Writing Concurrent Object-Oriented Programs using Smalltalk-80*

Trevor P. Hopkins and Mario I. Wolczko
Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL
United Kingdom
tph@uk.ac.man.cs.ux, miw@uk.ac.man.cs.ux

October 1989

Abstract

This paper considers a number of ways in which concurrent programs may be expressed within an object-oriented framework. It goes on to describe work investigating the expression of highly parallel programs in a conventional object-oriented language (**Smalltalk-80**). The relevance of this work to a new parallel object-oriented system is discussed. An implementation of an 'eager' evaluator is described, together with a 'throttling' mechanism capable of limiting the generation of concurrent processes. A system for suspending all processes performing a parallel computation is discussed. Finally, further work in investigating debugging environments for concurrent object-oriented systems is outlined.

Concurrent Object-Oriented Systems

There has been an increasing interest in recent years in what is called *object-oriented* programming, accompanied by a diversification of opinion. There seems to be little agreement about what is meant by the term 'object', or whether a particular programming language or programming style is truly 'object-oriented'. There has also been an increased interest in using object-oriented programming techniques as the basis for expressing concurrent solutions to a wide range of problems, with the aim of harnessing highly parallel computers [YT87b].

Object-Oriented Programming

Some general principles about object-oriented programming can be observed. At least some of the computation is expressed in terms of *objects* sending *messages* from one to another. Objects encapsulate some 'state' or a 'data structure', together with the only available mechanisms for changing (or enquiring about) that state. In order to discover or modify the state of some object, a suitable message must be sent to that object (the *receiver* of the message). The action (or *method*) performed by the receiver on the receipt of a message is entirely its own concern: other objects need not know or care how a particular object chooses to implement a certain function.

Most object-oriented languages, **Simula** [BDMN73] being the first, have a notion of objects being *instances* of *classes*. All instances of a class have the same functionality, while each instance has separate state. This allows 'code' held within the class to be shared between many instances of that class. Several advantages can be identified: code is located in just one place, thus saving space and allowing modifications to be made readily and in a controlled manner.

Many languages, including **Simula** and **Smalltalk-80**[1] [GR83], **C++** [Str86] and **Objective-C** [Cox86] take this notion further, where classes are arranged in an *inheritance* tree. An instance inherits functionality from its class, as well as all *superclasses* of that class. Methods defined in the class at the root of the tree are inherited by all classes. A class may have several *subclasses*; functionality common to several classes may be represented by a common superclass, which is then specialised differently by different subclasses. This approach encourages the reuse of code; when a new class is added to the system, it may inherit functionality from a similar class, and only functionality specific to that class need be added.

However, the single-inheritance mechanism is a little restrictive; in many cases, a clear separation of functionality into a strict hierarchy cannot be achieved. Some languages permit classes to inherit properties from more than one superclass directly, so that the hierarchy of classes becomes a directed acyclic graph, rather than a rooted tree. This *multiple inheritance* mechanism permits objects which inherit

*A modified version of this paper was published in The Computer Journal, **32**(4), Oct. 1989, pp.341-350.

[1] Smalltalk-80 is the latest version of a series of languages named Smalltalk [Ing83]. For brevity, Smalltalk-80 is referred to simply as Smalltalk throughout this paper.

large amounts of disparate functionality to be constructed rapidly. Such a mechanism has been added to Smalltalk [BI82] and similar mechanisms are available in Lisp-based object systems, including **Flavors** [Moo86], **CommonLoops** [BKK+86] and **CLOS** [DG87].

Concurrency

Because of this model of discrete objects communicating with one another, the object-oriented style has been considered a suitable basis for the exploitation of parallelism. Here there are three different approaches.

In the first approach, message sending is like a procedure call, where control passes from the sender object to the receiver object when the message is sent, and returns when the message has been processed. However, the procedure call is *dynamically bound*, so that the call address is determined at run-time, rather than at compile-time. Processes may be objects, but not all objects are processes. The objects are extremely fine-grained, and an instance of class Character, for example, is quite reasonable. **Smalltalk** [GR83] and **Trellis/Owl** [SCB+86] are examples of such languages. Processes will synchronise using conventional structures, such as semaphore objects or monitor objects. Generally, such schemes will have a high process overhead, and are best suited to a moderate-to-large level of granularity in the expression of concurrency [MK87].

In the second approach, objects are regarded as concurrently executable program modules, with internal activity starting when the object is created, and continuing after a message is sent. Typically, messages are only accepted at explicitly defined times. Not all computation is expressed in terms of message sending and other means of expressing the functionality within objects are required. Languages such as **POOL-T** [Ame85] fit into this category. Clearly, in this approach, objects are necessarily quite large, because of the process scheduling overhead. There is little point in having, say, an integer or character as an independent object. Again, such schemes will have a high process overhead, and are suited to a moderate level of granularity.

Finally, the *actors* [Agh86] approach is midway between the two approaches mentioned above. The sender of a message can proceed concurrently without waiting for the receiver of the message; messages are queued at the receiver. If several messages are sent, then many objects may be active. Typically, all computation is expressed in terms of communicating actors. This permits the expression of fine-grained parallelism, as well as coarser-grained parallelism, as the process overhead is expected to be quite small. Since all message-sending is potentially concurrent, explicit 'serializers' are required.

Actor Languages

One problem with traditional superclass inheritance, such as that provided by Smalltalk, is a lack of separation between inherited *implementation*, and inherited externally-observable *behaviour* [Sny86]. These aspects lie on different levels of abstraction within a system. The first of these is at a level of detail only of interest internally, while the second represents the important functionality visible to the users of a class. For example, in Smalltalk, there is no *enforced* separation between methods corresponding to messages intended to be sent from outside the local part of the class hierarchy, those intended to be sent only within this part of the hierarchy, and those intended to be used only within a particular class. This leads to a lack of encapsulation in the class hierarchy. A subclass frequently depends on the exact implementation of a feature in a superclass; re-implementing the superclass may require many of its subclasses to be modified as well. Some languages, such as **POOL-T** [Ame87], side-step this issue, omitting superclass inheritance altogether and using only the class-instance mechanism.

Another aspect which limits the ability to express parallelism in languages with superclass inheritance is the problem of what to do when an object sends a message to itself. This ability is widely used in object-oriented systems. If all objects are independent computational units, the interpretation of a message send to self is unclear and, if the message is sent using a synchronous communication mechanism, immediately leads to deadlock.

An alternative to the class-instance mechanism is the *delegation* approach. Here, each object may have its own functionality, but also knows about other objects to which it may *delegate* responsibility (*proxies*) when it is unable to respond to a message itself. A proxy may be able to respond to the message on behalf of its *client*, or may have to pass the message onto another proxy. Clearly, new functionality may be added on a per-object basis, by simply adding new code which responds to particular messages. Delegation can also be provided as well as inheritance, as in the **Actra** system [TLP86].

The idea of *prototypical* objects is often used in systems with delegation rather than inheritance, such as **Act 1** [Lie87a]. A new object is created by *cloning* an existing object in the system. The existing object becomes the new object's proxy. Initially, this new object has no additional functionality; new functionality can be added later.

Several languages combine the notion of delegation and prototypes, including **Act 1** [Lie87a] and **Act 3** [AH87]. Such Actor systems have considerably more expressive power than languages with a class hierarchy, and more flexibility in the way in which a new application is constructed. For example, it is possible to construct a conventional superclass hierarchy using prototypical objects in a straightforward manner, while the opposite case is impossible to implement [Lie86].

Actor systems have some advantages in a parallel object-oriented environment. The delegation mechanism sends a message to its proxy, rather than requiring a search of the class hierarchy which is 'hard-wired' into the in-

terpreter. This increases the number of potential actions which may be performed concurrently, as the actors are linked only by cause-and-effect. Furthermore, the objects may more readily move between processors. Nevertheless, a unique address is still required for each object. Efficient implementations of the delegation and prototyping mechanisms may prove to be difficult, especially in a loosely-coupled parallel computer. Finally, at present, there seems to be little experience of building large-scale applications in actor systems, and it is not yet clear how useful the extra flexibility achieved by delegation will be in practice.

Smalltalk

Smalltalk is a programming language entirely expressed in terms of the object-oriented model. Everything in the Smalltalk system is an object, and all computation is expressed in terms of message sending. The Smalltalk language is embedded in an environment which encourages reuse of code through the inheritance of functionality defined in classes. A single hierarchy of classes is normally used, although a limited multiple inheritance scheme is available. The environment is persistent, with objects coming into being on demand and lasting as long as necessary (be it microseconds or months). When no longer required, the space occupied by objects is recovered by *garbage collection*. Smalltalk also provides a mechanism for the construction of user interface components, and a range of programming tools (including browsers, debuggers and editors) which are constructed using those components.

A brief introduction to Smalltalk syntax is included in an appendix. Further details of the Smalltalk language can be found in [GR83], and the programming environment is described in [Gol83].

In order to ensure a highly portable implementation, the Smalltalk system is divided into two distinct parts [GR83]. The *virtual image* contains all the objects which make up the Smalltalk system, and is completely portable between implementations. The virtual image contains a class Compiler, instances of which can convert methods written by programmers into sequences of instructions called *bytecodes*. These bytecodes are contained in instances of class CompiledMethod.

The *virtual machine* consists of a mixture of hardware and software which actually performs operations on the objects in the image. It includes an *interpreter* for the instruction set generated by the compiler. This instruction set includes, for example, *send* and *return* instructions, conditional and unconditional jumps, and stacking operations. Some bytecodes cause *primitive* operations to be performed; these include integer and floating point arithmetic, tests for equivalence and creation of new instances. Some memory management functions are required as well, including the allocation of memory space to new objects, and the recovery of space left by objects no longer required. This latter function implies a need for a *garbage collection* scheme.

This separation into two parts maximises the portability of the Smalltalk system. Most of the functionality is in the virtual image, and only the virtual machine need be re-implemented when moving Smalltalk to new hardware. Unfortunately, straightforward implementations of the virtual machine are slow [McC83, FS83, UP83], and sophisticated implementation techniques need to be used in order to get adequate performance [Deu83, CWB86, Mir87], even on today's powerful single-user workstations [DS84]. The work described in later sections is entirely implemented using the Smalltalk system, and no virtual machine modifications are required. This has the advantage that a high-speed commercial implementation (based on [DS84]) can be used.

ConcurrentSmalltalk

Smalltalk has limited built-in support for concurrency. It supports Processes and Semaphores as primitive types. Other conventional inter-process communication and protection mechanisms, such as SharedQueues and critical regions, are implemented in terms of these primitives. However, the processor scheduler (part of the virtual machine) implements a naïve non-preemptive scheduling policy, with limited support for re-scheduling within a particular priority level. Further, once a high-priority process is running, no lower priority process will run until the high priority process suspends or terminates. For these reasons, the normal Smalltalk system contains only a few processes, and typical applications using concurrency do not create more than a few tens of processes.

ConcurrentSmalltalk [YT87a] is an extension of Smalltalk. It adds asynchronous message sends to the synchronous sends already available in Smalltalk. New types of object (**atomic objects**) are also introduced. Only one process can execute any of the methods associated with an atomic object at any one time. Messages sent to an atomic object are executed serially in a FIFO manner, with a single context created for each activation. A special interpretation is provided for message sends to self, which ensures that they execute within the same context.

ConcurrentSmalltalk is implemented by adding new bytecodes to the Smalltalk virtual machine, together with modifications to the compiler. The processor scheduling algorithm has also been modified to permit time-slicing, so that large numbers of processes can execute in a pseudo-concurrent manner. Thus, ConcurrentSmalltalk requires significant extensions to the virtual machine implementation. Unfortunately, high-performance Smalltalk virtual machines are complex pieces of software, and this approach is quite expensive in terms of the effort involved when experimenting.

The MUSHROOM Project

The aim of the MUSHROOM[2] project [HWW87] is to construct and evaluate a high performance interactive object-oriented system that actively encourages information sharing between different users, and is distributed transparently between many machines.

A new object-oriented language, MUST, based on Smalltalk-80 is being designed [Wol88a]; this includes features to support tight encapsulation, delegation, multiple inheritance and sharing between different users. This work is based on a formal analysis of Smalltalk and other object-oriented languages [Wol87, Wol88b]. The use of advanced compiler techniques forms a significant part of the investigation. New tools to support the development of large-scale applications, especially concurrent systems supporting multiple users, are also under consideration.

Architectural support for concurrent fine-grain object-oriented systems is also under investigation. Using Smalltalk-80 as a model system, a number of architectural aspects have been extensively simulated. These include caching strategies, instruction sets, garbage collection [Wil86], and hardware support for virtual memory [WWH87b], dynamic binding and runtime tag checking. A high-performance object-oriented shared-memory multiprocessor is currently being designed, based on the simulation results obtained.

Concurrent Programs in Smalltalk

The aim of the work described here was to investigate possible mechanisms for expressing solutions to inherently parallel problems in fine-grain object-oriented languages. As a Smalltalk implementation with adequate performance was available, and some experience with developing applications using this system had already been gained, it was decided to use Smalltalk as the basis for this investigation. No modification of this Smalltalk virtual machine could be considered, as source code for the virtual machine was not available.

Low-level Mechanisms

It was considered that the simple primitives implemented in classes Semaphore and Process, which are already present in the Smalltalk system, did not permit a sufficiently abstract approach to the expression of parallel solutions to problems. Consequently, a small number of new low-level operators to permit the expression of parallelism in a clean manner were constructed.

The most useful 'primitive' used in this work is an 'eager' evaluator, implemented as a class Future. A Future returns a marker or promise for a result which may take some time to evaluate. A separate process is created to calculate the result; in principle, this process could execute concurrently with the process which generated the Future. The Future may be used just like the result (passed as a parameter, for example) without interaction with the Future's process. Only when a message is actually sent to the Future is it necessary to synchronise the two processes. If the Future process has not completed by the time this message is received, the original process is suspended. This re-synchronisation is provided transparently by the Future implementation.

This eager evaluation mechanism has been used in a number of other object-oriented languages, including **Act 1** [Lie87a]. It has also been used in parallel Lisp-based languages, such as **MultiLisp** [Hal85].

The other primitive constructed is a 'lazy' evaluator, implemented as class Lazy. A Lazy only starts executing when the value is explicitly required. Thus, Lazy provides a completely demand-driven evaluation strategy, and can be used, for example, to express potentially infinite sequences of results, without requiring infinite machine resources. The 'lazy' evaluator also appears in MultiLisp.

The implementation of Future and Lazy in Smalltalk uses the *block* mechanism. Blocks represent deferred execution, which may be created, passed around as parameters, forked as processes or evaluated directly. Blocks are widely used to implement control structures within Smalltalk. Textually, blocks are contained within rectangular brackets '[ ]'. For example:

```
aBlock ← [10 factorial].
```

assigns a block, capable of being evaluated to give the result **3628800**, to variable aBlock. At a later time, aBlock (which is an object) may be sent the message value:

```
answer ← aBlock value.
```

This causes the block to be evaluated, and the numerical result assigned to answer. This could also have been expressed thus:

```
answer ← [10 factorial] value.
```

In order to express this computation using Futures, the following statement could have been used:

```
answer ← [10 factorial] futureValue.
```

This assigns a Future to answer, while spawning a process to compute the final value. This process is made runnable immediately. Only when the value of answer is required will the Future be forced to completion; for example, by displaying the result as a string.

```
Transcript show: (answer printString).
```

Of course, the Future may have already completed by the time the answer is required.

It is worth pointing out that Future and Lazy represent just two points in a spectrum of possible evaluation strategies. A Future represents the maximum concurrency available, while a Lazy is merely deferred serial execution. For this reason, Futures are much more useful in this work.

[2]The MUSHROOM project is currently supported by the UK Science and Engineering Research Council, under grant number GR/E/65050.

Other 'less-eager' evaluators (executing at lower priority, perhaps) are also possible, but these are not considered here.

Note also that the Future and Lazy constructs do not provide any protection from interference between activations of blocks with side-effects. Careful consideration by the programmer is necessary to ensure that a program is correct regardless of the activation order of the Futures.

More Complex Constructs

Based on Future, a number of more complex structures can be built. For example, blocks can accept arguments, so parameters can be passed to executing Futures. This example assigns a value equal to ten times the product of the arguments to answer.

```
answer ← [:x :y | 10 * x * y] futureValue: 3 value: 4.
```

Sometimes it is necessary to force a Future or Lazy to completion, without actually using the value. A touch method, understood by all objects, provides this feature. Examples of the use of touch are given later.

A further convenience is the ability to start any number of computations in parallel, but force all of them to complete before further actions are performed. This can be implemented in terms of Future and touch, and is conveniently encapsulated in class ParallelEvaluation. This allows a number of blocks to be explicitly executed in parallel. For example:

```
[10 factorial] inParallelWith: [20 factorial].
Transcript show: 'Finished both factorials'.
```

This computes the two factorials in parallel, forcing both to completion before the following expression is evaluated.

In many cases, however, some operation is performed on the results of the blocks in the ParallelEvaluation. A general mechanism for dyadic operations (the simplest non-trivial form) is provided:

```
difference ← [10 factorial]
  parallelPerform: #diff: with: [20 factorial].
```

As before, this example computes both blocks in parallel. Then, it sends the message diff: to the result of the first block, with the result of the second block as an argument. The overall result, assigned to difference, is the absolute difference of the two values.

The parallelPerform:with: method is quite cumbersome, however, and for frequently used operations (particularly arithmetic and logical operations), additional methods are provided:

```
sum ← [10 factorial] parallelAdd: [20 factorial].
```

This example computes the sum of the two factorials.

Finally, two complete examples are presented. The first is a single method for computing fibonacci numbers, using a doubly-recursive algorithm. This is implemented as a method in class Integer.

```
fibonacci
  (self <= 1)
    ifTrue: [↑self]
    ifFalse: [↑[(self - 1) fibonacci]
            parallelAdd: [(self - 2) fibonacci]]
```

Note how some arithmetic operations are expressed in the same block, and thus executed serially.

The second example[3] is more lengthy. It uses a parallel recursive sub-division algorithm to perform the integration of an arbitrary function [RR78], expressed as a Smalltalk block of one argument. The algorithm uses trapezium approximations to the area under the curve described by the function. It recursively divides the area into smaller trapezia, until the error computed when doing this is smaller than a specified tolerance.

A class BinaryIntegration encapsulates this algorithm, and defines an instance variable referring to the block representing the function to be integrated. New instances of BinaryIntegration, capable of integrating a particular function, are created using this class method:

```
function: aBlock
  ↑self new function: aBlock
```

This uses a private instance method (not shown), also called function:, to initialize the instance variable.

The basic recursive step of the algorithm is represented by the following method:

```
areaBetween: left and: right estimate: anEstimate tolerance: aTolerance
  | mid areaLeft areaRight newEstimate |
  mid ← (left + right) / 2.
  areaLeft ←
    [self trapeziumBetween: left and: mid] futureValue.
  areaRight ←
    [self trapeziumBetween: mid and: right] futureValue.
  newEstimate ← areaLeft + (areaRight touch).
  (anEstimate - newEstimate) abs < aTolerance
    ifTrue: [↑newEstimate]
    ifFalse: [↑[self
          areaBetween: left
          and: mid
          estimate: areaLeft
          tolerance: (aTolerance / 2)]
        parallelAdd: [self
          areaBetween: mid
          and: right
          estimate: areaRight
          tolerance: (aTolerance / 2)]]
```

This method divides the interval between left and right into two equal parts, and estimates the area for each part us-

[3] This example is based on an implementation in **SISAL** [MSA+83] by John Sargeant, which is in turn based on an implementation in **MAD** by Dave Bowen [Bow81].

ing a trapezium approximation. These two parts are performed in parallel, as they are each Futures. If the difference between the new estimate and the old estimate for the same interval is less than the specified tolerance, then the new estimated value is returned. Otherwise, the same method is called twice, once for each part. This is performed in parallel, using the parallelAdd: method.

A number of auxiliary methods are provided. The first of these calculates the area of a trapezium with width given by left and right, and height given by the value of the function at those points. Again, the values at the left at right points are computed in parallel.

```
trapeziumBetween: left and: right
    ↑(right - left) * ([function value: left]
        parallelAdd: [function value: right]) / 2
```

Finally, the following method starts the execution of a complete calculation. It starts the computation (with a Future) of an initial estimate of the area, again using the trapezium approximation. For simplicity, the tolerance is shown as a constant.

```
areaBetween: left and: right
    ↑self
      areaBetween: left
      and: right
      estimate:
        [self trapeziumBetween: left and: right] futureValue
      tolerance: 0.01
```

Thus, as an example, the following code integrates a cubic polynomial between 0 and 5:

```
| integration |
integration ← BinaryIntegration
    function: [:x| (3*x*x*x) + (2*x*x) + 5].
Transcript cr; show:
    (integration areaBetween: 0 and: 5) printString.
```

Implementation of Future

The key to the implementation of both Future in Smalltalk is the creation of entities which are *not* descendants of class Object, unlike all other classes in the system except Object itself. Consequently, instances of class Future, for example, will not understand *any* messages, except those specifically defined in class Future.

In Smalltalk, there are a small number of messages which are actually generated by the underlying virtual machine, rather than being 'real' messages sent from an object within the virtual image. One such message is doesNotUnderstand:, which is sent when an object receives a message where a corresponding method cannot be found in any of the classes from which that object inherits functionality. The argument to the message sent by the virtual machine is the message which was not understood by the original receiver. The normal use of the doesNotUnderstand: message is to signal a runtime error.

In this case, however, a method corresponding to doesNotUnderstand: is implemented by Future and Lazy. Any other message send to a Future (similarly for Lazy) will cause the doesNotUnderstand: message to be sent to that Future. At this point, the process sending the original message waits for the process associated with the Future, assuming that the Future's process has not already completed. The original message is then forwarded to the resulting object, now that it is available. Subsequent messages to the same Future are forwarded in the same way.

Each Future is created with a Semaphore used to support the resynchronisation action, and a process which evaluates the value promised by the Future. The Semaphore is signalled when the process is complete, and waited on when the Future receives the doesNotUnderstand: message.

A similar implementation technique is used for Lazy. It can be used to provide encapsulation of arbitrary objects [Pas86], including monitors to ensure consistency when an object receives messages from different processes.

Problems with Future Implementation

Primitives

One problem with using this method of implementing Future is that they do not work correctly with 'primitive' methods; i.e. methods not really selected by message sending, but directly executed by the underlying virtual machine. In the following example, which attempts to calculate the product of a factorial and a constant number, the message '*' is not really sent to the SmallInteger 2; instead, a primitive is invoked which manipulates fac directly. The primitive fails, as the arguments are not numbers, and an unexpected runtime error occurs.

```
| fac result |
fac ← [10 factorial] futureValue.
result ← 2 * fac.
```

If the programmer is aware of this problem, explicit touch operations can be used:

```
| fac result |
fac ← [10 factorial] futureValue.
result ← 2 * (fac2 touch).
```

In a better language, primitives would be better disguised; they would always behave as if a message had really been sent. It is possible to go some way toward providing this within Smalltalk, although some primitives, such as become:, seem impossible to encapsulate [Wol88b].

Contexts

Blocks with one argument are often used to form a control structure equivalent to a 'DO' (or 'for') loop in other languages. In the following example, the block is evaluated ten times, once for each element of the array squares.

The result is an array containing the squares of the first ten natural numbers.

```
| squares |
squares ← Array new: 10.
1 to: 10 do: [:each |
  squares at: each put: (each * each)].
```

In some cases, it might be useful to evaluate each activation of the block concurrently, especially if the calculation performed in the block was computationally expensive. Such a construct can indeed be implemented, allowing expressions like that shown below.

```
| squares |
squares ← Array new: 10.
1 to: 10 parallelDo: [:each |
  squares at: each put: (each * each)].
```

Unfortunately, the 'obvious' implementation using a ParallelEvaluation fails, as each activation of the block shares the same context. To fix this, it is necessary to copy the context for each block; this is done by modifying the value methods for blocks. Again, there is a small problem here; the value messages are treated as special cases by the compiler for efficiency reasons, so that the message sends are not interpreted in the expected manner. This requires that the Smalltalk compiler be modified so as not to generate these special cases, and then the entire system is recompiled to remove any existing use of these special selectors. This reduces the performance of the system, but seems to be tolerable in practice.

Scheduling

In the work described here, the standard Smalltalk scheduling mechanisms have been used. Processes at the same priority run to completion, unless they give up control explicitly. However, pre-emption by processes of a higher priority is possible. This behaviour is unfortunate when constructing concurrent programs, but it does not seem to be possible to improve this without modifications to the virtual machine (as made in **ConcurrentSmalltalk** [YT87a]) and the virtual image. This problem is exacerbated by differences between the specified behaviour of the scheduler, as defined in [GR83], and the implementation actually used. An attempt was made to implement an improved process scheduler within Smalltalk itself[4], but this led to such poor performance that its use was discontinued.

A further problem relates to the difference in dynamic storage requirements for Future and Lazy. When a Lazy is created but not started (perhaps because it is later discovered that the result is not required at all), only a small amount of storage is required, and there is little processor time wasted. Furthermore, the storage space is rapidly recovered by the garbage collector.

Unfortunately, similar behaviour is not exhibited by Future. As presently implemented, a Future which is started, but is subsequently discovered to be superfluous will use a potentially large amount of both storage space and processor time unnecessarily. In an ideal system, it should be possible for a Future to be stopped and the memory space recovered immediately, when it is discovered that this part of the computation is unnecessary. Clearly, it is important that this action is transparent to the programmer, and should be part of the functionality of the process scheduler and garbage collector.

[4]The improved scheduler was written by Ian Piumarta.

Runtime Control of Parallelism

The amount of parallelism available under different circumstances seems to vary widely depending on the application. Some problems are naturally serial in nature, while others permit a large number of activities to be performed concurrently. One problem which appears with the latter class of algorithms is that they are 'embarrassingly parallel' [RS87]; if left uncontrolled, they would generate a large number of concurrent activities or processes.

A large number of concurrently active processes causes several problems for machines with a limited number of processing elements. One problem is that each thread will have some 'stack' of activation records associated with it, and these must be stored. Typical processors will have facilities to support a few processes in hardware; making extensive use of many more processes will lead to inefficiencies. With even more processes active, the space required for the activation records may become a sizable fraction of the available real memory, and may reduce the effectiveness of any caches. If a virtual memory system is used, huge numbers of processes may cause excessive paging activity. Thus, it is desirable to limit the number of processes. Note that this is equally important in a serial implementation of a concurrent programming language.

Having too few processes available also has its limitations. If there are fewer active processes than processors, then some processors will be unused. Even if the number of runnable processes is greater than the number of processors on average, variations in this number may again cause some processors to be idle. This may be caused, for example, by one process being forced to wait for the result produced by another. For a particular system, there will be a range of values which are a reasonable compromise between too many, and too few, processes. Maintaining the number of runnable processes in this range is sometimes called *throttling*, and is an important attribute of any large-scale parallel computer system.

Implementation of Throttling in Smalltalk

In order to investigate the problems of parallelism control, a throttling mechanism for Futures was provided. To provide a concurrency control mechanism, a new class Throttle is created. Instances of this class maintain a count of the number of currently active processes using this par-

ticular Throttle, and the maximum number of active processes permitted. Every Future is assigned to exactly one Throttle.

When a new Future is created, a message is sent to the corresponding throttle to find out if a new process can be generated. If a new process is permitted, the Throttle increments its count, otherwise the block is executed immediately in the same process as that which created the Future. If a separate process was indeed generated, it informs the Throttle once it has completed, so that the count can be decremented. Naturally, it is important to ensure all manipulations of the count inside the Throttle are atomic. This is managed by ensuring that all the methods performing these operations are enclosed in a monitor.

In a simple case, a single global instance of Throttle, called SystemThrottle, is used by all Futures in the system. This is convenient for most purposes, and analogous to the single instance of ProcessorScheduler called Processor used in Smalltalk to represent the (single) physical processor. In a more general case, however, several instances of Throttle might be used, so that separate applications (perhaps belonging to different users) could be controlled separately. It might even be possible to consider multiple intercommunicating throttles, perhaps located on different physical processors.

It should be noted that the throttle setting (the maximum allowed number of processes) can be varied as the application runs. Instances of Throttle understand messages which increase and decrease the throttle; if an attempt to set the throttle to a value lower than the current number of active processes, then no further processes are created and the new throttle setting takes effect once the number of active processes has dropped below the threshold. It is even possible to set a throttle to zero: this implies that *no* additional processes are generated and, once existing processes have completed, all computation is performed serially.

The throttling mechanism described is effective in increasing the efficiency of highly parallel programs using Futures, even using the standard, uniprocessor Smalltalk system. For example, the parallel fibonacci method described earlier runs several times faster using the throttled version of the Future implementation, with the throttle set to 10. This is probably because of reduced use of the object memory, even though there is more overhead in the creation of Futures when throttling is implemented.

The effectiveness of this throttling mechanism on a parallel processor has yet to be investigated. Determining an appropriate throttle setting for a given hardware configuration is a matter for experimentation with a real system. Nevertheless, it is expected that the ability to control dynamically the amount of parallelism on the basis of individual applications will prove to be a useful feature of the MUSHROOM system.

Controlling Processes for Debugging

A second, related use of throttles is that they provide a single object which can be used to represent an entire process group. In particular, a throttle can be used as a convenient 'handle' for a parallel application for debugging purposes.

In Smalltalk, an error condition detected by the virtual machine may be signalled using the doesNotUnderstand: mechanism discussed previously. The default action is to create a *notifier* which is displayed on the screen for the user. This gives the user some indication of what has gone wrong, and provides the option of continuing the computation, abandoning it, or starting a debugger. Similar actions occur if a program-detected error occurs, a breakpoint is encountered, or the user deliberately interrupts processing.

In order to permit debugging, the notifier retains the entire activation stack of the process which was interrupted. As the activation records are objects in the Smalltalk image, they understand various messages, just like any other object. This makes the implementation of the debugger, for example, particularly straightforward. By sending the appropriate messages, the debugger can gain access to each suspended context, together with the corresponding source code, permitting modification of variables and recompilation of methods, before potentially restarting the computation. The use of the Smalltalk debugger is further described in [Gol83].

Clearly, a more sophisticated mechanism is required to permit a collection of processes (perhaps spawned by a future mechanism) to be debugged. An approach to this was explored using the throttled Future implementation in Smalltalk described earlier. Each Future already has a reference to its associated throttle, and the implementation of Throttle was modified to retain a reference to each Future using this throttle. When a runtime error condition or user interrupt occurs, the Future associated with the interrupted process can be interrogated to identify its throttle. This throttle then sends messages to all of its processes to cause them to suspend themselves. The throttle now has a reference to all its processes, all of which are suspended, and can be used as a starting point for a parallel debugging system. The Smalltalk version of this control mechanism takes advantage of the uniprocessor nature of the implementation. In general, however, it is important to provide a mechanism for supporting this in a parallel processor.

Other Aspects of Parallelism Control

In the implementation described here, no attempt is made to schedule any particular order of evaluation of Future evaluators. It is clear that some orders of evaluation will be more efficient than others, given that a process-switch operation will always cost some time on any real processor. Unfortunately, determining which evaluation order is optimum is, in general, an NP-complete problem [GLLK79]. Fortunately, it appears that the performance loss caused by

an inappropriate order of evaluation is less than a factor of two [Sar87], so that it may well not be worth the expense of computing an optimum (or near optimum) order.

Another aspect not considered here, but of vital importance in a multiprocessor environment, is managing the balance of workload between many processing elements. In an object-oriented framework, it is reasonably straightforward to move an object from one machine to another. However, the real problem is deciding when to move an object, and to which processor it should be moved. Clearly, information about the load of each processor is required to make such decisions; in a realistic system, such information is likely to be incomplete and out-of-date.

A number of simulation studies of various aspects of the performance of an object-oriented multiprocessor system are being carried out as part of the MUSHROOM work [Wil86, WWH87a]. An investigation to discover suitable heuristics for the dynamic migration of objects is planned as part of this work. Initially, static distribution schemes will be considered, to determine suitable groupings, with various dynamic mechanisms being based on the results of these studies. A local implementation of the Smalltalk virtual machine [Wol84] will be used as a vehicle for this work.

Conclusions and Future Work

The utility of Future demonstrated by this initial implementation has suggested that this form of construct would be a useful addition to the MUSHROOM system. It has also encouraged the active consideration of other concurrent object-oriented programming styles to be incorporated into such a language. The addition of an 'actor'-type system, using delegation and prototypical objects is very interesting and will be explored further.

This work has highlighted the requirements for synchronisation and communication mechanisms between processors in the MUSHROOM system. It has led to consideration of the performance and level of hardware support required for interprocessor communication, especially for semaphores. It has also highlighted aspects of the interaction between concurrent processes and the memory management system in an object-oriented system, especially the desirability of being able to dispose of ('garbage-collect') unwanted processes which are still executing, in a transparent manner.

The work on controlling concurrent processes, particularly the ability of Futures to control an application-specific group of processes, has led to consideration of a framework for the development of tools to aid in the construction of large-scale concurrent applications. Some work in this direction is under way. For example, a view mechanism which shows a graph of active processes dynamically evolving as the program runs is currently under construction. Other tools, including a debugger which works directly with instances of Future and Lazy, are being considered.

In the longer term, program development tools for other types of object-oriented programming environments are of interest. Browsers and debuggers for actor-like systems [Man87], and reversible interpreters [Lie87b] have been investigated elsewhere; the relevance of these experiences to the MUSHROOM work will provide useful input to the project.

Acknowledgements

John Sargeant must be thanked for numerous useful discussions on the problems of throttling, and for providing the binary integration example used in the text.

The authors would also like to thank members of the MUSHROOM group, especially Ifor Williams, for many critical comments on earlier drafts of this paper.

References

[Agh86] G. Agha. *Actors: a Model of Concurrent Computation*. Computer Systems. MIT press, 1986.

[AH87] G. Agha and C. Hewitt. Concurrent Programming using Actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT press, 1987.

[Ame85] P. America. Definition of the programming language POOL-T. ESPRIT Project 415 document 0091, Philips Research Laboratories, September 1985.

[Ame87] P. America. Inheritance and Subtyping in a Parallel Object-oriented Language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 234–242. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[BDMN73] G. M. Birtwisle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Input Two-Nine, 1973.

[BI82] A. H. Borning and D. H. H. Ingalls. Multiple Inheritance in Smalltalk-80. pages 234–237. Technical Report, Dept. of Computer Science, Univ. of Washington, Pittsburgh PA, 1982.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and Object-oriented Programming. In *OOPSLA '86 Proceedings*, pages 17–29. ACM, 1986.

[Bow81] D. L. Bowen. *Implementation of Data Structures in a Dataflow Computer*. PhD thesis, University of Manchester, May 1981.

[Cox86] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[CWB86] P. J. Caudill and A. Wirfs-Brock. A Third Generation Smalltalk-80 Implementation. In *OOPSLA '86 Proceedings*, pages 119–129. ACM, 1986.

[Deu83] L. P. Deutsch. The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 113–126. Addison-Wesley, 1983.

[DG87] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: an Overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 151–170. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. pages 297–302. Salt Lake City, UT, January 1984.

[FS83] J. R. Falcone and J. R. Stinger. The Smalltalk-80 implementation at Hewlett-Packard. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 79–112. Addison-Wesley, 1983.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimisation and Approximation in Deterministic Sequencing and Scheduling. In *Annals of Discrete Mathematics*, pages 287–326. North-Holland Publishing Co., 1979.

[Gol83] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1983.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Hal85] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[HWW87] T. P. Hopkins, I. W. Williams, and M. I. Wolczko. MUSHROOM — a Distributed Multi-user Object-oriented Programming Environment. Presented at the British Computer Society, Object-oriented Programing and Systems and Parallel Programming Specialist Groups Workshop, October 1987.

[Ing83] D. H. H. Ingalls. The Evolution of the Smalltalk Virtual Machine. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 9–28. Addison-Wesley, 1983.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA '86 Proceedings*, pages 214–223. ACM, 1986.

[Lie87a] H. Lieberman. Concurrent Object-oriented Programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT press, 1987.

[Lie87b] H. Lieberman. Reversible Object-oriented Interpreters. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 11–19. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[Man87] C. R. Manning. Traveler: The Apiary Observatory. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 89–97. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[McC83] P. L. McCullough. Implementing the Smalltalk-80 system: The Tektronix experience. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 59–78. Addison-Wesley, 1983.

[Mir87] E. Miranda. BrouHaHa — a Portable Smalltalk Interpreter. In *OOPSLA '87 Proceedings*, pages 354–365. ACM, 1987.

[MK87] J. E. B. Moss and W. H. Kohler. Concurrency Features for the Trellis/Owl Language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 171–180. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[Moo86] D. A. Moon. Object-oriented Programming with Flavors. In *OOPSLA '86 Proceedings*, pages 1–8. ACM, 1986.

[MSA+83] J. R. McGraw, S. K. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. W. W. Glauert, I. Dobes, and P. Henensee. SISAL – Streams and Iteration in a Single-Assigment Language. Language reference manual, version 1.0, Lawerence Livermore National Laboratory, 1983.

[Pas86] G. A. Pascoe. Encapsulators: A New Software Paradigm in Smalltalk-80. In *OOPSLA '86 Proceedings*, pages 341–346. ACM, 1986.

[RR78] A. Ralston and P. Rabinowitz. *A First Course in Numerical Analysis*, chapter 4.11. McGraw-Hill, 1978.

[RS87] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Machine. In G. Kahn, editor, *Functional Programming Languages and Computer Architectures*, pages 1–15. Lecture Notes in Computer Science 274, Springer-Verlag, 1987.

[Sar87] V. Sarkar. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. Technical Report CSL-TR-87-328, Computer Systems Laboratory, Stanford University, April 1987.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM, 1986.

[Sny86] A. Snyder. Encapsulation and Inheritance in Object-oriented Programming Languages. In *OOPSLA '86 Proceedings*, pages 38–45. ACM, 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TLP86] D. A. Thomas, W. R. LaLonde, and J. R. Pugh. ACTRA – a Multitasking, Multiprocessing Smalltalk. Technical Report SCS-TR-92, School of Computer Science, Carleton University, May 1986.

[UP83] D. M. Ungar and D. A. Patterson. Berkeley Smalltalk: Who knows where the time goes? In G. Krasner, editor, *Smalltalk-80: bits of history, words of advice*, pages 189–206. Addison-Wesley, 1983.

[Wil86] I. W. Williams. Memory Management on a High Performance Object Oriented Processor. *Transfer Report, Dept. of Computer Science, University of Manchester*, 1986.

[Wol84] M. I. Wolczko. Implementing Smalltalk-80 on the ICL PERQ. Master's thesis, Dept. of Computer Science, The University of Manchester, October 1984.

[Wol87] M. I. Wolczko. Semantics of Smalltalk-80. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 108–120. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[Wol88a] M. I. Wolczko. Introducing MUST — the MUSHROOM programming language. Technical report of the MUSHROOM project, Dept. of Computer Science, The University of Manchester, June 1988.

[Wol88b] M. I. Wolczko. Semantics of object-oriented languages. Technical Report UMCS–88–6–1, Dept. of Computer Science, The University of Manchester, May 1988.

[WWH87a] I. W. Williams, M. I. Wolczko, and T. P. Hopkins. Realisation of a Dynamically Grouped Object-oriented Virtual Memory System. In *Persistent Object Systems: their Design, Implementation and Use. Persistent Programming Research Report 44*, pages 298–308. University of Glasgow, 1987.

[WWH87b] I. W. Williams, M. I. Wolczko, and T. P. Hopkins. Dynamic Grouping in an Object-oriented Virtual Memory Hierarchy. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP '87: European Conference on Object-Oriented Programming*, pages 79–88. Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[YT87a] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT press, 1987.

[YT87b] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. Computer Systems Series. MIT press, 1987.

Appendix: Introduction to the Smalltalk Language

Expressions in the Smalltalk language consist of the names of objects and messages sent to those objects. The result of an expression is the object answered by the corresponding method. Thus, the expression:

```
anArray size.
```

sends the message size to the object named anArray. The result will be an object; if anArray is indeed an array of objects, then the result will be a number representing the size. Expressions are terminated by full stops '.'.

Messages can have one or more parameters associated with them. Any object can be used as a parameter. For example:

```
seventhElement ← anArray at: 7.
```

sends the message at: to the object named anArray, with the parameter being the object 7. This example also illustrates assignment; the resulting object is named seventhElement.

Where more than one parameter is required, the message selector is distributed, with the arguments placed after each part of the message selector:

```
anArray at: 5 put: 'fifth'.
```

This example sends anArray the message at: put:, with the first parameter being '5' and the second parameter being the string 'fifth'.

In Smalltalk, *classes* are also objects, and can understand messages. This is used, for example, to create new instances of classes. Conventionally, the names of classes have initial capital letters.

```
somePoint ← Point x: 4 y: 9.
```

This expression sends the message x: y: to class Point, which represents positions in two-dimensional space. The resulting object, which is an instance of class Point with x-coordinate 4 and y-coordinate 9, is assigned to somePoint.

Round brackets '( )' can be used to enforce the order of message sending. For example, if pointArray is an array of points, then the expression:

```
(pointArray at: 4) x: 12
```

sets the x-coordinate of the fourth point in pointArray to 12.

Smalltalk methods are defined by stating the message selector and giving formal parameter names to any arguments. Conventionally, the message selector and parameters are reproduced in a bold sans-serif font. Optionally, they are followed by declarations of temporary variables, zero or more expressions which send messages to variables representing objects, and zero or more assignments to variables. The object actually receiving a message can be accessed as self within the method being executed. The object returned when a method completes is indicated by an expression preceded by a vertical arrow '↑' symbol.

```
averageWith: aNumber
    | temp |
    temp ← self + aNumber.
    ↑(temp / 2)
```

This method (assumed to be implemented in class Number) assigns the sum of the receiver and the parameter aNumber to the temporary variable temp. The method then answers with a number obtained by dividing temp by two.