Automatic, Incremental, On-the-fly Garbage Collection of Actors

by

Jeffrey Ernest Nelson

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dennis G. Kafura, Chairman

Sallie M. Henry

James D. Arthur

February, 1989

Blacksburg, Virginia

Automatic, Incremental, On-the-fly Garbage Collection of Actors

by

Jeffrey Ernest Nelson

Dennis G. Kafura, Chairman

Computer Science and Applications

(ABSTRACT)

Garbage collection is an important topic of research for operating systems, because applications are easier to write and maintain if they are unburdened by the concerns of storage management. The actor computation model is another important topic: it is a powerful, expressive model of concurrent computation. This thesis is motivated by the need for an actor garbage collector for a distributed real-time system under development by the Real-Time Systems Group at Virginia Tech. It is shown that traditional garbage collectors—even those that operate on computational objects—are not sufficient for actors. Three algorithms, with varying degrees of efficiency, are presented as solutions to the actor garbage collection problem. The correctness and execution complexity of the algorithms is derived. Implementation methods are explored, and directions for future research are proposed.

Acknowledgements

This work would not have been possible without the help and support of many people. First, I thank my family who always encouraged me to strive for excellence: ; ; ; . A special thanks to .

Second, I wish to thank the people who most influenced my life as I grew up; they helped shape me into the person that I am today: ; ; ; .

Third, I wish to thank the people who influenced my education at Virginia Tech. First and foremost is my advisor, Dennis G. Kafura, who provided inspiration and guidance as we both strove to learn more about real-time systems and garbage collection. Working under his direction has been a very enjoyable experience. I also wish to thank the members of my graduate committee, Sallie M. Henry and James D. Arthur, for their contributions to my graduate career and for their feedback of this thesis. Next, I wish to acknowledge the support and advice of the Real-Time Systems Group.

Next, I wish to thank my friends for their encouragement and support:

Thanks also to all the Creatures for providing me with an escape from reality. A very special thanks to for proof-reading this manuscript. Thanks also to for letting me use his office key at strange hours.

Next, I wish to thank Digital Equipment Corporation for its tremendous financial and emotional support, without which this degree would not have been possible. I especially wish to acknowledge the support of the Graduate Engineering Education Program staff, under the direction of ., the members of my development group, CDD/Plus, the management of the Distributed Systems Information Group, ., and the management of the Technical Languages and Environments group, .
To each I express thanks for the opportunity to pursue this degree.

Finally, I wish to express my most sincere love and thanks to my wife, .
Her constant source of love, devotion and encouragement has kept me sane and on the right track. She has been and continues to be a tremendous source of support and inspiration for me. Thank you, .

Table of Contents

List of Illustrations

List of Tables

1.0 Introduction

The Real-Time Systems Group at Virginia Polytechnic Institute and State University is investigating the following major research question [KAFU88]:

How can a concurrent object-oriented language be used to implement real-time systems?

This research involves diverse, but related areas such as:

- Real-time systems design and implementation;
- Distributed computing environments;
- Concurrent computation models;
- Object-oriented programming; and
- Language design, implementation and run-time support.

This thesis is motivated by the need for an automatic, incremental, real-time, distributed garbage collector of concurrent computational objects known as actors. A garbage collector with these characteristics is required for several reasons. First, the actor model does not provide for explicit reclamation of actors. This is because the model is designed to explore concurrency issues without concern for implementation issues. For example, one of the assumptions made by the model is an

infinite storage capacity. Because systems do not have infinite storage, garbage collection is necessary.

Second, applications are easier to write, debug and maintain if they are free from the concerns of storage management. Bloom and Zdonick state that "automatic storage management, and in particular automatic storage reclamation are crucial for managing software complexity, enhancing software reliability and reducing debugging time" [BLOO87].

Third, the published garbage collection algorithms do not solve the actor garbage collection problem. That is because actors are fundamentally different from data objects. Actors are computational objects that are associated with state information. Even garbage collectors that reclaim computational objects are not able to reclaim actors.

The new actor garbage collection algorithms presented in this thesis are based on the theoretical actor model developed by Agha [AGHA86] and Agha and Hewitt [AGHA87] and is briefly reviewed below. Research which builds upon and expands the theoretical model—like the Rosette system architecture developed at MCC [AGHA88]—may be able to benefit from the work described in this thesis. After the description of the actor model, a description of the real-time computing environment is presented. It is then shown why previous garbage collection algorithms are unsuitable for the actor garbage collection problem. Finally, the design goals of a new actor garbage collection algorithm are presented.

1.1 The Actor Computation Model

An actor is a concurrent computational object composed of:

1. A mail queue. The mail queue is the storage for unread messages (also called tasks) that are

addressed to the actor. Each mail queue is associated with exactly one actor and has infinite storage capacity. Mail queue addresses are unique.

2. A set of behaviors. A behavior is a script that defines how the actor processes a message.
3. A set of acquaintances. Each behavior has a set of mail queue addresses that it knows, called acquaintances. The set is never empty because a behavior always knows the address of its own actor's mail queue. Acquaintances may be explicitly named in the behavior's script (a static acquaintance) or they may be added dynamically. Dynamic additions to the acquaintance set results from the creation of an actor or the receipt of an actor's mail queue address in a message. Deletions to the acquaintance set are caused when a replacement behavior is nominated and the replacement behavior's set of acquaintances does not contain all of the acquaintances of the current behavior.

A conceptual view of an actor is shown in Figure 1.

A behavior receives the next message (implicitly) from the mail queue when it begins execution. Messages are received in the order in which they arrive. If the mail queue is empty, the behavior blocks until a message is available. A behavior receives exactly one message and each message is bound to exactly one behavior. Upon receipt of a message, a behavior may execute one or more of the following kinds of statements:

- Computations. These are normal arithmetic computations.
- Become. The **become** operation is used to nominate a replacement behavior and define the acquaintance set of the replacement behavior. Because a behavior receives exactly one message, nominating a replacement behavior is how an actor processes multiple messages in its mail queue. Nominating a replacement behavior is also how the actor redefines its interface and how it propagates (or drops) acquaintances. Dropping an acquaintance is the only way that garbage is created. If a behavior reaches the end of its script without nominating a replacement behavior, the same behavior (with the same acquaintance set) is nominated by default. The become operation may be specified no more than once by a behavior.

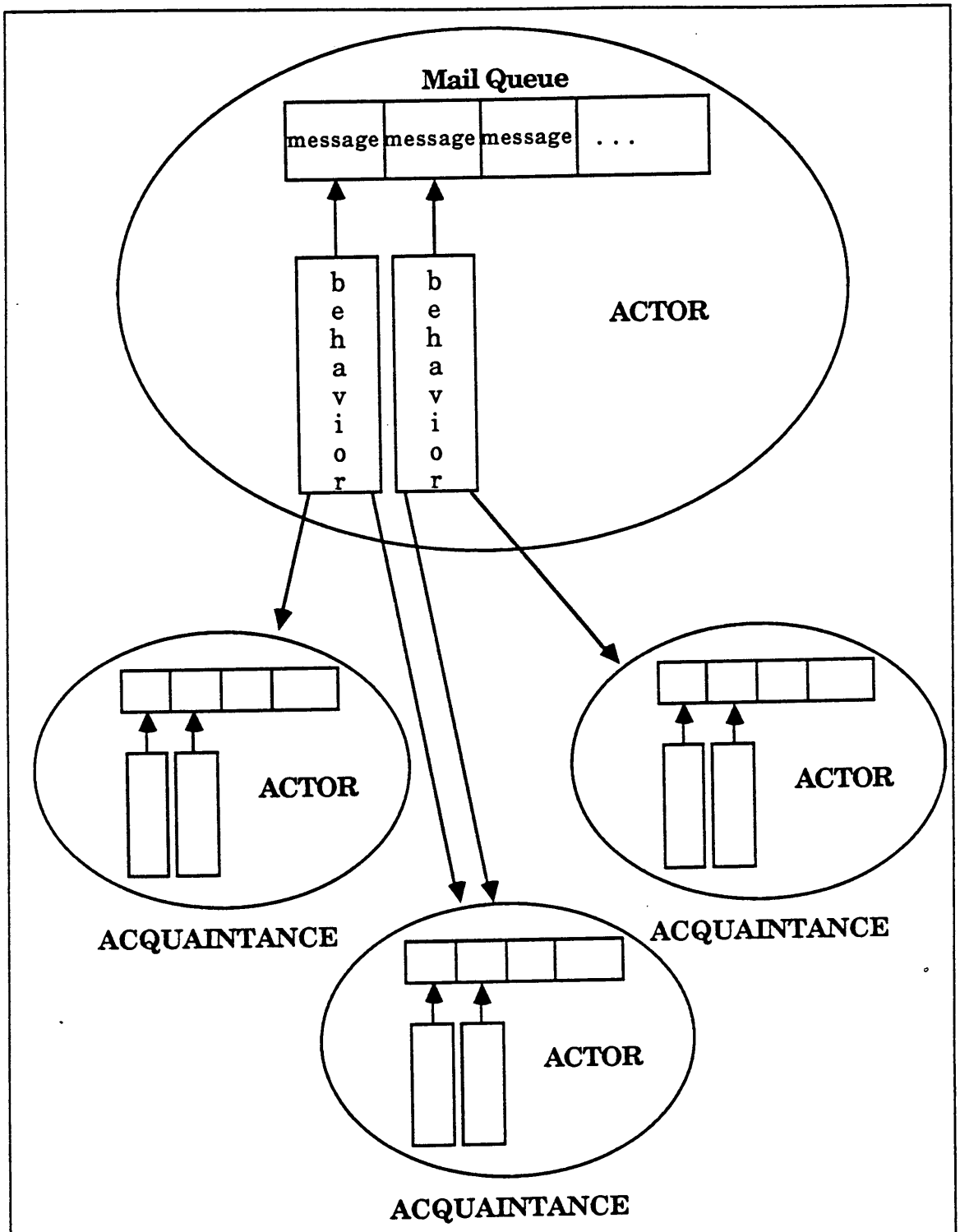

Figure 1. An Actor

- New. The **new** operation creates a new mail queue and a new actor associated with the queue. The new actor's initial behavior and initial set of acquaintances are the two parameters to the function. The operation returns the mail queue address, so the behavior invoking the **new** operation has its acquaintance set increased by one.
- Send. **Send** transmits a message to an actor's mail queue. There are two arguments to the operation: the destination mail queue address and the message to be sent. The actor model guarantees that all messages are delivered to their destination. However, the model imposes no upper bound on the message delivery time, nor does it guarantee to preserve message order in the destination mail queue. In addition to data, messages may contain mail queue addresses. The operation completes asynchronously.

The only control structure in the primitive actor model is the **if-then-else** statement. Therefore, behaviors are guaranteed to terminate because they never enter an infinite loop. (Note that this does not guarantee infinite loops of another kind. Consider an actor that defines an initial behavior to send a message to itself, and then nominates itself as the replacement behavior.) An actor computation that wishes to perform a looping task must do so by sending messages (possibly to itself) containing relevant data values along with a looping index or some other termination criteria.

An example actor definition from [AGHA86] is reproduced in Figure 2. The actors presented in the figure define a recursive factorial function.

To request an evaluation of **n!**, an actor sends a message containing the value **n** and its own mail queue address to the factorial actor. Upon receipt of a message, the factorial actor binds the parameters to the variables **n** and **u**, respectively. It then specifies itself as the replacement behavior before processing the contents of the message. This increases the parallelism of the actor, because it is now ready to process the next incoming message without having to determine the contents of the current message. Finally, the integer value is tested; if it is zero, then the factorial actor returns the factorial of zero and it is finished. If the value is not zero, then a customer actor is created. The customer actor knows the address of two other actors: **n** and **u**. Note that the integer **n** is an actor,

```
actor factorial ≡
   behavior factorial ≡
      with acquaintances self
      interface is integer n, mail queue address u

      become factorial with acquaintances self
      if n = 0
      then
         send (1) to u
      else
         c = new customer with acquaintances n and u
         send (n - 1, c) to self
      endif
   endbehavior factorial
endactor factorial


actor customer ≡
   behavior customer ≡
      with acquaintances integer n, mail queue address u
      interface is integer k

      send (n * k) to u
   endbehavior customer
endactor customer
```

Figure 2. Actor Example: Factorial

because the actor model defines everything as an actor. The factorial actor then sends itself a message containing the value **n** minus one, and the address of the customer actor. Thus, the factorial actor executes recursively by sending itself messages until the value of the integer reaches zero, all the while creating customer actors that each know a unique value of an integer and an address of an actor.

Upon receipt of a message containing the value **k**, the customer actor multiplies the value by the value of the acquaintance **n** and sends the result to actor **u**. For example, the last customer actor created by the factorial actor is the first to receive a message, and it contains the value one. The customer actor multiplies the value by the value of **n** and sends the result to **u**. This occurs repeatedly, until finally the last customer actor receives a message. It performs the multiplication and

sends the result back to its acquaintance, **u**, which is the address of the original actor that requested the factorial computation.

1.2 Design Goals

The design goals for a new actor garbage collection algorithm are now discussed [KAFU88]. The garbage collector should be:

1. Real-time. To be real-time means that the creation of objects, the access to objects, and the reclamation of objects are all bounded by a constant [BAKE78b, LIEB83]. In addition, the garbage collector must be real-time in the sense that it must not interfere with the execution and operation of the real-time application.
2. Incremental. An incremental garbage collector reclaims a small amount of garbage on every invocation. The invocations of the collector are synchronized with object creation so that the garbage is collected before memory is exhausted. Non-incremental collectors halt the application before they reclaim storage. Halting real-time applications for long periods of time to reclaim all of memory is obviously undesirable.
3. Concurrent. It must execute concurrently (logically or physically) with the real-time application.
4. Automatic. The application does not invoke the garbage collector explicitly; the collector executes automatically and independently.
5. Distributed. The real-time system is distributed across many processors. The garbage collector must understand distributed acquaintances and cooperate with the local garbage collection efforts that is occurring on other processors. For example, it must acknowledge receipt of messages indicating state changes that result in garbage, and it must generate state change messages for other processors.

1.3 Justification for a New Garbage Collector

Traditional, distributed and real-time garbage collectors all have characteristics which make them unsuitable for the actor garbage collection problem. The characteristics of the actor garbage collection problem that make other garbage collection algorithms unusable are listed here.

First, the objects to be collected are computable objects. This is in contrast with nearly all other garbage collectors that only operate on passive data. Some exceptions exist and are reviewed in Chapter 2. However, even these exceptions primarily treat the objects as passive data. The computable property plays an important role in garbage collecting actors, because it is shown that traditional methods of garbage collection (which neglect the computable property) incorrectly identify some objects as non-garbage, when in fact they are garbage.

Second, the methods used to access actors are radically different than the access methods of data objects. Actors themselves do not have direct memory access to other actors; they always use the functions **send**, **new** or **become** to name actors of interest. Thus, access to actors is always handled exclusively by the kernel whenever the functions are invoked, even in the absence of a garbage collector. However, data objects are accessed directly by the application through hardware fetch and store instructions. The kernel is not involved in the access of data objects, except possibly for virtual memory paging or when garbage collectors are introduced.

Third, the real-time system places unique constraints on the garbage collector. These constraints are more demanding than in other systems. For example, the method of stop-and-copy garbage collection simply does not work in a real-time system where hard deadlines must be met. There are several reasons why this is so. One reason is predictability. Predictable response time is an important characteristic of real-time systems. Some collectors, like stop-and-copy garbage collectors, interrupt application processing unpredictably. A second reason is that the application misses

crucial deadlines if the garbage collector suspends the application at the wrong moment, or if the application is suspended for relatively long periods of time.

Finally, the garbage collector must operate in a distributed environment. The objects to be collected are actors that are executing on processors connected together in a local area network. There are several reasons why the garbage collector must be implemented as a collection of cooperating processes, each on a single processor. One reason is that each processor operates in its own memory space, and none of the memory spaces are shared. Therefore, the garbage collectors are responsible for reclaiming the resources for the machines on which they are executing. In addition, the garbage collectors cooperate with each other to identify garbage in the entire system. Another reason is performance. It is neither easy nor efficient to implement a centralized garbage collector for a distributed system. The processor executing the garbage collection algorithm must devote much of its resources to this task. The effective communication bandwidth for the application is reduced because the garbage collector must interrogate each machine to identify actors and acquaintances. Most important, however, is the fact that the machine running the garbage collection algorithm becomes a single point of failure. If the processor or any of its communication hardware ceases to function, then the entire system fails.

1.4 Thesis Organization

The remaining chapters are organized as follows:

- Chapter 2 describes previous work in the study of garbage collection. Garbage collection algorithms are classified and presented.
- Chapter 3 presents three algorithms that garbage collect actors.

- Chapter 4 shows that the third algorithm is correct. An analysis of the three garbage collection algorithms is presented. These algorithms are compared with the previous work in garbage collection. The chapter concludes with an implementation strategy for an actor garbage collection algorithm.
- Chapter 5 presents conclusions and discusses areas of future study.
- The Appendix contains a glossary of terms used in this thesis.

2.0 Garbage Collectors

The earliest papers on garbage collection date back to 1960 [COLL60, MCCA60]. This chapter reviews the garbage collection algorithms developed since that date. The garbage collection algorithms are classified into the following categories:

Reference Counting

Mark and Sweep

Semi-space Copying

Generational

On-the-fly

Virtual Memory

Distributed

Computational Objects

A more comprehensive survey of garbage collection algorithms prior to 1981 can be found in [COHE81]. A more recent survey of mark and sweep and semi-space algorithms is by Lang and Dupont [LANG87].

The following terms are used throughout this chapter.

Mutator. The application that creates objects and modifies the relationships between objects.

Garbage. Objects that are not accessible by the mutator.

Live data. Objects that are accessible by the mutator.

Collector. The garbage collector that returns garbage objects to the free pool of storage.

2.1 Reference Counting

One of the earliest garbage collection schemes [COLL60] is reference counting. The storage for a reference counter is allocated when the object is allocated. The storage for the reference counter is not necessarily coincident with the object itself. When the object is allocated, the reference counter is initialized to one. Every time a pointer is assigned to the object, the object's reference counter is incremented by one; each pointer deassignment decrements the counter. When the counter reaches zero, the object is garbage and is deallocated.

Reference counting has several problems. Circular structures are a problem because the reference counter of the objects in the circular structure never reach zero; therefore, they are never garbage collected. Deutsch and Bobrow [DEUT76] present a solution to this problem: reference counting is used as a means of reclaiming non-circular structures, and a compacting mark and sweep collector (discussed below) is used periodically to reclaim and compact memory. Another solution is proposed by Christopher [CHRI84] in which he describes a reference counting garbage collector that reclaims circular structures, albeit inefficiently.

A second problem with reference counting is the storage requirement for the reference counter. The counter must be at least $\log_2(n)$ bits large to record a reference from every object in the system,

where **n** is the number of objects in the system. In some cases, the storage for the counter may more than double the size of the object.

A variation on reference counting proposed by Wise and Friedman [WISE77] offers some relief. The counter size is restricted to just one bit. The theory is that the interesting objects are those that have exactly one reference, because they are the objects that become free the soonest. In addition, a fixed-size cache of objects is introduced. The cache contains objects whose reference counts should be two, but are not because there is just one bit. As objects change from having one reference to two, they are placed into the cache; if the cache is full, an object is bumped out according to some replacement algorithm and its reference count is reset to zero. When an object is dereferenced, one of three conditions is true:

1. The object is in the cache. The object's true reference count is two (because it is in the cache). The object is removed from the cache, thereby decrementing the object's reference count.
2. The object is not in the cache and its reference count is zero. This means that the object at one time was a member of the cache, but was bumped out to make room for another object. Its true reference count is unknown, so the object cannot be garbage collected.
3. The object is not in the cache and its reference count is one. This is an interesting object because its reference count is going to zero; therefore, the object is garbage. It is reclaimed.

Objects outside of the cache that have zero reference counts are reclaimed by a mark and sweep garbage collector that runs in the background. The mark and sweep collector also reclaims circular structures and computes the new value of the reference bit of those objects that remain.

A third problem with reference counting garbage collectors is that they are not real-time. Although the operations "increment" and "decrement" are well-defined and can be implemented in a constant number of steps, the problem occurs when the counter reaches zero. The unreferenced object must be deallocated which in turn may cause other reference counters to be decremented (some of which may become zero). Lazy garbage collection [GLAS87] is an attempt to solve this problem, but it

is designed for uniformly-sized objects. The decrement function is replaced with a procedure that, instead of decreasing the reference count of the object, places the object on a stack. When the mutator issues an object allocation request, the system pops an object from the stack and decrements its reference count. If the reference count is non-zero, the object is not yet garbage, so it is not used. The allocator keeps popping objects off the stack and decrementing their reference counts until an object with a reference count of zero is found. Once an object with a reference count of zero is found, it is used to satisfy the allocation request because the object is garbage. First its references to other objects must be accounted for, so the objects it references are pushed onto the stack. This scheme helps bound the overhead of the garbage collector by deferring the reclaiming of objects until they are actually needed.

2.2 Mark and Sweep

The other early garbage collection algorithms in use in the 1960s are the mark and sweep variety. McCarthy [MCCA60] was the first to suggest this type of garbage collection. Each variation of mark and sweep garbage collection executes the same basic algorithm. First, the mutator is halted. Second, all of the objects that are reachable by the mutator are marked. The marking algorithm traces the set of reachable objects by starting with the addresses of objects in the mutator's registers, and sets the mark bit of each. The marking algorithm continues by selecting a marked object and setting the marked bit of the objects that it references. The marking algorithm terminates when no more objects are marked. Third, the entire memory space is swept. All unmarked objects are garbage and their space is returned to the free pool. Finally, the mutator is resumed.

One of the fundamental problems of executing a garbage collector is that the collector must execute using only a fixed number of storage cells, because obviously there is no memory available when the garbage collector is invoked. The Schorr-Waite marking algorithm [SCHO67] is the first to

suggest a solution to the problem. As the garbage collector traces objects from parent to child, the pointer field is reversed, so the algorithm can find its way back to the root. If the root has more than one child, some additional bits are required so the algorithm knows how to restore the pointer field.

There are deficiencies of the basic mark and sweep algorithms. First, virtual memory becomes fragmented. Compacting mark and sweep collectors [HADD67, BOBR68, HANS69, WEGB72, FISH74, THOR76, HANS77, TERA78, FITC78, MORR78, JONK79, MORR79] solve this problem by relocating the live data (or the garbage objects) to one end of memory or the other. Most mark and sweep collectors are of the compacting variety.

Second, the basic mark and sweep algorithm is not incremental. Bobrow [BOBR68] suggests executing the mark and sweep collector in a separate process that timeshares the processor, but he assumes that the mutator does not consume storage so fast that the collector does not finish. Another solution to this problem is to use an on-the-fly mark and sweep collector; this type of collector is discussed later in this chapter.

Third, these algorithms need some sort of data marking to distinguish addresses of objects from other data values. Consider that when a mark and sweep algorithm starts, it must know the root nodes of the objects to be copied. It determines this by considering the registers of the process as addresses of reachable objects. This is incorrect if the value of a register is not an address but some other data value. Some solutions involve tagging [THOR76, FRIE76, HANS77], where one or more bits of a data value is reserved and used to indicate if the value is an address or some other data type.

Finally, mark and sweep collectors require one pass through the live data (the marking phase) and another pass through all of memory (the sweep phase). Some compacting mark and sweep collectors require a third pass through memory to compact the live data to one end of memory; other

compacting mark and sweep collectors combine the compacting phase with the sweep phase. But, as memory sizes increase, the sweep and compacting phases take longer to execute.

2.3 Semi-space Copying

Semi-space copying algorithms are an attempt to efficiently solve two problems of mark and sweep algorithms: fragmentation of memory and execution time of the collector. The copying collector by Fenichel and Yochelson [FENI69] divides memory into two regions: **fromspace** and **tospace**. Both spaces are initially empty. All objects are created in tospace. When tospace becomes full, a **flip** is performed: the mutator is halted and the objects which are reachable by the mutator are copied from fromspace to tospace. This compacts the live data into a contiguous region of memory. The roles of fromspace and tospace are reversed, and the mutator is resumed.

Two important characteristics of this collector are:

1. It executes in time proportional to the number of live objects, not proportional to the size of memory, because the collector relocates the live data as it traces them. Thus, it never examines the garbage objects.
2. It compacts objects into a smaller amount of memory. This results in better execution performance on virtual memory systems, because of the principle of locality of reference.

Concurrent, incremental copying collectors are the next major improvement in garbage collection. They allow the mutator to execute concurrently with the garbage collector. The first implementation is the collector by Baker [BAKE78b]. Not only is his collector concurrent and incremental, it is also real-time: the collector's operations are bounded by a constant. In this type of collection

algorithm, memory is again divided into two regions, fromspace and tospace. Tospace is further divided into three areas:

1. New. This area is where new objects are created.
2. Scanned. This area contains objects that have been scavenged by the collector (scavenging is discussed below).
3. Unscanned. This area contains objects that have been relocated from fromspace, but have not yet been scavenged.

Pointers define the boundaries of these areas, as shown in Figure 3.

When the mutator creates objects in tospace, the garbage collector scavenges a constant number of objects in the unscanned area. An object is scavenged when the objects it points to are relocated from fromspace to the unscanned area of tospace. Scavenging leaves behind a forwarding address at the relocated object's old location so the garbage collector does not relocate objects more than once. Scavenging updates the scavenged object's pointers so they point to the new object's address in tospace. Finally, scavenging increments the scavenged pointer in tospace as objects are relocated from fromspace. When the scavenged pointer catches up to the unscanned pointer, all reachable objects in fromspace have been moved to tospace; the objects that remain behind are garbage.

All objects are allocated in the new area of tospace after which the new pointer is decremented. Eventually the mutator is unable to allocate objects in tospace because the new pointer meets up with the unscanned pointer. When this happens, a flip is performed:

1. Tospace becomes fromspace and fromspace becomes tospace.
2. The objects pointed at by the mutator's registers are copied from fromspace into the unscanned area of the new tospace; the mutator's registers are updated to point to the new addresses.
3. The allocation is performed.

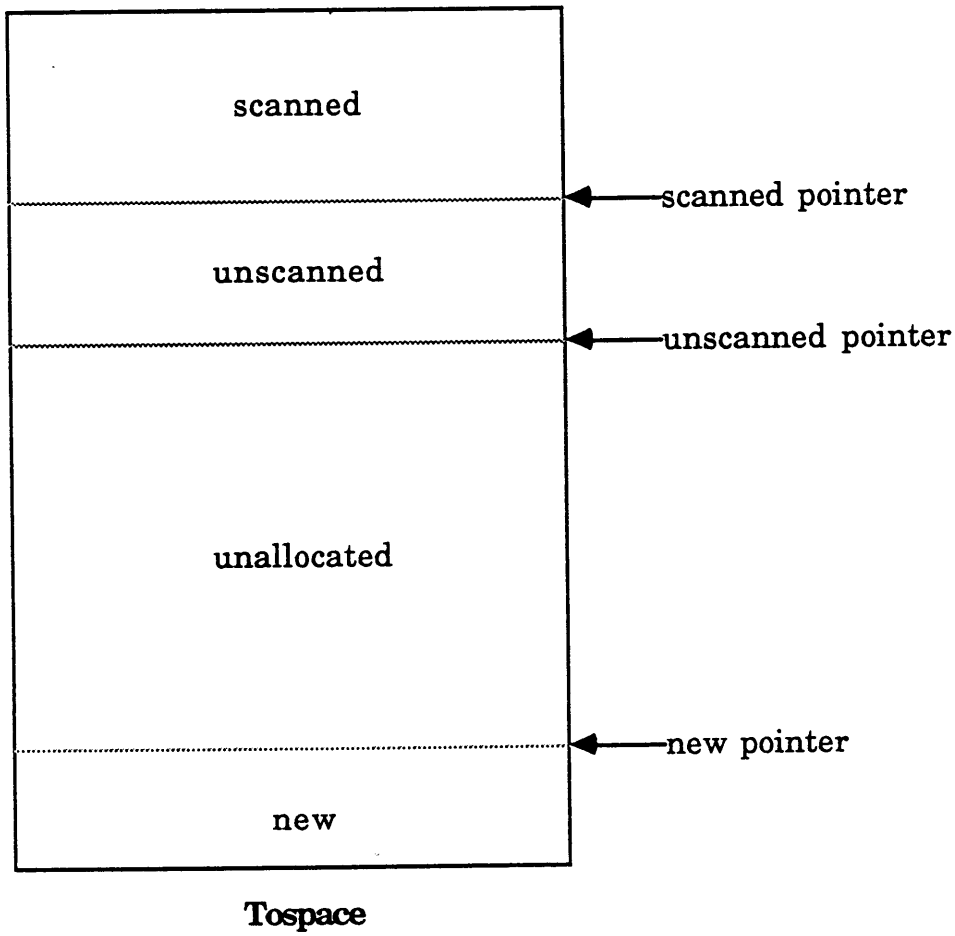

Figure 3. Layout of Tospace

Because the objects pointed to by registers are relocated before the mutator is resumed and because the allocation procedure always scavenges a constant number of objects, the garbage collector is guaranteed to finish scavenging before tospace becomes full.

Baker's collector is designed for the LISP environment where each object is a LISP atom of exactly two cells. The LISP function **cons** allocates new objects, so it is here that the incremental garbage collection is implemented. The LISP functions **car**, **cdr** and **atom** verify that the addresses they return to the mutator are in tospace; if not, they move the fromspace object to the end of the unscanned area of tospace, leave a forwarding address behind in the fromspace location, update the mutator's address of the object so that it now refers to the tospace location, and then return the value the mutator requested.

One problem with Baker's algorithm is that it is slow on stock hardware because of the overhead on every fetch (e.g., **car**, **cdr** and **atom**) that forces all addresses to be in tospace. Brooks [BROO84] offers modifications to Baker's algorithm that reduces this overhead considerably, at the expense of increased storage requirements and an extra indirection pointer for each object.

A problem of all copying semi-space algorithms is that they halve the amount of memory available for the mutator. This is solved somewhat by generational collectors where, instead of two spaces, there are many spaces. Another drawback of copying semi-space algorithms is that they are difficult to implement on a distributed network of processors, because the collectors on each node must synchronize so that the flips occur simultaneously.

Other work on copying garbage collectors includes Lang and Dupont's [LANG87] collector, which is a combination of a mark and sweep and a copying collector. Their strategy for using two collectors is to overcome the shortcomings of each: mark and sweep's necessity for sweeping all of memory and copying collector's dividing available memory in half. Nilsen [NILS88] describes an incremental copying collector that operates on variable-sized objects. The algorithm is also real-time because each allocation of an object is bounded by a constant. Finally, Courts [COUR88]

discusses performance characteristics of copying algorithms and suggests ways to improve the locality of reference property.

2.4 Generational

An extension of Baker's copying garbage collector, the generational collector by Lieberman and Hewitt [LIEB83] is efficient for a number of reasons. It offers the same characteristics as Baker's: it is real-time and it improves the locality of reference property. In addition, it is more efficient because it concentrates the garbage collection effort to where it is needed most. This method assumes that most objects are short-lived, therefore the generational garbage collector concentrates on objects that are the newest, because they are most likely to be garbage.

The algorithm accomplishes these improvements by dividing memory into many semi-spaces, all of uniform size and small enough to be managed easily by the processor. These spaces are called generations. Each generation is associated with a version number that indicates the number of times the particular generation has been reclaimed. Objects are created in the newest generation. When it fills up, another generation is allocated. The garbage collector spends the majority of time garbage collecting newer generations, and less time collecting older generations. The rationale being that the longer an object lives, the longer its expected lifetime.

The garbage collector reclaims generations by three operations: condemning, evacuation and scavenging. Condemning a generation indicates the garbage collector's intent to recycle the generation. A new generation is created with the same generation number as the old, but with a version number one higher. Evacuation is when the live objects from the condemned generation are relocated to the new version of the generation. Scavenging is the process of scanning all other gener-

ations to update the pointers to the relocated objects. After these steps, the condemned generation is totally reclaimed.

2.5 *On-the-fly*

On-the-fly algorithms are designed for multiprocessor systems. A justification for developing multiprocessor algorithms is described by Dijkstra et al. [DIJK78]. They claim that as operating system housekeeping functions become more complex and take longer to execute, it becomes worthwhile to consider offloading the housekeeping to an attached processor of a multiprocessor system. Knuth [KNUT68] credits Minsky as the first to consider executing the garbage collector in parallel with the mutator. On-the-fly algorithms assume that the multiprocessor system consists of two or more processors; the garbage collector executes on one, and the mutator executes on those that remain. The processors all share the same physical memory.

On-the-fly algorithms are extended versions of the mark and sweep variety [STEE75, WADL76, DIJK78, HICK84, BENA84, ELLI88]. It does not make sense to design an on-the-fly collector using reference counting, because most of the overhead (incrementing and decrementing a counter each time an object is referenced and dereferenced) cannot be efficiently moved to the attached processor. Semi-space copying collectors are not chosen because they restrict the mutator to half of the available memory. Thus, compacting mark and sweep collectors are almost always chosen for transformation into on-the-fly algorithms.

Some of the hardest problems with on-the-fly algorithms are defining and implementing them correctly, and proving their correctness. One of the difficulties is that it is desirable to design the collector to be as non-intrusive as possible, so that synchronization and other effects on the mutator are minimized. The effort by Dijkstra et al. is an exercise in designing an algorithm and proving

that it is correct. The experience of the authors "has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible" [DIJK78].

For each algorithm on multiprocessor garbage collection, there is at least one report that analyzes and proves its correctness. Wadler [WADL76] analyzes a real-time, multiprocessing, mark and sweep collector and compares its performance with that of uniprocessor collectors. Hickey and Cohen [HICK84] analyze the operation of Dijkstra's algorithm and tri-color garbage collection algorithms in general. They conclude that there are three states the system can be in: stable, oscillating, and critical. They describe the system state using conditional differential equations.

2.6 *Virtual Memory*

The garbage collector developed at the Digital Equipment Corporation Systems Research Center by Ellis, Li and Appel [ELLI88] and reported by Appel [APPE88a, APPE88b] is a real-time, multiprocessor, incremental garbage collector that takes advantage of virtual memory paging characteristics. The authors note the main problem with Baker's [BAKE78b] real-time algorithm: the mutator is slowed down considerably because the collector maintains the invariant that each address the mutator sees is a tospace address. Because each fetch must check the invariant, processor cycles are wasted when the address is in tospace. In fact, it is reported that early implementations of the algorithm were turned off by users [ELLI88]. Brooks's improvement to Baker's algorithm [BROO84] costs an extra word of memory per object plus an extra indirection pointer. The virtual memory algorithm does not suffer from these performance or storage penalties.

Instead of checking every reference, the virtual memory algorithm only updates the addresses it knows the mutator references. Like Baker's algorithm, memory is divided into fromspace and tospace; tospace is further divided into new, scanned and unscanned areas. The access protection

of the fromspace pages and the unscanned area's pages is set so the mutator has no access; the scanned and new areas of tospace are protected to allow the mutator read and write access. Object creation occurs in the new area. There are two execution threads of the garbage collector: a page fault handler, which is invoked when a page fault trap is generated, and a thread that executes concurrently on an attached processor.

When the mutator references an object in fromspace or in the unscanned area of tospace, a page fault is generated. If the faulted address is a fromspace address, the fault handler relocates the object to the end of the unscanned area of tospace and resumes the mutator at the faulting instruction. (Note that this generates another fault because the unscanned area is also protected against access by the mutator.) For faults generated in the unscanned area, the page fault handler performs the following steps:

1. Scavenges all of the objects on the faulted page, relocating referenced objects in fromspace to the unscanned area of tospace.
2. Leaves a forwarding pointer behind in fromspace for all the moved objects.
3. Unprotects the faulted page.

The mutator is resumed at the faulting instruction which now succeeds because the reference is to a scanned page in tospace that contains only tospace addresses. Note that there is no clean division of the scanned and unscanned areas, as in Baker's [BAKE78b] algorithm, because the page fault handler scans and unprotects pages anywhere in the unscanned area.

While the mutator executes, the other thread of the garbage collector is executing concurrently on another processor that has access to the same shared memory. Its responsibility is to scavenge the unscanned pages to move referenced objects that are in fromspace to tospace. When there is no more unscanned area to scavenge, the thread suspends itself until a flip occurs.

When tospace becomes filled to within a threshold value, a flip is performed. First, the mutator is halted and the remaining unscanned pages are scavenged. Then the roles of fromspace and tospace are reversed, the object addresses in the mutator's registers are relocated and the collector on the attached processor is resumed. Last, the mutator is resumed. The threshold is set to a value less than 100% because the garbage collector must leave enough memory to finish scavenging unscanned pages (if any exist) when the flip is initiated.

The garbage collector is able to manipulate the objects on the protected pages even though the mutator can not, because the page protections are set differently for the different execution modes of the processor. The mutator executes in user mode but the page fault handler and the attached processor garbage collector execute in kernel mode.

2.7 Distributed

Garbage collectors that operate across a distributed network of processors build upon the styles of garbage collectors discussed earlier in this chapter. Some are distributed reference counting collectors. Eckart and LeBlanc [ECKA87] contend that reference counting is the best kind of distributed garbage collector, because objects can be collected independently without requiring synchronization with other processors. Eckart and LeBlanc describe an improvement to reference counting that they call reference marking. Their algorithm is motivated by the needs of functional programming languages, and works by an application of a coloring algorithm to the arguments of invoked functions. Lermen and Maurer [LERM86] describe a protocol for distributed reference counting that requires synchronization only when a reference to an object is deleted.

A distributed copying collector is proposed by Rudalics [RUDA86] that reclaims objects more slowly than regular collectors and requires a synchronized flip of all processors when they are in-

volved in global garbage collection. However, local garbage collection (which ignores references to objects on other nodes) is asynchronous. Rudalics claims that distributed copying garbage collectors cannot be made into real-time collectors, because classical real-time collectors (like Baker's [BAKE78b]) synchronize object creation with garbage collection. Because of this and the fact that some processors create very few objects, the global garbage collection is slowed to match that of the processor that has the slowest rate of object creation.

2.8 Computational Objects

The last type of garbage collection algorithms are those that operate on computational objects. A computational object is one that is associated with both data and code and has scheduling attributes. Examples of computational objects are processes, lightweight processes, and actors.

Baker and Hewitt [BAKE77] report on a garbage collector for functional languages that is incremental. Each expression needed by the functional program is *eagerly* evaluated; that is, the computation to determine its value is begun immediately. This in turn generates more expressions, which are also eagerly evaluated. Each expression is independent and may be evaluated by a process concurrently with the others. To coordinate the receipt of values, a "future" is created for each expression. When the expression evaluation terminates, the future is filled in with the value, and all of the other expressions waiting for the value read it and resume their computations.

During the course of execution, the values of some expressions are no longer needed. The algorithm described by Baker and Hewitt [BAKE77] reclaims the processes by an incremental garbage collector. The collector uses three colors—white, gray and black—to trace the graph of objects (processes) connected to the root process. Initially all processes are white with the exception of the root process, which is colored gray. As long as gray processes exist, the algorithm chooses one

and colors it black and colors its children gray in one indivisible step. When there are no more gray processes, the algorithm is finished. White processes are garbage and are terminated; black processes are allowed to continue. The garbage collection algorithm is analogous to a non-relocating mark and sweep collector of data objects.

Baker [BAKE78a], Halstead [HALS78] and Agha [AGHA86] each made contributions that describe various aspects of actor computation. Baker focuses on actors for real-time systems; his contribution to the garbage collection problem is a restatement of the work by Baker and Hewitt [BAKE77]. Halstead describes distributed actor computations. Halstead's garbage collector is based upon his concept of an actor reference tree. An actor's reference tree is a set of processors and connections between processors such that each processor has a reference to the actor. To perform garbage collection, the reference tree of the actor is reduced until it contains a single processor. Then a local garbage collector is used to determine if the actor is garbage. This scheme is very much like reference counting, and it shares one of the problems of reference counting collectors: it cannot handle circular structures.

Agha's dissertation presents actors as "a model of concurrent computation in distributed systems" [AGHA86]. His contribution to garbage collection is twofold. First, he makes the observation that an actor is garbage if it is not processing any messages, if it is not the target of some undelivered message, and if it is not an acquaintance of any other actor (except itself). Second, Agha notes that forwarding actors may be garbage collected. A forwarding actor sends all messages it receives to exactly one other actor. The forwarding actor serves only to slow the communications between the other two actors, so it can be garbage collected resulting in improved communication and processor efficiency.

The last computational object algorithm is a distributed garbage collection algorithm by Mehta and Christopher [MEHT88]. Their collector is based on mark and sweep techniques and is motivated by the actor computation model [CHRI88]. The collector starts by marking all actors that are computable. A computable actor is one that is processing a message, or one that is the target of

an as-yet unreceived message. Thus, computable actors form the root objects that the collector considers to be analogous to the mutator registers in previous garbage collection algorithms. The collector continues by propagating the marking to the computable actor's acquaintances, and to the acquaintances of the acquaintances, and so on until no more markings are made. When the marking phase terminates, all unmarked actors are reclaimed as garbage.

2.9 Summary

All of the garbage collectors presented in this chapter collect objects which are not connected to root objects; objects which are connected are not garbage. Connected objects are discovered by starting at the root objects and following the pointers to child objects until no more objects are discovered. (The exception to this is reference counting collectors, which implicitly keep track of the connected information with a reference count.)

3.0 Garbage Collection Algorithms

This chapter describes three garbage collection algorithms that are designed to reclaim actors. The organization of this chapter is as follows. First, the assumptions that are made by the garbage collectors are given. Second, the garbage collection algorithms are described. The first algorithm examines all but "isolated" objects. It is a transformation of a traditional "passive-data" garbage collector into an "active-data" collector: one that collects computable objects. The second algorithm uses a state table but examines all objects (potentially many times). The third algorithm examines only "active" and "reachable" objects. It is an efficient implementation of Algorithm Two. For simplicity, the algorithms are presented in a high level of detail and do not show the distributed collectors working in concert; a distributed solution to the actor garbage collection problem is outside the scope of this thesis and is deferred for future work.

3.1 Assumptions

All three of the garbage collection algorithms presented in this chapter make assumptions about the support that is provided by other parts of the real-time system.

One assumption is that the compiler for the actor language generates code to inform the garbage collector about the static acquaintances of actors. The static acquaintances are those specified in the actual code of the actor. These acquaintances always remain acquaintances of the actor; it is not possible for the actor to remove them from its acquaintance list.

A second assumption made by the garbage collection algorithms is that the actor language compiler and run-time system—specifically, the message-passing system—informs the garbage collector whenever the mail queue address of an actor is passed in a message. In addition, the garbage collector must be informed about the address-passing at the beginning of the message transmission, because the message may take an arbitrary long time to be delivered to the recipient actor. The recipient actor must be treated as if it receives the message (and knows the new acquaintance) the instant the message is sent. Further, the recipient actor is considered to know the new acquaintance even if the message remains in the recipient's mail queue indefinitely. Chapter 4 presents implementation solutions of these assumptions.

3.2 Terms and Definitions

The following terms and definitions are used throughout this chapter.

Blocked. An actor is blocked if all of the following conditions hold:

1. The actor's mail queue is empty.
2. The actor has exactly one behavior.
3. The behavior is awaiting the arrival of an as-yet unsent message in the actor's mail queue.

Computable. An actor is computable if it has at least one executing behavior or if it is the recipient of an as-yet unreceived message. Computable and blocked are mutually exclusive.

Actor acquaintance list. The acquaintance list of an actor is formed by a union of all of the actor's behaviors's acquaintance lists. The unioned acquaintance list is combined with the mail queue addresses stored in the actor's mail queue of unread messages to form the actor's acquaintance list. The actor acquaintance list identifies all of the actors that can be addressed by the given actor; that is, all of the actors to which the specified actor can send a message. An actor always has itself as an acquaintance.

Actor inverse acquaintance list. The inverse acquaintance list identifies all of the actors that can address the given actor; that is, all of the actors that can send a message to the specified actor. Note that the inverse acquaintance list is never empty, because an actor always knows its own address.

Anchor point. This term is used to define an actor that is the real-time application's internal representation of objects by which the application monitors and controls the real world. It is analogous to the objects pointed to by the mutator's registers in traditional garbage collection schemes. Examples of physical objects that have this classification are sensors, actuators, and clocks. Two characteristics of anchor point actors are:

1. Anchor point actors are never blocked; that is, the garbage collector assumes that they are always computable. This assumption is made because the physical world is always changing, therefore the anchor point actor must always reflect the changes (by sending messages and changing its internal state) to correctly represent the external environment to the application. This assumption implies the second.
2. Anchor point actors are never garbage collected because—in traditional garbage collection terms—the anchor point actors serve as roots of the network of actors.

Isolated. An actor is isolated if its inverse acquaintance list contains no addresses other than its own. If an isolated actor is blocked, then it remains blocked forever, because no other actor can send it a message to cause it to become computable. Isolated actors are garbage.

Reachable. An actor X is reachable if an actor with the anchor point characteristic can send a message to X or receive a message from X, either directly (no intervening actors) or indirectly

(with some actors in between the two). An actor that is not reachable is **unreachable**. Unreachable actors are garbage.

Owner. When observing an acquaintance between two actors, the actor that is the originator, or holder, of the acquaintance is the owner of the acquaintance.

Member. The member of an acquaintance is the actor that is the target of the acquaintance.

3.3 Algorithm One

The first algorithm employs the use of three markings to classify actors during the garbage collection phase. The colors of the markings are white, gray and black. Their meanings are given below.

- White. This is the initial coloring of all actors. An actor that remains white after the algorithm terminates means that the actor is isolated (and therefore is garbage).
- Gray. An actor with this color after the algorithm terminates is unreachable (and therefore is garbage).
- Black. Actors colored black by the algorithm are not isolated and they are reachable, therefore they are not garbage.

3.3.1 Operation

The algorithm consists of the coloring rules presented in Figure 4. A coloring rule is applied only if it results in the actor's color becoming "darker;" rules which would not change the color of the actor, or those that would "lighten" the color are not applied. Black is darker than gray which is darker than white.

Initialization Step. All actors are colored white, with the exception of actors with the anchor point characteristic: they are colored black.

Marking Step. Repeat the following operations until no more markings are made:

1. Color black all acquaintances of black actors.
2. Color black all inverse acquaintances of black actors if the inverse acquaintance is not blocked.
3. Color gray all inverse acquaintances of black actors if the inverse acquaintance is blocked.
4. Color gray all acquaintances of gray actors.
5. Color black all inverse acquaintances of gray actors if the inverse acquaintance is not blocked.
6. Color gray all inverse acquaintances of gray actors if the inverse acquaintance is blocked.

Termination Step. Actors that are colored black are not garbage. Gray and white actors are garbage and are reclaimed.

Figure 4. Algorithm One

3.3.2 Example

Table 1 shows a key that explains the actor configuration used in the next figure. Briefly, actors with the anchor point characteristic are shown as squares; other actors are represented as circles, and acquaintances are the directed arcs between the actors. Each actor is identified with a unique uppercase letter. The actors drawn with shaded borders are active; actors drawn with solid borders are blocked.

A configuration of actors is displayed in Figure 5 to illustrate the operation of Algorithm One. Figure 5a is the initial state of the actor configuration after the initialization step of the garbage collector executes.

Figure 5b presents a snapshot of the system when the garbage collector is in the marking step phase, but before all of the markings are made. All of the anchor point actors are colored black,

Table 1. Key to Actor Configuration Symbols

Symbol	Feature	Meaning
	Square	Anchor point actor
	Circle	Non-anchor point actor
	Shaded border	Active actor
	Black border	Blocked actor
	White interior	Isolated actor
	Gray interior	Unreachable actor
	Black interior	Connected, reachable actor

indicating that they are not garbage. Actors **G** and **H** are also colored black after applying the first coloring rule: they are an acquaintance and indirect acquaintance, respectively, of an anchor point actor. Actors **D** and **E** are colored gray because of rules six and three, respectively. Actor **C** is colored black because of rule five; the algorithm has determined that actor **C** is not garbage because it is active and able to send a message to an anchor point.

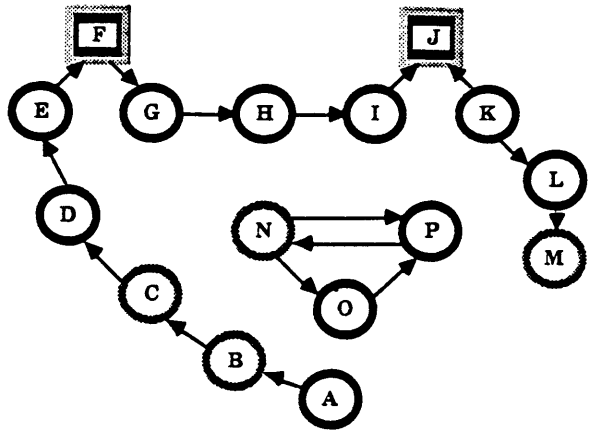

Figure 5a

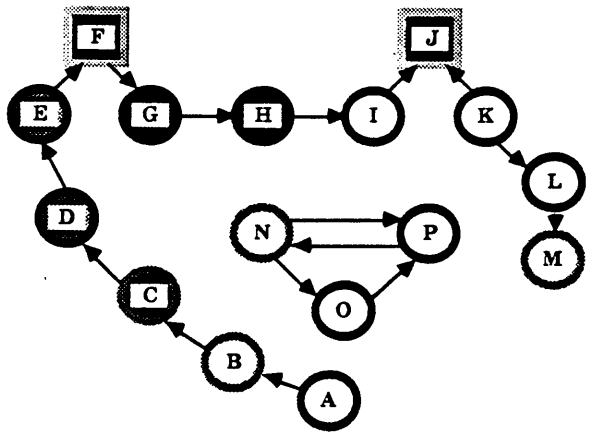

Figure 5b

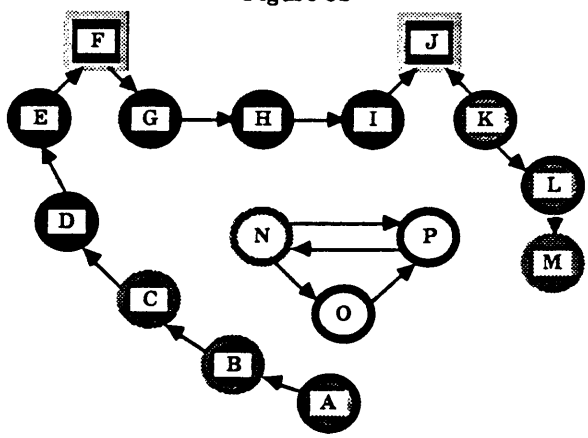

Figure 5c

Figure 5. Three Snapshots of Algorithm One

In Figure 5c, the garbage collector is finished with the marking phase; all actors are classified as garbage or non-garbage. All black nodes are not garbage: **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I** and **J**. Actors **A**, **K**, **L** and **M** remain gray, because they are unreachable, and are therefore garbage. Even though actor **K** is acquainted with anchor point actor **J**, it can never send a message to it, or any other actor, and thus it can neither affect the physical world nor be affected by the physical world. Actor **M** is also unreachable. In addition, the actor is computable, therefore it is not only wasting memory resources, but processor resources as well.

The last three actors—**N**, **O** and **P**—are garbage because they are colored white. This means the actors are isolated, and can neither affect nor be affected by the outside world.

The algorithm shades more actors than is necessary to determine the garbage present in the system. All of the gray actors are inspected by the algorithm, but are still garbage.

3.4 Actor State Table Algorithm

The last two algorithms presented in this chapter utilize a state table to propagate the reachable and computable properties. The table provides a more precise definition of the capabilities of each actor, and shows how the capabilities are passed from one actor to the next.

Table 2 contains the marking rules used by Algorithm Two and Algorithm Three. Each row of the table is numbered for identification purposes. The entries in the table are:

0—This bit corresponds to the Boolean value False, meaning the bit is disabled.

1—This bit corresponds to the Boolean value True, meaning the bit is enabled.

Table 2. Actor State Change Table

	Owner		Member	
	Reachable	Computable	Reachable	Computable
1.	0	0	0	0
2.	0	0	0	1
3.	0	0	1	x
4.	0	1	0	x
5.	x	1	1	x
6.	1	x	x	x

x—This is the "don't care" value. When the table is scanned during the marking algorithm, this bit is ignored. After a matching rule is located, all "don't care" bits in the rule are enabled in the owner and member actor's state information.

The rows have the following meanings.

1. The first row simply says that if the owner of an acquaintance is neither reachable nor computable, the member of the acquaintance can not be made either reachable or computable.
2. The second row says that when the owner of an acquaintance is neither reachable nor computable and the member of the acquaintance is computable, no state bits can be changed.
3. The third row indicates that if the owner of an acquaintance is neither reachable nor computable and the member is reachable, then the member is always computable. This is because the member may receive a message from an anchor point actor (by virtue of being reachable), therefore even if the member is not computable now, it may become computable. Thus the garbage collector should not reclaim the member actor.
4. The fourth row shows that if the owner of an acquaintance is only computable, then the member is also computable. This rule propagates the computable property from an actor to its acquaintances, because a computable actor may send a message to an acquaintance and cause it to become computable.

5. The fifth row states that if the owner of an acquaintance is computable and the member is reachable, then the owner is both reachable and computable, and the member is also reachable and computable. Because the member actor is reachable, that automatically makes it computable (the same as in rule three). Additionally, because the owner actor is computable, that means it may choose to send its mail queue address to the member actor which makes the owner actor reachable.
6. The sixth row states that if the owner of an acquaintance is reachable, then it is also computable, and that the member of the acquaintance is also reachable and computable. Any actor that is an owner of an acquaintance and is also reachable is able to become computable (receive a message from an anchor point actor). The owner actor may send messages to the member actor causing it to become computable. Finally, the member actor is reachable by the anchor point actor by virtue of the fact that the owner actor is reachable.

3.5 Algorithm Two

This algorithm uses two bits to record the state information of the actors in the system. Only two bits are needed because the previous algorithm used three colors, which can be encoded in two bits. One of the bits is assigned to mean that the actor is reachable. The other bit is assigned to mean that the actor is computable.

Table 3 shows a key that explains the actor configurations in the remaining figures throughout this thesis. Briefly, actors with the anchor point characteristic are shown as squares; other actors are represented as circles, and acquaintances are the directed arcs between the actors. Each actor is identified with a unique uppercase letter. The actors drawn with shaded borders are active; actors drawn with solid borders are blocked. The leftmost bit of an actor is its reachable bit; the rightmost, its computable bit.

Table 3. Key to Actor Configuration Symbols

Symbol	Feature	Meaning
	Square	Anchor point actor
	Circle	Non-anchor point actor
	Shaded border	Active actor
	Black border	Blocked actor
00	Leftmost bit	Connected bit
00	Rightmost bit	Reachable bit

3.5.1 Operation

The algorithm uses the bit assignments as shown in Table 2. The algorithm is presented in Figure 6.

Initialization Step. The state bits of each actor are initialized as follows: computable actors' computable bits are enabled; anchor point actors' computable and reachable bits are enabled; all other actors are blocked, and have no state bits enabled. Each acquaintance arc is placed into an internal set of arcs.

Marking Step. Repeat the following operations until the set becomes empty.

1. Remove an acquaintance arc from the set.
2. Compute the state bit assignments of the owner and member actors of the acquaintance arc, by matching the bit patterns of the owner and member state with the patterns in the table.
3. If the state bits of the owner actor are changed by the table lookup, place all of the owner's acquaintance and inverse acquaintance arcs into the internal set of arcs.
4. If the state bits of the member actor are changed by the table lookup, place all of the member's acquaintance and inverse acquaintance arcs into the internal set of arcs.

Termination Step. Reclaim the garbage actors. Garbage actors are those with the reachable bit disabled.

Figure 6. Algorithm Two

3.5.2 Example

The same configuration of actors shown in Figure 5 is used to illustrate the operation of Algorithm Two in Figure 7. Figure 7a shows the system of actors after the initialization step is performed: anchor point actors have the reachable and computable bits enabled, computable actors have the computable bit enabled, and blocked actors have no bits enabled. Each acquaintance arc is identified with a unique number, for discussion purposes.

In Figure 7b, the configuration is presented when the algorithm is in the marking phase, but before all of the markings are made. The algorithm has examined at least the acquaintance arcs numbered 3, 4, 6, 7, 14 and 16, because actors **D**, **E**, **G**, **H**, **O** and **P** all have their computable bits enabled.

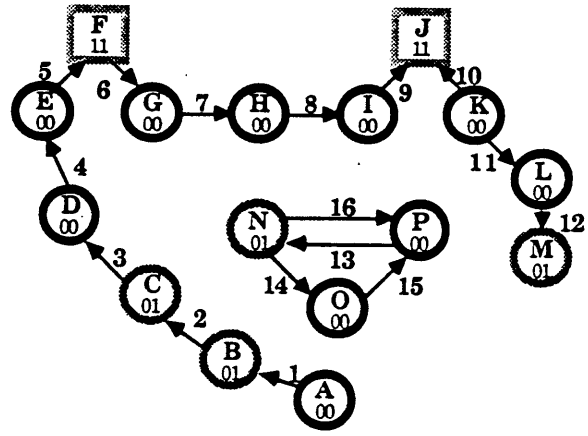

Figure 7a

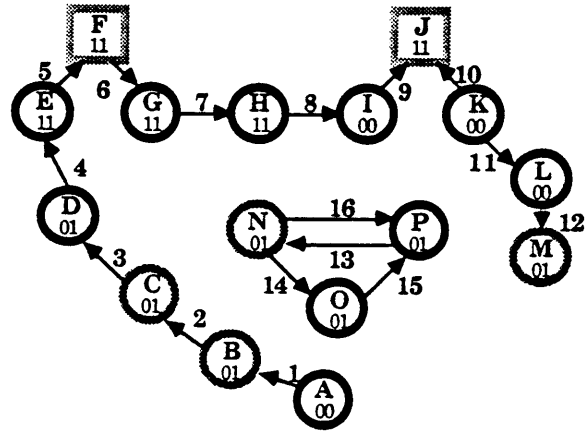

Figure 7b

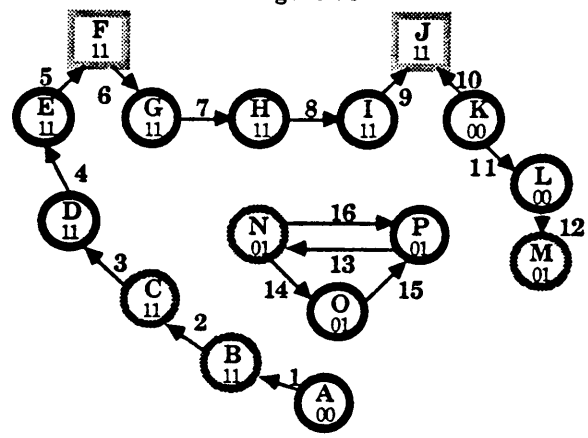

Figure 7c

Figure 7. Three Snapshots of Algorithm Two

In fact, the algorithm may have examined all of the arcs at least once at this point in its execution. This is because it is impossible to determine the order in which the acquaintance arcs are examined.

Figure 7c shows the configuration at the termination of the marking phase of the algorithm. The algorithm identifies the same garbage actors as Algorithm One.

Algorithm Two is more inefficient than Algorithm One because it must look at the entire system of actors in order to distinguish garbage from non-garbage. In the best case, Algorithm Two examines each acquaintance arc at least once. In the worst case, however, the algorithm examines each arc four or more times—depending on the number of acquaintances of an actor—because of the unpredictability of the selection of acquaintance arcs. Thus, Algorithm Two is worse than Algorithm One. Chapter 4 presents a more in-depth comparison and analysis the three garbage collection algorithms.

3.6 Algorithm Three

Algorithm Three is an efficient implementation of Algorithm Two, because it examines actors no more than twice. In addition, it does not need to examine inverse acquaintance arcs, one of the drawbacks of Algorithm One. Algorithm Three is more efficient because of a fundamental change in the way actors are examined.

Initialization Step. The state bits of each actor are initialized as follows: anchor point actors have the computable and reachable bits enabled; actors which are computable have the computable bit enabled; all other actors are blocked, and have no state bits enabled. All computable actors are placed into an internal set of actors.

Marking Step I. Repeat the following operations until the set of actors becomes empty.

1. Remove an actor from the set.
2. Enable the actor's visit bit.
3. Insert the actor's non-visited acquaintances into the set.
4. For each non-visited acquaintance of the actor:
   a. Apply the table lookup function to determine the rule that matches the current state bits of the owner and member actors of the acquaintance arc.
   b. If the rule of the table lookup function that matches the current state bits contains a "don't care" value, then insert a temporary acquaintance arc from the original member actor to the original owner actor. This arc is used in the second phase of the algorithm.
   c. Store the new state bits of the owner and member actors, enabling the "don't care" value bits.

Marking Step II. Place all anchor point actors into an internal set of actors. Repeat the following operations until the set becomes empty.

1. Remove an actor from the set.
2. Disable the actor's visit bit.
3. Place the actor's visited acquaintances into the set.
4. For each visited acquaintance of the actor, compute and store the new state bits of the owner and member actors by applying the table lookup function.
5. If the acquaintance arc is a temporary arc placed by the first phase of the algorithm, remove the arc.

Termination Step. Reclaim the garbage actors. Garbage actors are those with the reachable bit disabled.

Figure 8. Algorithm Three

3.6.1 Operation

Algorithm Three first propagates the computable bit by examining only actors which are not blocked. It then propagates the reachable bit on those actors which it examines in the previous step. The algorithm is presented in Figure 8.

The efficiency of Algorithm Three is not without cost: an additional state information bit must be introduced into the algorithm. The extra bit is used to indicate if the actor is visited and serves two important functions. First, it prevents the algorithm from entering an infinite loop when circular structures are encountered. Second, it optimizes the algorithm's second marking step. It does this by indicating if the actor is visited during the first step; actors that are not visited by marking step one are not examined by marking step two.

3.6.2 Example

To illustrate the operation of Algorithm Three, an example configuration of actors is presented in Figure 9. It is the same configuration that is used to show the operation of the previous two algorithms. Figure 9a shows the system of actors after the initialization step is complete. It is no different from Figure 7a, because the initialization of the actor state bits in Algorithm Three is identical to that of Algorithm Two.

Figure 9b displays the configuration after the execution of the first marking step. The state bits of actors **D**, **O** and **P** are changed to enable the computable bit. Actors **E**, **G**, **H** and **I** have both reachable and computable bits enabled. Temporary acquaintance arcs are inserted by the first step, so the second step is able to find all computable actors. The temporary arcs are drawn shaded to distinguish them from the acquaintances established by the application. Note that the temporary

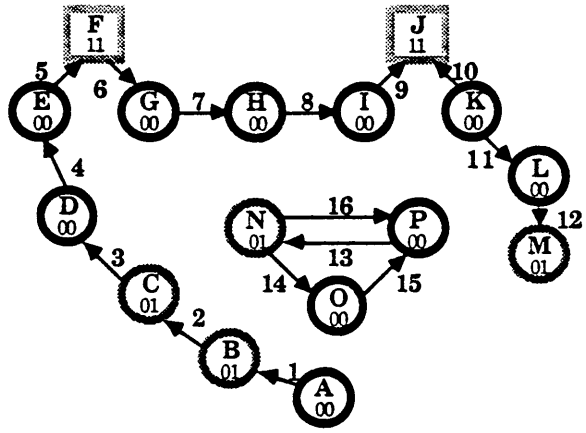

Figure 9a

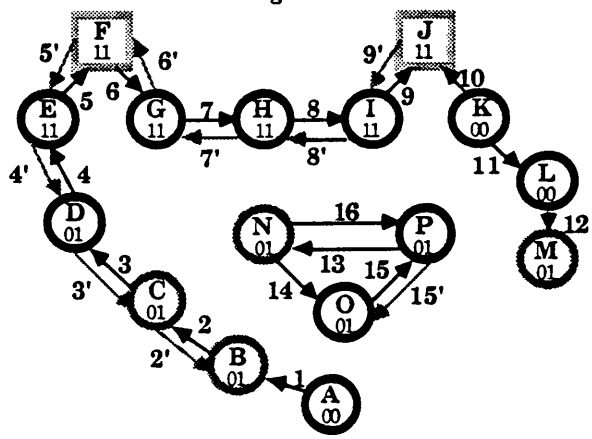

Figure 9b

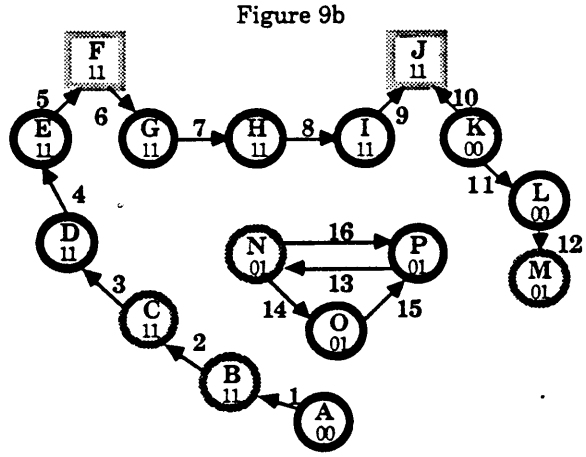

Figure 9c

Figure 9. Three Snapshots of Algorithm Three

acquaintance arc **3′** is created by the algorithm, even though the table lookup did not change the state bits of actor **D**. Because the state bits would have been changed if the actor were not already computable, the temporary acquaintance is created so the second step of the algorithm finds the computable actors beyond actor **D**.

A further examination of Figure 9b reveals the potential for an optimization of Algorithm Three. It is not necessary to insert temporary acquaintance arcs **6′**, **7′**, **8′** and **9′** because each member actor of the arcs is already marked reachable by the first marking step. Thus, there is no need for the second marking step of the algorithm to visit the actors again.

The configuration of actors after the marking steps are concluded is presented in Figure 9c. The temporary acquaintance arcs have been removed by the second marking step. The garbage actors identified by the algorithm are the same as those identified in Algorithms One and Two.

3.7 Summary

This chapter presented three algorithms for garbage collecting actors and gave examples of each. Each algorithm uses the scheduling state information of the actor in addition to its reachable property to conclude whether or not it is garbage.

4.0 Analysis and Implementation

This chapter presents an analysis of the three algorithms presented in Chapter 3. First, the correctness of Algorithm Three is informally proved. Second, the execution time complexity is derived for all of the algorithms. Third, the real-time execution characteristics of the algorithms are explored. Fourth, it is shown that Algorithms One, Two and Three cannot be directly compared with previous garbage collection algorithms, because the information used by the previous algorithms is not as complete as the information used by the three new algorithms. The chapter concludes with a recommended implementation strategy that may be used for either Algorithms One or Three.

4.1 Proof of Correctness

In order to show that Algorithm Three correctly identifies garbage actors, it is necessary to define precisely the term "garbage" and other terms used in the proof. The proof is an informal argument and is not a strict mathematical proof.

4.1.1 Definitions

Definition. An actor is **garbage** if it is unable to interact with an anchor point actor.

An actor that cannot interact with an anchor point actor is either unable to receive messages from an anchor point or is unable to send messages to an anchor point, or both. Anchor point actors are representations of devices in the physical world directly controlled by the real-time system. An actor that cannot communicate with these devices neither effects nor is affected by the physical world. Therefore, the actor may be safely removed from the real-time system. The real-time application produces the same results as seen by the physical world, with or without the garbage actor. (This conclusion assumes that the garbage actor does not tie up resources that prevent the application from successfully executing.)

Definition. A **path** is a series of one or more actors and acquaintances such that:

- any actor on the path is able to send a message along the path to an anchor point actor, or
- any actor on the path is able to receive a message along the path from an anchor point actor.

Note that the definition of a path refers to *ability*, and not necessarily the current configuration of actors and their acquaintances. For example, consider the simple actor configuration in Figure 10.

In this current configuration, actor **C** cannot send or receive a message from the anchor point actor **A**. However, because actor **B** is computable and because it knows the addresses of both **A** and **C**, it is able to change the configuration. Actor **B** can change the configuration by any one of these actions:

1. **B** sends a message to **A** containing the address of **B**. Now the anchor point actor can send a message to actor **B**, and it in turn can pass the message to actor **C**.

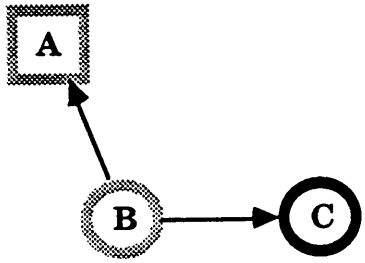

Figure 10. A Simple Actor Configuration

2. **B** sends a message to **A** containing the address of **C**. After this step, actor **C** is able to receive messages directly from the anchor point actor.

3. **B** sends a message to **C** containing the address of **A**. Now actor **C** can send messages directly to the anchor point actor.

4. **B** sends a message to **C** containing the address of **B**. Actor **C** can now send a message to actor **B** which can then pass the message to the anchor point actor.

Thus, all of the actors (including actor **C**) are on a path with an anchor point actor. Note, however, that if actor **B** is blocked, then the actors **B** and **C** are not on a path with actor **A**, because **B** cannot send any messages to change the configuration.

4.1.2 Statement of Informal Proof

The correctness of Algorithm Three is now informally proven.

Theorem 1. Algorithm Three correctly identifies all garbage actors.

The theorem is shown to be true in three steps. First, it is shown that the algorithm terminates. Next, it is shown that any actor which is not garbage is correctly identified as such by the garbage collector. This shows that the garbage collector does not incorrectly mark an actor as garbage. The final step of the informal proof shows that an actor that is garbage is correctly marked as such. This shows that the algorithm does not miss marking an actor as garbage.

4.1.2.1 Algorithm Termination

It is easy to show that Algorithm Three terminates. First, there is a finite number of actors to examine because memory is bounded. This bounds the initialization, marking and termination steps of the algorithm. Second, there are a finite number of bits in each actor that are enabled by the marking steps of the algorithm: two at the most. Finally, the number of visits the marking steps of the algorithm makes to an actor is bounded by a constant. This constant is two—the same as the number of bits used by Algorithm Three—because the marking steps are written in two passes, one for each bit. Each pass also enables a special "visited" bit to ensure that the actor is not visited more than once by each pass. Thus, Algorithm Three terminates.

4.1.2.2 Identifying Non-Garbage

The argument that Algorithm Three correctly identifies all non-garbage is by contradiction. It is shown that an actor that is reachable cannot be marked as garbage.

Lemma 1. If the reachable bit of an actor is disabled, then the actor is not on a path.

Lemma 1 is shown to be true by the operation of Algorithm Three. In the first marking step of Algorithm Three, the computable property is propagated by starting at all active actors (which are already computable) and marking all their acquaintances as computable. Temporary backward acquaintance arcs are inserted so the algorithm only has to follow forward acquaintances in the second marking step. The algorithm continues to propagate the computable property to the acquaintances of the acquaintances, and so on. Rules 3, 4, 5 and 6 from Table 2 propagate the computable property.

The second marking step of Algorithm Three starts with anchor point actors and propagates the reachable property by marking all their acquaintances as reachable. Next, the acquaintances of the acquaintances are marked, and so on. Rules 5 and 6 of the state table presented in Table 2 propagate the reachable property.

Assume that there exists an actor that is live, but is garbage collected by the algorithm. That means the algorithm did not set the actor's reachable bit, because it is the reachable bit that determines if the actor is considered garbage. If the reachable bit is not set, then from Lemma 1, the actor is not on a path with an anchor point actor. This means the actor cannot affect nor be affected by the physical world, so it is garbage. This contradicts the initial statement that says the actor is not garbage. Therefore, Algorithm Three correctly identifies all non-garbage.

4.1.2.3 Identifying Garbage

The last step of the informal proof is to show that Algorithm Three correctly identifies all garbage actors. The argument is by contraction, and shows that an actor that is not reachable is marked as garbage.

Assume that there exists an actor that is garbage, but is not garbage collected by Algorithm Three. That means that the algorithm must have set the actor's reachable bit. By Lemma 1, the actor must be on a path with an anchor point actor, which means the actor can affect the physical world and therefore is not garbage. This contradicts the initial statement that claims the actor is not garbage.

All three conditions of the informal proof of Theorem 1 are met, so Algorithm Three correctly identifies all actors that are garbage. The algorithm terminates, correctly marks non-garbage actors, and correctly marks garbage actors.

4.2 Complexity Analysis

This section analyzes and compares the execution characteristics of Algorithms One, Two and Three. It shows that Algorithms One and Three are better than Algorithm Two, because they examine fewer actors. However, no clear comparison between Algorithm One and Algorithm Three is possible because there are cases where each examines fewer actors than the other.

All three algorithms are analyzed to determine how many actors each examines in finding the garbage actors. In the following discussion only the marking phases are analyzed, because it is assumed that the initialization and termination steps are the same for each algorithm.

To analyze the execution complexity, actors are classified into one of four types. The first type is actors that are isolated. The second type is actors that are reachable. The third type, called connected, consists of actors that are neither reachable nor isolated. They are called connected because although they are connected to an anchor point actor through a series of one or more acquaintance arcs, they cannot communicate with the anchor point actor. Examples of these three types of actors

are shown in Figure 9c. Isolated actors are **N**, **O** and **P**, reachable actors are **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I** and **J** and connected actors are **A**, **K**, **L** and **M**. All of these types of actors are disjoint.

The fourth classification, actors that are active, is different from the others because it is not a disjoint classification. These are actors which have only their computable bit enabled. The classification is not disjoint because actors may be both isolated and active. For example, actors **N**, **O** and **P** in Figure 9c are both active and isolated.

4.2.1 Algorithm One

Algorithm One starts from the anchor point actors and follows the acquaintance arcs (forwards and backwards). It examines all actors except those that are isolated. Thus, its execution time is on the order of **O(reachable + connected)**. As shown in the example actor configuration in Figure 5c, Algorithm One examines a total of thirteen actors.

4.2.2 Algorithm Two

This algorithm must examine every actor in the system because it examines all acquaintance arcs at least once. Its execution time is on the order of **O(reachable + connected + isolated)**. The sum accounts for every actor in the system.

4.2.3 Algorithm Three

Algorithm Three examines all of the reachable and computable actors, so its execution time is on the order of **O(reachable + computable)**. As shown in the example actor configuration in Figure 9c, Algorithm Three examines a total of thirteen actors.

4.2.4 Summary

It is inconclusive as to which algorithm is the best. Algorithm Two is obviously the worst of the three because it examines every actor in the system, so it is not considered further. Both Algorithms One and Three include reachable actors in their execution times, so it is factored out of the execution time equations. The remaining terms, connected and computable, cannot be compared theoretically to determine which is larger than the other, because they are entirely dependent upon the configuration of the actors and acquaintance arcs in the real-time system. Thus, because Algorithm One examines reachable and connected actors and Algorithm Three examines reachable and computable actors, it is not possible to determine which algorithm is the best, because there are configurations that have more connected than computable actors (and Algorithm Three is best) and there are configurations that have more computable than connected actors (and Algorithm One is best).

4.3 *Real-time Analysis*

One of the goals of this research to produce a garbage collector that is real-time. By real-time, it is meant that the operations of the application (the mutator) that the garbage collector affects are bounded by a constant, as in the sense of Baker [BAKE78b]. These operations are:

1. Creation of acquaintances
2. Deletion of acquaintances
3. Access to acquaintances
4. Reclamation of actors

The operations are considered for these two classes of acquaintances:

- Static acquaintances
- Dynamic acquaintances

Static acquaintances are those that are explicitly named in the script of an actor's behavior. Dynamic acquaintances are generated when a new actor is created or when an actor's mail queue address is received in a message.

4.3.1 Creation of Acquaintances

At behavior initialization, the static acquaintance list of the behavior is communicated to the garbage collector by virtue of the actor language compiler. The compiler is written to insert a call to the garbage collection system containing the addresses of the static acquaintances. The list of static acquaintances is bounded by a constant, because the list is of a fixed size and cannot be changed.

Creation of a new actor is bounded, because the **new** function creates exactly one actor. Thus, the operation that affects the garbage collector is that exactly one actor is inserted into the acquaintance list.

4.3.2 Deletion of Acquaintances

When a behavior terminates, the acquaintances of that behavior must be removed from the actor acquaintance list. This is real-time because the acquaintance lists of each behavior are disjoint. A simple way to achieve real-time performance is to enable a bit in the kernel's information for the behavior. The garbage collector checks the bit and if set, does not include the acquaintances of the terminated behavior into the actor's acquaintance list.

4.3.3 Access to Acquaintances

To be real-time, the operations that access actors must be bounded by a constant. The access operations are sending a message to an acquaintance and passing an actor's mail queue address in a message.

Sending a message that does not contain mail queue addresses is bounded, because exactly one actor is named as the recipient of the message. In addition, the garbage collector is not informed when these messages are sent, nor must it be. That is because to send a message, the actor must already know the destination address; thus, the garbage collector is already aware of the acquaintance relationship.

However, sending a message that contains mail queue addresses requires intervention by the garbage collector, because the acquaintance list of the destination actor is being changed. To detect an ac-

tor's mail queue address being passed in a message to another actor, the actor language compiler and run-time message system must assist in garbage collection. One way this is achieved is for the message-passing system to check the data type of every item in a message and inform the garbage collector if any mail queue addresses are being sent. This is similar to Baker's algorithm [BAKE78b] where it verifies that all accessed objects are in tospace and contain tospace pointers. An optimization is for the message-passing system to inform the garbage collector only when it knows an address is being passed, similar to Ellis's [ELLI88] optimization of Baker's algorithm. This is realized by the implementation of the actor language, as follows. The ACT++ language, an actor language built on top of C++ [KAFU88] supports overloading of operators. The overloading is also data type dependent. That is, an operator like "+" generates different code if, say, the operands are integer values than if the operators are complex values.

Therefore, to send mail queue addresses in real-time, the compiler assists the garbage collector at compile time. The operator that sends or builds a message for transmission is overloaded depending upon the data type of the item being sent in the message. If the data type of the item is an actor's mail queue address, the compiler generates code to inform the garbage collector that an actor mail queue address is being passed in a message. The garbage collector inserts the new acquaintance into the destination actor's acquaintance list. Because there is just one acquaintance and because the insertion is bounded by a constant, sending messages containing actor mail queue addresses is also bounded by a constant.

4.3.4 Reclamation of Actors

All of the previous functions (creation, deletion and access) are those that the mutator explicitly initiates. This function, reclaiming actors, is the responsibility of the garbage collector. Reclaiming in incremental, real-time garbage collectors like Baker's [BAKE78b] is synchronized with object

creation: each time a new object is created, a constant amount of work is expended to reclaim a constant amount of non-garbage objects.

None of the three new actor garbage collection algorithms are real-time in the sense of Baker, because none can reclaim a constant amount of non-garbage actors with a constant amount of work. Algorithm Two is not real-time because there is no bound on the number of acquaintance arcs it examines before it marks actors as non-garbage.

Algorithm One is not real-time because a non-garbage actor must be on a path with an anchor point, and paths are of arbitrary length. For example, consider the actors **C**, **D**, **E** and **F** in Figure 5c. For Algorithm One to mark actor **C** as non-garbage, it must follow the acquaintance arcs backwards starting from actor **F**. Because the number of actors between **F** and **C** may be arbitrarily large, Algorithm One cannot determine if an actor is garbage in a bounded number of steps.

Algorithm Three does not execute in real-time because both bits must be evaluated before an actor is known to be non-garbage. Similar to Algorithm One, the number of steps between the enabling of one bit and the enabling of the other may be arbitrarily large. Consider the configuration of actors in Figure 9c. For Algorithm Three to mark actor **C** as non-garbage, it must follow the acquaintance arcs from **C** to **F** and back again along the temporary arcs (shown in Figure 9b). This path may be arbitrarily long, therefore Algorithm Three cannot determine if an actor is garbage in a bounded number of steps.

Therefore, because two marking steps are required to garbage collect actors and because paths can be arbitrarily long, it is doubtful that any garbage collector of actors modeled after traditional garbage collectors can be made real-time.

4.4 Comparison with Other Garbage Collectors

The lack of the real-time characteristic in all three algorithms presented in Chapter 3 makes it tempting to use a traditional, incremental, real-time garbage collector like ones presented in Chapter 2 for the storage management of actors in a real-time system. To do this, the traditional collector must be modified to examine acquaintance arcs in either the forward or backward direction, so that any actor that is reachable or connected is considered non-garbage by the algorithm. Otherwise, non-garbage actor and acquaintance structures (like the ones connecting actors **C**, **D**, **E** and **F** in Figure 5c) are incorrectly marked as garbage. Choosing a garbage collector modified in this way is incorrect, for two reasons.

First, the processor is an important resource in real-time systems. Computations in real-time systems have hard deadlines to meet. If a computation misses its deadline, the computation and perhaps the entire system fails. An actor which is computable but not reachable wastes processor resources computing results that have no effect on the physical world. Even worse, the actor may be scheduled so that it causes other non-garbage actors to miss their deadlines.

A stronger reason not to use a modified traditional garbage collector is that—because they use only one characteristic to determine if an actor is garbage—they mark as non-garbage not just actors with the reachable property, but also all of the actors with only the connected property. This means that actors with the connected property are considered non-garbage by the modified traditional garbage collector and therefore, more memory is required because not all garbage actors are reclaimed. If this is the only drawback, a simple recommendation is to procure sufficient memory to avoid this shortcoming. However, consider the configuration of actors **K**, **L** and **M** in Figure 5c. Note that these actors are considered garbage by the three algorithms presented in the previous chapter. In particular, actor **M** is garbage even though it is computable. The modified traditional garbage collector marks these actors as non-garbage because they are connected to an anchor point actor.

Because **M** is computable, it may execute and create new actors (which must also be garbage). Thus, unreclaimed garbage actors are able to create more garbage actors, which the modified traditional garbage collector cannot detect. This is a serious problem because it means that eventually memory may be consumed and ultimately exhausted by garbage actors. Note that even the algorithm by Metha and Christoper [METH88] that is designed specifically for actor systems [CHRI88] fails to mark actors **K**, **L** and **M** as garbage.

4.5 Recommended Implementation

The traditional real-time garbage collectors which only examine the connected property are not suited for garbage collection of actors. The garbage collectors presented in Chapter 3 are suited for the task, but they are not real-time. This presents a dilemma which must be overcome if actors are to be useful in real-time systems. The proposed solution is to make each node of the network a shared memory multiprocessor system. The garbage collector, say, Algorithm Three, executes in one of the processors, using compacting mark and sweep garbage collection techniques. The application runs in the remaining processor(s). All processors operate on a common, shared address space.

The processor executing the garbage collection algorithm may be used for other real-time functions besides garbage collection. For example, Stankovic and Ramamritham [STAN87] suggest that the best place for a real-time scheduling algorithm is in a separate processor. This claim is made for two reasons. First, the real-time scheduler must schedule the real-time tasks effectively so that they meet their deadlines. Such a scheduler, particularly in a distributed system, may take an arbitrary amount of time to determine the scheduling sequence of tasks, given factors such as the number of real-time tasks to execute, their deadlines, and other relevant scheduling information. However,

while the scheduler is executing, the real-time tasks are not. Thus, the scheduler prevents the tasks from executing and may cause them to miss their deadlines.

The second reason why scheduling should be delegated to an attached processor is predictability. The scheduler executes at unpredictable times because blocked tasks become unblocked and aperiodic tasks appear unpredictably. Both must be scheduled into execution along with the other executing tasks, which requires a recomputation of the execution order of tasks.

4.6 Summary

This chapter presented an informal proof that Algorithm Three correctly identifies garbage actors. It was shown that traditional garbage collection algorithms cannot identify all garbage actors, because they consider only one bit of information. However, Algorithms One, Two and Three are not real-time. Further, it is possible that no actor garbage collection algorithm can be made real-time. Therefore, the solution to performing non-real-time actor garbage collection that does not interfere with the mutator is to execute either Algorithm One or Algorithm Three in a shared memory multiprocessor.

5.0 Conclusions and Directions for Future Work

This chapter presents conclusions and suggests areas of future research.

5.1 Conclusions

It is worthwhile to evaluate the actor garbage collection algorithms presented in Chapter 3 to determine how well the original goals stated in Chapter 1 are met. The goals that motivated the development of a garbage collector are that it should be:

- Incremental. The new actor garbage collection algorithms are incremental, because they may be implemented to synchronize allocation of actors with garbage collection of a constant number of actors.
- Real-time. The algorithms are not real-time, because the bound on the number of steps needed to recover non-garbage actors is not bounded by a constant. This is overcome by implementing the garbage collector on an attached processor.
- Concurrent. The garbage collector is concurrent because it executes on an attached processor.

- Automatic. The garbage collection algorithms developed are not explicitly invoked by the application. They execute automatically and independently.
- Distributed. None of the garbage collectors are distributed. This is an area for future work.

Therefore, nearly all of the goals are met. Because the real-time goal is not achieved, another important conclusion that may be drawn from this research is that it may be impossible to implement a real-time actor garbage collection algorithm because two marking steps are required to correctly identify non-garbage actors and because paths may be of arbitrary length.

5.2 *Future Work*

There are many areas of interest for future work. They include the following:

1. The impact of garbage collection on an actor language
2. Deadline scheduling
3. Distributed garbage collection
4. Empirical measurements of real-time actor systems
5. Structured programming rules

5.2.1 Actor Language Design

The actor language and run-time support system must assist the garbage collector in learning about the acquaintances of an actor and when the actor's acquaintance list changes. This impacts the design of the actor language and also the language's run-time support. The recommendations in Chapter 4 for implementing this support need to be analyzed for their feasibility.

Another impact concerns the ACT+ + language [KAFU88]. Included in the ACT+ + language is the concept of a special mail queue, called a cbox. A cbox is a programming convenience that simplifies continuations [AGHA86]. Instead of specifying a replacement behavior to receive the results of a computation, a cbox allows the current behavior to receive messages other than the initial message in the actor mail queue. A behavior creates a cbox, sends a message to an actor specifying the cbox address as the mail queue address to receive the results of the computation, and then issues a read statement specifying the cbox address. The read statement blocks until the cbox contains a message.

The impact of the cbox construct upon the garbage collector is unclear, even though cboxes are expressible in terms of the original actor model.

5.2.2 Deadline Scheduling

Real-time applications have hard deadlines to meet; if the deadline is missed, then the computation and perhaps the entire system fails. Garbage collection algorithms for real-time systems should incorporate the concept of deadlines into their design. For example, the scheduler can assist in garbage collection by refusing to schedule tasks whose deadlines are expired. Instead, the scheduler informs the garbage collector that the task is garbage.

5.2.3 Distributed Acquaintances

The garbage collection algorithms developed in this thesis cannot handle distributed acquaintances; an actor garbage collection algorithm for distributed systems must be able to deal with an actor having an acquaintance on another processor. One way this may be realized is in the same way that Lieberman [LEIB83] deals with interprocessor addresses, where each processor in the network

maintains two special indirection tables. The first table, called the exit table, records the references to actors on other processors by the actors on the current processor. The second table, the interest table, records references to actors on the current processor from other processors in the network. A protocol for updating the tables and implementing distributed garbage collection is an area for future work.

5.2.4 Empirical Measures

Real-time actor systems need to be studied and measured for three reasons. First, the length of paths need to be determined to discover if the actor garbage collection algorithms may be safely implemented on a single-processor system without seriously impacting the mutator. Recall that it is because paths may be arbitrarily long that makes the actor garbage collection algorithms fail the real-time property.

Second, studies are needed to determine the types of garbage actors that are most frequent: actors that are connected, or actors that are active. This helps determine which actor garbage collection algorithm to implement, because Algorithm One is better than Algorithm Three when there is less connected and more active actors, and Algorithm Three is better than Algorithm One when there is more connected and less active actors.

Third, real-time actor systems need to be studied to determine how they are configured. Traditional garbage collectors are effective with one bit of information because the systems they operate upon are all configured so that pointers to non-garbage objects always descend (point "down") from another non-garbage object, and so on, eventually terminating at a root object. In actor systems, the configuration of acquaintances is not so constrained. Not only is non-garbage pointed to by root objects, non-garbage can also point to root objects.

5.2.5 Structured Programming Techniques

It may be useful to investigate structured programming techniques to see if rules exist to help constrain actor configurations. These rules would restrict the configuration in two ways. First, they would restrict the configuration to conform to the way that traditional objects are configured, so that traditional garbage collectors may be used to garbage collect actors. Second, if the first restriction fails, then the constraints would attempt to bound the length of paths, so that actor garbage collectors can be made real-time.

Bibliography

[AGHA86] Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*, M.I.T. Press, Cambridge, Massachusetts, 1986.

[AGHA87] Agha, Gul and Carl Hewitt. "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," *Research Directions in Object-Oriented Programming*, Schriver and Wegner, eds., M.I.T. Press, Cambridge, Massachusetts, 1987, pp. 49-74.

[AGHA88] Agha, Gul, Won Kim, Mark Scheevel, Vineet Singh, Chris Tomlinson and Rebecca Will. "Rosette: An Object-Based Concurrent Systems Architecture," MCC Technical Report, ACA-ST-254-88(Q): August, 1988.

[APPE88a] Appel, Andrew W., John R. Ellis and Kai Li. "Real-time concurrent Collection on Stock Multiprocessors," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* June, 1988.

[APPE88b] Appel, Andrew W. Private correspondence, November 3, 1988.

[BAKE77] Baker, Henry G., Jr. and Carl Hewitt. "The Incremental Garbage Collection of Processes," M.I.T. Artificial Intelligence Laboratory, Memo 454, December, 1977.

[BAKE78a] Baker, Henry G. "Actor Systems for Real-time Computation," M.I.T. Laboratory for Computer Science, Technical Report 197, March, 1978.

[BAKE78b] Baker, Henry G. "List Processing in Real-time on a Serial Computer," Communications of the ACM **21**(4): April, 1978, pp. 280-294.

[BENA84] Ben-Ari, Mordechai. "Algorithms for On-the-fly Garbage Collection," ACM Transactions on Programming Languages and Systems, **6**(3): July, 1984, pp. 333-344.

[BLOO87] Bloom, Tony and Stanley B. Zdonick. "Issues in the Design of an Object-Oriented Database Programming Language," ACM Conference on Object Oriented Programming Systems, Languages and Applications ("OOPSLA"), October, 1987, pp. 441-451.

[BOBR68] Bobrow, D. G. "Storage Management in LISP," *Proceedings of the IFIP Working Conference on Symbol Manipulation Languages and Techniques,* D. G. Bobrow (ed.), North-Holland(1968), pp. 291-301.

[BROO84] Brooks, Rodney A. "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* 1984.

[CHRI84] Christopher, Thomas W. "Reference Count Garbage Collection," IEEE Software—Practice and Experience **14**(6): June, 1984, pp. 503-507.

[CHRI88] Christopher, Thomas W. Private correspondence, December, 1988.

[COHE81] Cohen, Jacques. "Garbage Collection of Linked Data Structures," ACM Computing Surveys **13**(3): September, 1981, pp. 341-367.

[COLL60] Collins, G. E. "A Method for Overlapping and Erasure of Lists," Communications of the ACM **3**(12): March, 1960, 655-657.

[COUR88] Courts, Robert. "Improving Locality of Reference in a Garbage-Collecting Memory Management System," Communications of the ACM **31**(9): September, 1988, 1128-1138.

[DEUT76] Deutsch, L. Peter and Daniel G. Bobrow. "An Efficient, Incremental, Automatic Garbage Collector," Communications of the ACM **19**(9): September, 1976, pp. 522-526.

[DIJK78] Dijkstra, Edsger W., Leslie Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation," Communications of the ACM **21**(11): November, 1978, pp. 966-975.

[ECKA87] Eckart, J. Dana and Richard J. LeBlanc. "Distributed Garbage Collection," *Interpreters and Interpretive Techniques, SIGPLAN 1987 Symposium,* St. Paul, MN, June 24-26, 1987, pp. 264-273.

[ELLI88] Ellis, John R., Kai Li and Andrew W. Appel. "Real-time Concurrent Collection on Stock Multiprocessors," Digital Equipment Corporation Systems Research Center, Technical Report 25, February, 1988.

[FENI69] Fenichel, Robert R. and Jerome C. Yochelson. "A LISP Garbage-Collector for Virtual-Memory Computer Systems," Communications of the ACM **12**(11): November, 1969, pp. 611-612.

[FISH74] Fisher, David A. "Bounded Workspace Garbage Collection in an Address-order Preserving List Processing Environment," Information Processing Letters **3**(1): July, 1974, pp. 29-32.

[FITC78] Fitch, J. P. and A. C. Norman. "A Note on Compacting Garbage Collection," The Computer Journal **21**(1): February, 1978, pp. 31-34.

[FRIE76] Friedman, Daniel P. and David S. Wise. "Garbage Collecting a Heap which Includes a Scatter Table," Information Processing Letters **5**(6): December, 1976, pp. 161-164.

[GLAS87] Glaser, H. W. and P. Thompson. "Lazy Garbage Collection," IEEE Software—Practice and Experience **17**(1): January, 1987, pp. 1-4.

[HADD67] Haddon, B. K. and W. M. Waite. "A Compaction Procedure for Variable Length Storage Elements," The Computer Journal **10**(2): August, 1967, pp. 162-165.

[HALS78] Halstead, Robert Hunter, Jr. "Multiple-Processor Implementations of Message-Passing Systems," M.I.T. Laboratory for Computer Science, Technical Report 198, April, 1978.

[HANS69] Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation," Communications of the ACM **12**(9): September, 1969, pp. 499-507.

[HANS77] Hanson, David R. "Storage Management for an Implementation of SNOBOL4," IEEE Software—Practice and Experience **7**(2): March, 1977, pp. 179-192.

[HICK84] Hickey, Tim and Jacques Cohen. "Performance Analysis of On-the-fly Garbage Collection," Communications of the ACM **27**(11): November, 1984, pp. 1143-1154.

[JONK79] Jonkers, H. B. M. "A Fast Garbage Compaction Algorithm," Information Processing Letters **9**(1): July, 1979, pp. 26-30.

[KAFU88] Kafura, Dennis G. "Concurrent Object Oriented Real-Time Systems Research," Virginia Polytechnic Institute and State University, Department of Computer Science, Technical Report 88-47, September, 1988.

[KNUT68] Knuth, Donald E. *The Art of Computer Programming, Volume 1, Fundamental Algorithms,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1968.

[LANG87] Lang, Bernard and Francis Dupont. "Incremental Incrementally Compacting Garbage Collection," *Interpreters and Interpretive Techniques, SIGPLAN 1987 Symposium,* St. Paul, MN, June 24-26, 1987, pp. 253-263.

[LERM86] Lermen, Claus-Werner and Dieter Maurer. "A Protocol for Distributed Reference Counting," *Proceedings of the 1986 ACM Conference on LISP and Functional Programming,* pp. 343-350. **16**(2): February, 1986, pp. 145-163.

[LIEB83] Lieberman, Henry and Carl Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects," Communications of the ACM **26**(6): June, 1983, pp. 419-429.

[MCCA60] McCarthy, J. "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I," Communications of the ACM **3**(4): April, 1960, pp. 184-195.

[MEHT88] Mehta, Sanjay and Thomas W. Christopher. "Garbage Collection of Distributed Objects," Illinois Institute of Technology, 1988.

[MORR78] Morris, F. Lockwood. "A Time- and Space-Efficient Garbage Compaction Algorithm," Communications of the ACM **21**(8): August, 1978, pp. 662-665.

[MORR79] Morris, F. Lockwood. "On a Comparison of Garbage Collection Techniques," Communications of the ACM **22**(10): October, 1979, p. 571.

[NILS88] Nilsen, K. "Garbage Collection of Strings and Linked Data Structures in Real Time," IEEE Software—Practice and Experience **18**(7): July, 1988, pp. 613-640.

[RUDA86] Rudalics, Martin. "Distributed Copying Garbage Collection," *Proceedings of the 1986 ACM Conference on LISP and Functional Programming,* pp. 364-372.

[SCHO67] Schorr, H. and W. M. Waite. "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Communications of the ACM **10**(8): August, 1967, pp. 501-506.

[STAN87] Stankovic, J. A. and K. Ramamritham. "The Design of the Spring Kernel," *Proceedings of the Real-Time Systems Symposium,* December, 1987, pp. 146-157.

[STEE75] Steele, Guy L., Jr. "Multiprocessing Compactifying Garbage Collection," Communications of the ACM **18**(9): September, 1975, pp. 495-508.

[TERA78] Terashima, Motoaki and Eitchi Goto. "Genetic Order and Compactifying Garbage Collectors," Information Processing Letters **7**(1): January, 1978, pp. 27-32.

[THOR76] Thorelli, Lars-Erik. "A Fast Compactifying Garbage Collector," BIT **16**(4): 1976, pp. 426-441.

[WADL76] Wadler, Philip L. "Analysis of an Algorithm for Real Time Garbage Collection," Communications of the ACM **19**(9): September, 1976, pp. 491-500.

[WEGB72] Wegbreit, B. "A Generalized Compactifying Garbage Collector," The Computer Journal **15**(3): August, 1972, pp. 204-208.

[WISE77] Wise, David S. and Daniel P. Friedman. "The One-bit Reference Count," BIT **17**(3): 1977, pp. 351-359.

Glossary of Terms

This appendix contains a glossary of terms that are used to describe actors, traditional garbage collectors and the new garbage collectors described in this thesis.

Acquaintances. An acquaintance is an actor mail queue address that is known by a behavior.

Active. An actor is active if its computable bit is enabled.

Actors. An actor is a concurrent computational object composed of a mail queue, a set of behaviors, and a set of acquaintances. The actor computation model is designed to explore concurrency issues.

Actor acquaintance list. The acquaintance list of an actor is formed by a union of all of the actor's behavior's acquaintance lists. The unioned acquaintance list is combined with the mail queue addresses stored in the actor's mail queue of unread messages to form the actor's acquaintance list. The actor acquaintance list identifies all of the actors that can be addressed by the given actor; that is, all of the actors to which the specified actor can send a message. An actor always has itself as an acquaintance.

Actor inverse acquaintance list. The inverse acquaintance list identifies all of the actors that can address the given actor; that is, all of the actors that can send a message to the specified actor. Note that the inverse acquaintance list is never empty, because an actor always knows its own address.

Anchor point. This term is used to define an actor that is the real-time application's internal representation of objects by which the application monitors and controls the real world. It is analogous to the objects pointed to by the mutator's registers in traditional garbage collection schemes. Examples of physical objects that have this classification are sensors, actuators and clocks.

Application. An application is a synonym for mutator: it is the name of the real-time computation.

Automatic. A garbage collector is automatic if it executes without the knowledge of the mutator; the mutator does not invoke it explicitly.

Become. The become operation specifies a replacement behavior for the current behavior. It also specifies the acquaintance set of the replacement behavior. If a behavior reaches the end of its script without nominating a replacement behavior, the current behavior with the current acquaintance list is nominated by default. The become operation may be specified no more than once by a behavior.

Behaviors. A behavior is a script that defines how to process a message received in the actor's mail queue.

Blocked. An actor is blocked if all of the following conditions hold:

1. The actor's mail queue is empty.
2. The actor has exactly one behavior.
3. The behavior is awaiting the arrival of an as-yet unsent message in the actor's mail queue.

See computable.

Collector. The shorthand name of the garbage collector. The garbage collector reclaims garbage objects and returns them to the free pool of storage.

Compacting mark and sweep collectors. A compacting mark and sweep collector relocates garbage and live data to opposite ends of the memory space as the collector classifies the objects. Compaction results in smaller working set sizes and defragments memory.

Computable. An actor is computable if it has at least one executing behavior or if it is the recipient of an as-yet unreceived message. Computable and blocked are mutually exclusive.

Computational objects. A computational object is one that is associated with code and data. Computational objects are subject to scheduling and execution. Examples are processes, lightweight processes and actors.

Concurrent. A garbage collector is concurrent if it executes simultaneously with the mutator on an attached processor.

Connected. An actor is considered connected to a root object if it is neither reachable nor isolated.

Distributed garbage collectors. A garbage collector is distributed if it cooperates with garbage collectors on other processors in a local area network to reclaim garbage.

Dynamic acquaintances. A dynamic acquaintance is an actor that is either created with the **new** function, or is an actor's mail queue address.

Flip. A flip is executed when tospace fills to within a threshold value. A flip reverses fromspace and tospace and copies the root objects from fromspace to tospace.

Fromspace. Fromspace is the half of memory that is being garbage collected.

Garbage. An object is garbage if it is not accessible by the mutator. Further, an actor is garbage if it is unable to interact with an anchor point actor.

Generational garbage collector. The generational garbage collector is an extension of the semi-space copying collector. Instead of just two memory regions, there are many regions. The garbage collector executes a semi-space copying algorithm on the newer regions more frequently than on older regions, because most garbage is short-lived.

Generations. The name given to the semi-spaces of memory in a generational garbage collector.

Incremental. A garbage collector is incremental if, instead of reclaiming all of memory when it is invoked, reclaims a constant number of objects. If the reclaiming of a constant number of objects can also be done in real-time, the garbage collector is a candidate for meeting the real-time criteria (see the definition of real-time).

Isolated. An actor is isolated if its inverse acquaintance list contains no addresses other than its own. If an isolated actor is blocked, then it remains blocked forever, because no other actor can send it a message to cause it to become computable. Isolated actors are garbage.

Live data. An object is live data if it is accessible by the mutator.

Mail queue. A mail queue is the storage for messages addressed to a single actor. Mail queues are assumed to have infinite storage. The mail queue address is used to name the actor.

Mark and sweep. Mark and sweep garbage collectors and their derivatives are the most common garbage collector. The collector operates in two phases: first, it marks all the non-garbage objects. Second, it sweeps through memory and reclaims the unmarked objects as garbage.

Marking algorithm. A marking algorithm classifies objects into either live data or garbage.

Member. The member of an acquaintance is the actor that is the target of the acquaintance.

Message. A message is a communication sent to an actor. Besides data, messages may also contain actor mail queue addresses.

Mutator. The application that creates objects and modifies the relationships between objects.

New. The **new** operation creates a new mail queue address and a new actor associated with the queue. The new actor's initial behavior and initial set of acquaintances are the two parameters to the operation. The **new** operation returns the mail queue address of the new actor, so the behavior invoking the operation has its acquaintance set increased by one.

New area. The new area of tospace is where new objects are created.

On-the-fly garbage collectors. An on-the-fly garbage collector is a concurrent multiprocessor garbage collector. The collector executes a mark and sweep algorithm on one processor while the mutator executes in the remaining processor(s). The processors share the same physical memory.

Owner. When observing an acquaintance between two actors, the actor that is the originator, or holder, of the acquaintance is the owner of the acquaintance.

Path. A path is a series of one or more actors and acquaintances such that:

- any actor on the path is able to send a message along the path to an anchor point actor; or
- any actor on the path is able to receive a message along the path from an anchor point actor.

Reachable. An actor X is reachable if an actor with the anchor point characteristic can send a message to X or receive a message from X, either directly (no intervening actors) or indirectly (with some actors in between the two). An actor that is not reachable is **unreachable.** Unreachable actors are garbage.

Real-time. A garbage collection algorithm is real-time if the creation of objects, access to objects (both by the mutator), and reclamation of objects (by the garbage collector) are all bounded by a constant.

Reference counting garbage collectors. A reference counting garbage collector keeps track of pointers to an object by a reference counter field associated with the object. Creating a new pointer to the object increments the reference count; destroying a pointer to the object decrements the reference count. When the reference count reaches zero, the object is garbage.

Registers. The mutator's registers contain addresses of root objects (roots).

Roots. Root objects are the objects which are directly accessible by the mutator through its registers. All live data is accessible through the root objects. In actor systems, root objects are those with the anchor point characteristic set.

Scanned area. The scanned area of tospace contains objects that have been scavenged by the garbage collector.

Scavenging. Scavenging is when the pointers in an object are checked to verify that they point in tospace. If the pointers are to fromspace, the pointed object is relocated to tospace, a forwarding address is left behind at the old location, and the pointers are updated.

Semi-space copying collector. The semi-space copying garbage collector is an improvement over mark and sweep collectors, because not all of memory needs to be examined to reclaim garbage. Memory is divided into two regions, fromspace and tospace, and the collector copies

non-garbage objects from fromspace to tospace; objects left behind in fromspace are garbage. When tospace is full, a flip is performed: fromspace and tospace switch roles.

Send. The **send** operation transmits a message to an actor's mail queue. The operation accepts two arguments: the destination mail queue address and the message to be delivered. The operation completes asynchronously.

Static acquaintances. A static acquaintance is one that is explicitly named in the script of an actor's behavior.

Tospace. Tospace is the half of memory that is being used by the mutator to access objects and allocate new objects.

Virtual memory collectors. A virtual memory garbage collector is an extension of both on-the-fly and semi-space copying collectors. The collector's implementation uses virtual memory page protections and paging techniques to reclaim garbage.

Unscanned area. The unscanned area of tospace contains objects that have been relocated from fromspace but have not yet been scavenged.

The two page vita has been removed from the scanned document. Page 1 of 2

The two page vita has been removed from the scanned document. Page 2 of 2