubeSquare` program on a combination of the graduate research machines and the undergraduate machines, which do not share any file space. The program ran successfully using the available resources on each set of machines. The user interface can be modified later to integrate the results on a specific processor, and also to ensure the binary and directory structure exist on the remote machines.

This feature should also permit Enterprise applications to be run across a wide–area network. However, it would be better to run a sub–branch of the program on the far–remote machines rather than individual processes to cut the message costs during launch and execution. One possibility is to introduce a *remote executive*, a process with similar responsibilities to the Enterprise process but only managing a far–remote branch of the program. This process would allow the far–remote branch to use resources local to it, hierarchically combining results in larger pieces at the end of an execution.

4.7 Timing Results

An objective of the new implementation was to avoid any serious impact on the performance of an Enterprise program. While the designed algorithms were chosen with performance in mind, little was done as far as language optimizations, such as inline procedure calls. However, the two systems can still be compared to one another.

To determine how the new implementation has affected performance, the application `CubeSquare` was used. This program was chosen because it does little computation, so most of the execution time results from the run–time system. This test should provide a good indication of how the run–time systems compare.

The application was run on 6 processors, the ideal case for both systems given a static process structure and a replication factor of two for the `Cube` and `Square` assets. The new implementation contains a total of 6 processes, one for each processor. The previous run–time system contains 8 processes (two external managers for `Cube` and `Square`), but places the manager processes on the same processor as the callers to avoid a network message. The execution script was modified to time the execution of the program after any external communication manager initialization. Thus, the times includes program launch, execution, and shutdown.

As well, the program was executed from the command line, eliminating the overhead incurred by piping the output of the program to the user interface for display.

Over 5 runs, the wall clock time for the new implementation was 10.6% faster.

There are several possible explanations for this improved performance. First, although not of great concern in this example, is that a program can begin execution as it launches. Once a process finishes its launch, which spawns the next assets in the program, it can start processing requests rather than waiting for the launching procedure to finish.

Next, the use of the behaviours reduces the amount of computation that needs to be done to determine the proper handling of a message. The run–time behaviour is completely encapsulated in these classes, eliminating the run–time checks used in the older implementation.

Finally, the managers for `Cube` and `Square` are collapsed into the receptionist in this program. This saves the messages to and from the manager. The old implementation tries to limit this cost by placing the manager on the same processor as the receptionist, but this still requires a local message.

From the above discussion, the new implementation has lower overhead than the previous system for the example program.

4.8 Summary

This chapter presented results obtained from the new implementation of the Enterprise run–time system, which are compared against the goals set for the design and implementation. Some of these results are direct measurements of some of the metrics established as design, and others are algorithms that may be implemented to show both the flexibility in the new code. All of the goals were met or exceeded, although some have yet to be validated.

Chapter 5

Conclusion

5.1 Future Work

This thesis concludes with a look at some of the research that remains to be done. The new implementation of the Enterprise run–time system should provide a framework for this work.

5.1.1 Missing Functionality

The first issue to be resolved is to complete the new system so its functionality matches the old version. There are features that are either not present or not functional. Some of these features are:

Events Event logging, used for post–mortem debugging and animation of Enterprise programs, has not been implemented. This work is being done by Ian Parsons.

Replay The mechanism for controlled execution of a program from an event file does not exist yet. In addition to lacking an event file, this feature requires messages to be held by the run–time system so they can be sent/received in the order specified by the event file.

5.1.2 Information Gathering

Future research in the Enterprise system will include changing the run–time behaviour of the program as it executes, based on gathered statistics. Examples of this work will be automatically changing asset calls from parallel to sequential if the work is insufficient for large–grained parallelism and dynamically adding new workers to a replicated asset. These statistics will need to be gathered and sent to the processes where these decisions will be made.

Gathering the statistics in an Enterprise program is not a difficult problem. Measurements can be made in the run–time library before the stubs code is invoked, which will also include the time for unmarshaling the data. Alternately, the stub code can take the measurements and pass this data to the run–time library.

The statistics can be returned to the caller by placing them inside the message, which is implemented as a C++ class. The class can provide aggregate totals or break down the statistics for each asset call generated from processing this request, if desired.

The change from parallel to sequential is accomplished by inserting a flag around the code generated by stubs, which either executes the parallel or sequential version of the code. This flag has been used to execute the entire program sequentially, but can also be used by a process to execute its code sequentially. Dynamic workers were addressed in Section 4.3.

5.1.3 Flow Control Between Processes

Section 4.5 discussed how collapsed managers provided a form of flow control for Enterprise programs. However, flow control needs to be implemented between any communicating processes, not just between collapsed managers and their workers. This section provides some possible methods of dealing with flow control in different scenarios. Each of these methods are discussed separately as they depend on the function of the communicating processes.

The first place to provide control is between managers and workers. This is done automatically by the definition of manager; if no worker is available, the message is not sent. As a result, any communication between a manager (even collapsed) and a worker is controlled. This property holds for both explicit and collapsed managers.

Another place to apply some form of flow control is between external managers and their callers. Currently, the manager receives the complete request from the sender and forwards it to a worker, with the reply making a similar two–step trip (the left hand side of Figure 5.1). For large request messages, this extra step can be expensive and undesirable. A better way is to use the manager to send back the ID of a worker that the caller contacts directly, as shown on the right side of Figure 5.1. Messages 1 and 2 are the request for a worker and the reply. The calling process then makes its request directly to the worker (messages 3 and 4). The last message is used to indicate the availability of the worker to the manager. This new algorithm prevents large messages between the calling process and the manager from flooding the network. This protocol can be implemented using a combination of the manager behaviour and the process instance. The process instance can use the graph to detect the presence of an external manager (which can be derived from graph information) and request a worker. The message that generated this special request can be placed in a queue indicating that it has requested a worker, which is drained as worker IDs are returned from the manager. If this queue grows above a specified threshold, subsequent requests are simply blocked as though no workers are currently available, as before. As the first queue drains, the queued messages can request workers, which prevents a flood of worker requests. The manager behaviour would need to be modified to obey the new protocol, which is a matter of making the internal details of the administrator algorithm available to the behaviour.

The final place to apply flow control is communication between two non–replicated assets. In the current system, there are few options to do this effectively as an asset call may be purely asynchronous, with no futures generated and no reply message expected. However, if statistics are gathered as per Section 5.1.2, these messages will

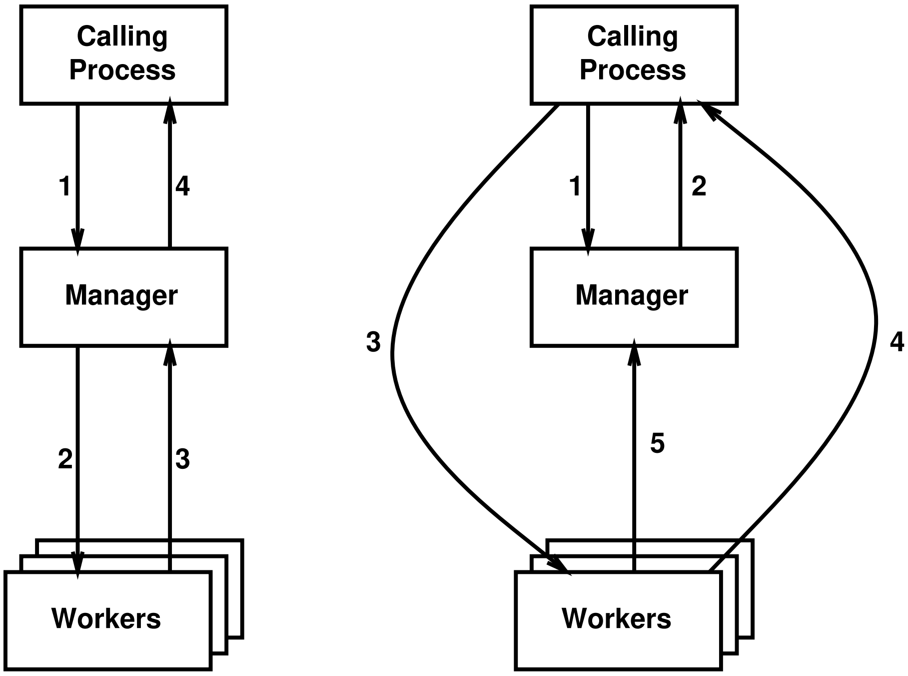


Figure 5.1: A old and new method of handling replicated assets.

be returned even for asynchronous calls, and can be used to throttle messages.

5.1.4 Cost of Dynamic Behaviour

The run–time system has been written so that Enterprise programs can exhibit dynamic behaviour instead of the static behaviour used previously. Before this feature is used, however, the cost of dynamic behaviour and the conditions under which it should be used must be determined. Other work has found that the cost of dynamic behaviour is proportional to the amount of state that needs to be transferred between processes [18] and is relatively cheap if the number of processors changes infrequently [6]. More accurate results will have to be established for Enterprise programs to better determine the available opportunities for adaptive execution.

5.1.5 Cost of Collapsed Managers

Collapsed managers provide performance gains by reducing the number of messages needed to process a request and by implementing flow control. But the processes that contain the managers must multiplex their own duties with their managerial responsibilities, which is accomplished using a timer that ticks once a second.

Some additional work needs to be done to determine the effects of changing the timer interval. If the interval is decreased, incoming worker messages can be processed faster, but perhaps the other duties of the process starve. If the timer interval is increased, the duties of the process will get more processing time, but the decreased response time for incoming worker messages may result in poor utilization of managed processes. Some effort should be made to either determine the best interval or find a method of deriving the interval for an application.

5.1.6 New Division Behaviour

This new implementation of the run–time system used a new model for the division asset, providing a worker pool rather than a static tree. This model is simpler and is expected to result in better process utilization by avoiding a static structure.

As well, Nicholas Kazouris is investigating different parallel recursive program structures in Enterprise. This work attempts to provide faster execution by allowing a process to execute part of its own recursively generated work rather than waiting for results from other processes, usually using hierarchical work queues. This will not affect the model of a division, just the run–time characteristics.

5.1.7 Introducing Threads

One topic that has arisen is the introduction of threads into the Enterprise run–time system. These threads are not intended for the user, but rather to provide multiple contexts for a process so that several different responsibilities can be handled concurrently.

The first application of threads would be executing a collapsed manager. The `alarm` facility is used to provide an interrupt–driven implementation, but threads may be a cleaner, more intuitive solution. This idea may be more important given the ongoing work in parallel I/O [25] and distributed shared memory [23], where the responsibilities of an Enterprise process may also include acting as managers for these new constructs. The alarm handler can be expanded to meet these requirements, but it will become complex and time–consuming, undesirable traits for interrupts.

However, threads will be difficult to introduce into the run–time system. The main problem is writing thread–safe or re–entrant code, so that a thread may be switched out and back without any side effects. Since the user cannot and should not be expected to write thread–safe code, the run–time system must exhibit this behaviour if threads are to be included. This will pose a difficult problem given the complexity of the system and the use of dynamic memory allocated on the heap, a common non re–entrant operation.

5.2 Summary

This thesis speculated that the Enterprise run–time system could be designed and implemented using object–oriented techniques. The project also sought to meet the goals of creating a new system with clean, flexible code, allowing an Enterprise program to dynamically spawn and use new processes during execution, reducing the amount of stubs code generated by the compiler, and encapsulating the communications system to prevent dependencies on a particular implementation. This work also introduced some changes to the programming model, such as a new specification and implementation for the division asset and the introduction of patient/impatient assets. The important aspects of the design were discussed in detail, showing how object–oriented techniques clarified the design and resulted in a better implementation. The remaining design goals were either justified or validated by providing results obtained from the new implementation.

Bibliography

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 30th Conference of the American Federation of Information Processing Societies*, pages 483–485, 1967.

[2] Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Run–time support for dynamic load balancing and debugging in Paralex. Technical Report UBL CS–92–3, University of Bologna, September 1992.

[3] Adam Beguelin, Jack Dongarra, Al Giest, Robert Manchek, and Keith Moore. HeNCE: A hetergeneous network computing environment. Technical Report UT-CS-93-205, University of Tenessee, August 1993.

[4] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and G.R. Zarnke. μC++: Concurrency in the object–oriented language C++. *Software: Practice and Experience*, 22(2):137–172, February 1992.

[5] Derek L. Eager and John Zahorjan. Chores: Enhanced run–time support for shared–memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[6] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 827–832, 1995.

[7] W. Fenton, B. Rankumar, V. A. Saletore, A. B. Sinha, and L. V. Kale. Supporting machine independent parallel programming on diverse architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 193–201, August 1991.

[8] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task–parallel programming languages. Technical Report ANL/MCS–TM–205, Argonne National Laboratory, February 1995.

[9] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Software: Practice and Experience*, 11(5):435–466, May 1981.

[10] A. S. Grimshaw. Easy to use object–oriented parallel programming with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[11] Andrew S. Grimshaw, Jon B. Weissman, and W. Timothy Strayer. Portable run–time support for dynamic object–oriented processing. Technical Report CS–93–40, University of Virginia, July 1993.

[12] R. Halstead. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[13] Paul Iglinski. An execution replay facility and event-based debugger for the Enterprise parallel programming system. Master's thesis, University of Alberta, 1994.

[14] Paul Iglinski, Steve MacDonald, Chris Morrow, Diego Novillo, Ian Parsons, Jonathan Schaeffer, Duane Szafron, and David Woloschuk. Enterprise user's manual version 2.4. Technical Report TR 95–02, University of Alberta, January 1995.

[15] Isis Distributed Systems Inc. *The Isis Distributed Toolkit: Version 3.0, User Reference Manual*, 1992.

[16] ParcPlace System Inc. *VisualWorks2.0 User's Guide*, 1994.

[17] Vijay Karamcheti and Andrew Chien. Concert – Efficent runtime support for concurrent object–oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, pages 598–607, 1993.

[18] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive parallelism under Equus. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 172–182, 1994.

[19] G. Lobe, D. Szafron, and J. Schaeffer. The object–oriented components of the Enterprise parallel programming environment. In *Proceedings of Technology of Ojbect–Oriented Languages and Systems Conference II*, pages 215–229, August 1993.

[20] Greb Lobe. The Enterprise user interface and program animation component. Master's thesis, University of Alberta, 1993.

[21] T. A. Marsland, T. Breitkreutz, and S. Sutphen. A network multi–processor for experiments in parallelism. *Concurrency: Practice and Experience*, 3(1):203–219, 1991.

[22] S. N. McIntosh-Smith, B.M. Brown, and S. Hurley. Intelligent algorithm decomposition for parallelism with Alfer. In *Programming Environments for Massively Parallel Distributed Systems*, pages 47–56, 1994.

[23] Diego Novillo. Transparent shared memory in multiprocessor environments. Candidacy document, September 1995.

[24] Ian Parsons. An appraisal of the Enterprise model. Master's thesis, University of Alberta, 1993.

[25] Ian Parsons. Parallel I/O templates for Enterprise. In *Proceedings of the 1995 CAS Conference, CD Edition*, 1995. To appear.

[26] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[27] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[28] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.