Acid-Free Copy

# SURF: AN ABSTRACT MODEL OF DISTRIBUTED GARBAGE COLLECTION

William Brodie-Tyrrell

February 2008

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

# Contents

# List of Figures

# Abstract

Garbage collectors (GCs) automate the problem of deciding when objects are no longer reachable and therefore should be reclaimed, however, there currently exists no automated process for the design of a correct garbage collector. Formal models exist that prove the correctness of individual GCs; more general models describe a wider range of GCs but do not prove their correctness or provide a concrete instantiation process. The lack of a formal model means that GCs have been designed in an ad-hoc manner, published without proof of correctness and with bugs; it also means that it is difficult to apply experience gained from one implementation to the design of another.

This thesis presents **Surf**, an abstract model of distributed garbage collection that bridges the gap between expressibility and specificity: it can describe a wide range of GCs and contains a proof of correctness that defines a list of requirements that must be fulfilled. Surf's design space and its requirements for correctness provide a process that may be followed to analyse an existing collector or create a new GC.

Surf predicts the abstract behaviour of GCs; this thesis evaluates those predictions in light of the understood behaviour of published GCs to confirm the accuracy of the model. A distributed persistent implementation of the Train Algorithm is created as an instantiation of Surf and the model is used to analyse progress in the GC and drive the design of a partition selection policy that provides a lower bound on progress and therefore reduces the GC's complexity to completeness. Tests with mesh data structures from finite element analysis confirm the progress predictions from Surf.

Published GCs cluster mostly in one corner of the Surf design space so this thesis explores the design of a GC at an unoccupied design point: the Tram Algorithm. Analysis via Surf leads to the prediction that Trams are capable of discovering topology in the live object graph that approximately identifies the strongly connected components, permitting $O(1)$ timeliness that is unique to the Tram Algorithm.

# Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University of Adelaide Library, being made available in all forms of media, now or hereafter known.

William Brodie-Tyrrell
May 3, 2008

# Acknowledgments

I would like to thank my supervisors, Assoc. Prof. David S. Munro and Dr Katrina Falkner, for their unceasing support, insight, guidance and motivation; this thesis would not exist without their Herculean efforts. The Jacaranda Research Group has provided a friendly research environment; I thank Dr Henry Detmold for the breadth and sharpness of his insight, the postgraduate students of the School for their time, encouragement and camaraderie, and the staff of the School for their support.

I also thank my family for their years of unquestioning support and encouragement. In particular, my parents have demonstrated that the term "standing on the shoulders of giants" refers not only to the work of previous academics.

This work was supported by the Commonwealth of Australia through an Australian Postgraduate Award and Australian Research Scholarship (National). Cluster computation time for experimentation was donated by the South Australian Partnership for Advanced Computing.

# Chapter 1

# Introduction

Computation has three fundamental requirements: a processing unit that performs arithmetic and logical operations, software that defines the actions taken by the processing unit and storage containing the data that the computation is operating over. This thesis is concerned with automatic storage management, specifically an abstract model of distributed garbage collection, i.e. analysis of the class of algorithms concerned with reclaiming unused storage in a distributed system.

The ability to re-use storage space is important because it is a finite resource; programs typically generate new state and there must exist some means to reuse the space that is occupied by state that is no longer required, i.e. no longer live. Manually deallocating space when it is no longer required has proven to be error-prone and it is a particularly difficult task in distributed systems because the liveness of any particular piece of data is a global predicate. Detecting that some data is no longer live in a distributed system requires the knowledge that no references to that data exist on any other site or in transit between other sites; obtaining this knowledge in the face of asynchronous communication is a non-trivial task. It is therefore desirable to have some means of automatically analysing the state of a program and determining which storage regions are no longer required; providing this functionality can improve the reliability of programs by removing the opportunity for application programmers to erroneously discard state that is still in use or fail to reclaim all unused space.

There exists a spectrum of abstractions for representing storage, varying in the interface, underlying mechanism and the manner in which updates are applied. The interface defines how storage is accessed: addressing, granularity of access and whether the store understands any semantics beyond numerical values; at the simplest extreme is a linear array of numbers while more complex systems introduce the concepts of pointers, records and strong typing [1, 25]; still other systems may represent data as tables of records that may be searched [29]. The

combination of records and pointers produces the **object** abstraction: typically small regions of memory composed of fields where some fields contain **pointers**: the names of other objects. The contents of an object store form a directed graph wherein each object is a node of that graph and the pointers represent directed edges between objects. Since it is possible to construct a graph wherein one region is unreachable from another region, this admits the possibility that some objects will be unreachable from a special **root object** from which the application operates; these objects are unusable to the application and should be reclaimed. The process of detecting and reclaiming unreachable objects is referred to as **garbage collection** (GC).

A distributed system is one composed of a number of physically separate computing sites, each site typically containing a processing unit, store and software of its own with some means to communicate with other sites in the system. The distinction between a simple network of computers and a distributed system is that the latter closely cooperates to perform a particular task, typically a computation that is distributed across a number of sites with the aim of increasing the net performance of the computation. Distributed systems have become an attractive way to construct computing systems that are scalable in their storage and computing power: if software can be made scalable, then the performance and available storage space of a distributed system may be increased merely by adding storage and computing resources. This is in contrast to uniprocessor systems where there is a nonlinear relationship between price and performance so financial and physical constraints severely limit their scalability.

The increasing use of distributed software to solve compute-intensive problems brings with it a need for distributed storage systems that provide the same abstractions as available on uniprocessor systems, including the ability to operate over objects and have them automatically reclaimed. Providing the object abstraction with garbage collection in a distributed system implies the existence of a **distributed garbage collector**: a mechanism that is capable of finding isolated components of an object graph that is distributed over physically separated sites. The existence of distributed garbage collection permits application programmers to more easily construct reliable distributed programs, thereby reducing the cost of producing reliable scalable software.

Garbage collectors are often published [2, 18, 49, 67, 68] as a mechanistic description of the collector, tightly bound to the programming system in which it exists, as opposed to an abstract exposition of the garbage collection algorithm that underlies the collector. For this reason, it can be difficult to compare different collectors either empirically or analytically and it is difficult to apply knowledge and experience gained in the implementation

of one collector to the design of new collectors. Empirical comparisons are constrained by uncontrolled factors relating to the systems in which collectors are implemented. Analytic comparisons require a means to extract the essential garbage collection algorithms from detailed descriptions of implementations and then some common theoretical framework in which the algorithms may be analysed and compared. A secondary problem with the individual and mechanistic publication of collectors instead of algorithms is that the algorithm may not be proved and in some cases, published collectors have later been shown to be incorrect [42, 43, 87, 102].

A more recent trend (mostly but not entirely within the last decade) is the emergence of formal **models** of GC [64, 82, 102] that may be used to define one or more GC algorithms. If a model is capable of describing a range of garbage collection algorithms, it may be used to extract the essential garbage collection algorithm from an implementation, prove its correctness and serve as the common theoretical framework in which multiple algorithms may be analysed and compared. Formal models of GC therefore provide a rigorous means to analyse, construct and compare GC algorithms independently of the programming systems in which they may be implemented. As of the publication of this thesis, the most general existing model does not provide a concrete process to instantiate or analyse a collector, while the most formal and concrete models are very specific in the range of collectors they are capable of describing; these models are discussed in detail later.

The primary contribution of this thesis is the definition of a formal model of distributed garbage collection, **Surf**, that is capable of defining a range of garbage collectors, contains a formal proof of correctness that is applicable to collectors conforming to the model and provides a concrete process that may be followed to instantiate a new collector or analyse an existing collector.

Before going into detail on the specifics of existing research and the new Surf model in later chapters, this introduction will outline the required and desirable properties of a garbage collector, define what exactly is difficult about programming in a distributed system then introduce the intersection of these two areas of research: distributed garbage collectors and the models that describe them.

## 1.1 Garbage Collection

The purpose of a garbage collector is to reclaim those objects that are no longer required by the computation. Objects that will be accessed by the computation at a later time are considered **live** and all others are considered **dead**; most collectors

take a conservative approach by equating reachability and liveness. Clearly an object which is reachable will not necessarily be accessed but there is no way of exactly determining this without an analysis of the computation that is equivalent to the halting problem [23, 96]; techniques exist whereby compilers may be able to detect some such reachable dead objects though they are not considered here.

For the purpose of this thesis, analysis of the computation is ignored and reachable objects are defined as live, thereby reducing the GC problem to one of graph analysis. The GC problem is solvable only because unreachability is a stable property: once becoming unreachable, there is no way for an object to become reachable again since applications are permitted only to copy pointers, not manufacture them; the stability of garbage is critical in distributed systems because the global system state is not instantaneously observable.

This thesis is concerned only with the **detection of garbage**; methods of representing free space, allocating space and returning detected-as-unreachable space to the free pool are system specific and considered no further.

### 1.1.1 Required Properties of a Garbage Collector

The graph-connectivity definition of garbage collection produces the first two requirements for a garbage collector:

- safety: that the garbage collector will never reclaim a live object, and
- completeness: that the garbage collector will eventually reclaim every dead object.

Safety is clearly a necessary property since a collector that reclaims live objects will interfere with and likely break the computation it is meant to be supporting. Completeness is more interesting as it has been argued by some that "obscure" forms of garbage do not typically arise and where they do, that they will constitute only a small portion of the space consumed in the system; in fact it is possible to prove for a small subset of languages that such structures will never occur. For general purpose languages operating on arbitrary data structures, it is more difficult to argue the case against completeness since it transpires that the "obscure" form of garbage is formed by a very common data structure used in scientific and engineering applications: the finite element analysis mesh. Supporting the general case and these important applications requires completeness, therefore this thesis is concerned only with complete garbage collectors.

Figure 1 shows a very small but representative object graph with three kinds of objects: reachable (white), acyclic garbage (black) and cyclic garbage (grey). A

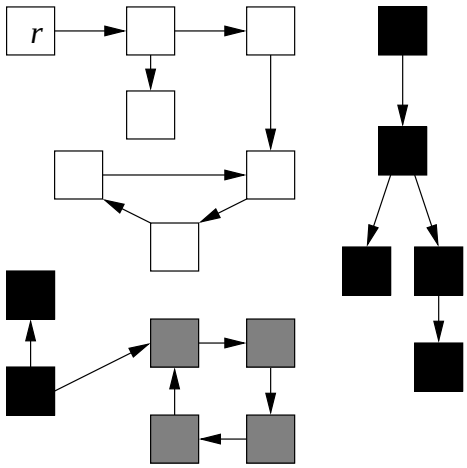


Figure 1: Reachability Examples

safe collector must detect no white objects as garbage and a complete collector will detect all grey and black objects as garbage. Incomplete collectors are typically incomplete with respect to cycles of garbage, i.e. they are unable to reclaim the grey objects.

### 1.1.2 Desirable Properties of a Garbage Collector

The primary benefit of garbage collection is reduced application complexity in conjunction with increased reliability: the determination of liveness is a difficult problem and a frequent cause of bugs in languages lacking GC. Clearly the determination of reachability inside the collector has non-zero cost and this cost should be minimised where it has any impact on application performance. The cost of GC however does not constitute a good argument against garbage collection in complex applications because the "manual" approach will reduce to the application programmer implementing an ad-hoc GC, e.g. by writing reference-counting smart pointer classes.

The desire for high performance garbage collection and minimal impact on application performance results in the following three additional desirable properties of a garbage collector:

- high throughput,
- good timeliness, and
- low induced latency.

Throughput refers to the rate at which the collector can detect and reclaim garbage; where throughput is lower than the garbage creation rate, the store will eventually fill with garbage, making the allocation of new objects impossible and

destroying the illusion of infinite space by preventing or deferring the allocation of memory by the application.

Timeliness refers to the delay between an object becoming garbage and its being detected as such by the collector; good timeliness means that the quantity of floating (undetected) garbage in the systems will be reduced, resulting in improved performance [56] and the ability to allocate more live objects before space is exhausted.

Induced latency refers to pauses induced in the computation due to garbage collection operation, e.g. a pause while the collector is executing or the temporary inability to access data or create objects due to collector activity. Poor induced latency is the typical argument presented against the use of garbage collection on performance grounds however garbage collectors have improved in this regard over the previous three decades with a move away from stop-the-world collectors to incremental and concurrent collectors. A stop-the world collector halts mutator[1] activity so that it may have exclusive access to the object graph to perform an entire phase of GC, incremental collectors interrupt mutators for shorter periods of time in which they perform some small increment of collection work and concurrent collectors are capable of operating without mutator interruption. Concurrent collectors are more difficult to construct than stop-the-world because they must be safe in the face of concurrent mutations to the graph.

Garbage collection has been an active area of research for approximately half a century, the term being coined by early LISP researchers [30]. Wilson [101] provides a good summary of the existing approaches to the problem as it applies to uniprocessor (non-distributed) systems and Jones & Lins [56] provide an overview of the field in textbook form. Reviewing the literature reveals the wide range of approaches to designing, describing and occasionally proving the correctness of garbage collectors; there is no single framework capable of describing the wide range of approaches taken in the literature to the detection of garbage.

## 1.2 Distribution

For the purposes of this thesis, a distributed system is one composed of a number of physically separate computational resources connected by some communications medium and cooperating on a single computation. The physical separation of the resources — **sites** — implies that there is non-zero latency in

[1]The mutator is the computation which operates over and mutates the object graph that the garbage collector is observing

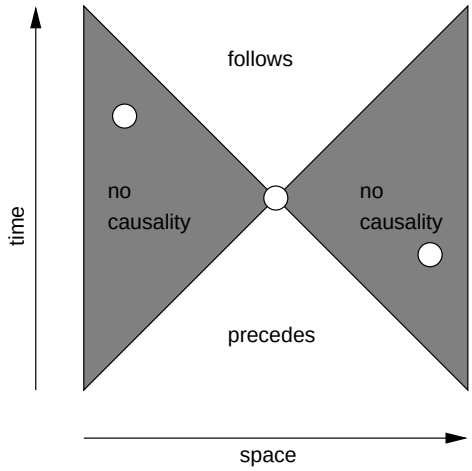


Figure 2: Relativistic Light Cone

communication between sites and that it is therefore impossible to have up-to-date knowledge of the global system state. This limitation is identical to the concept of a light cone in special relativity [37]: one event **precedes** another if it is possible for a photon to travel between the two events, likewise an event precedes another in a distributed system only if there was communication between the sites that the two events were located at. Relativistically, there is no causal relationship between events separated by more space than time and likewise there is no causal relationship between events at two sites if there exists no communication.

A light cone is illustrated in Figure 2 and the corresponding causality relationships in a distributed system are shown in Figure 3, in each case an event is denoted by a circle. In the distributed-systems case, message transmission is indicated by dotted lines and the vertical lines indicate the progression of computation at each site. The lack of causality in the grey areas with respect to the central event of these diagrams means that events occurring in those grey areas are not observable to the central event nor may they observe the central event, therefore it is never possible to have instantaneous knowledge of the global state of a distributed system.

Despite these observability difficulties, distributed systems may be used to solve computational problems where both the computation itself and the data that it operates over may be partitioned; each site within the system performs some part of the computation using local knowledge and then communicates with other sites in the system. The more work that may be done independently by a site, i.e. without communication or synchronisation with other sites, the more amenable the computation is to application in a distributed system and such computations are said to be **scalable**.

The benefit of running such readily distributed applications on a distributed

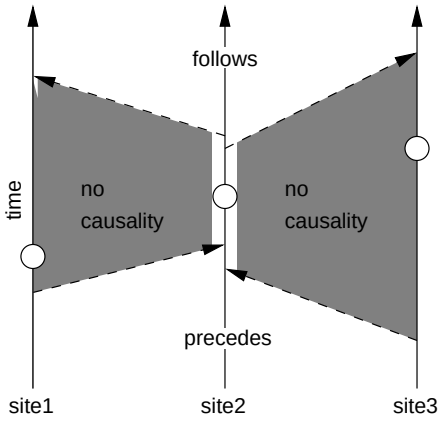


Figure 3: Causality in Distributed Systems

system is that much lower price per throughput is achievable at higher levels of throughput than with uniprocessor systems, hence the recent popularity of computational clusters using commodity hardware [90]. It is simpler and cheaper to scale the performance of a distributed system: one needs only to add new sites and networking infrastructure to the system and, assuming a suitably scalable application, the performance will increase. In contrast, the performance of a uniprocessor system may be increased only by replacing components of the system with faster instances and there are physical and financial limits to the performance available from a single processor.

A secondary benefit of a distributed system is the ability to build in redundancy and therefore fault tolerance [83, 88, 91]: computations may be designed in such a way that the loss of some computing resources will at worst reduce the performance of the computation, not the correctness. Such an approach is not possible on uniprocessor systems: the failure of the processor implies that the computation will halt and it is likely that data will be lost. By distributing a system physically across large distances, the system may gain immunity not only from random component failures but also from external disasters and/or malicious interference, the canonical example of such a system being the Internet.

## 1.3 Distributed Object Stores and GC

Given the performance scalability available from distributed systems, there is a need for programming techniques for such systems that are reliable, high performance and preferably not significantly more difficult to use than their uniprocessor counterparts. One approach to high performance distributed computation is to have the programming model represent the underlying computational structure, i.e. a collection of processes that communicate via the

transmission and reception of messages, e.g. MPI [44]. A drawback to this approach is that it is more difficult to achieve determination of global state than in a uniprocessor system because each site may not observe the state on any other site, they must perform all communications explicitly.

An approach that is friendlier to programmers is to implement a global address space, i.e. permit an application at any site to view any piece of data within the distributed system, though with constraints implied by consistency. There are many examples of such systems, taking a variety of approaches to decisions such as granularity of sharing [53, 54], naming [22, 38], replication [62], consistency [3, 58], update models [45], persistence [10, 8] and fault tolerance [7, 40].

More importantly, the existence of a distributed object store implies the necessity of a distributed garbage collector, i.e. a means to determine the reachability of objects given the presence of inter-site pointers. This is a difficult problem because unreachability is a property requiring knowledge of the global state and the nature of distributed systems makes it difficult to obtain a consistent view of this state. The lack of instantaneous global knowledge means that the feasibility of distributed garbage collection is dependent on the stability property of garbage; once an object becomes unreachable it is permissible to detect it as such some finite time later, by definition this latency will in the general case be non-zero due to the time required to propagate reachability information across the distributed system.

The cost of communication makes it desirable to make decisions locally where possible, i.e. without requiring interaction from other sites; the reason being that message round-trip times are typically large compared to processor cycle times, often by a factor of $10^5$ or more. The significance of synchronisation delays within a distributed system brings additional desirable properties for a distributed garbage collector:

- asynchrony or decoupling, and
- scalability.

Asynchrony and decoupling refer to the ability of the collector to perform work concurrently and without synchronisation. Specifically, performing collection work at one site may or may not require:

- synchronisation with the mutator, or
- synchronisation with collection effort at other sites.

Synchronisation with the mutator is particularly problematic since it may require mutator activity to halt for unacceptably long periods of time, resulting in poor induced latency. Synchronisation between collector instances is also undesirable because it reduces the throughput of the collector, making it unscalable; a system with an unscalable garbage collector cannot support scalable applications because the garbage creation rate will outstrip the detection and reclamation rate as the system grows.

Garbage collection scalability is a function both of the collector's ability to operate concurrently, i.e. without synchronisation and also its ability to operate without any communication with sites where such communication is not required. For example, it should be possible to reclaim a garbage component without communication with any site that the component does not span; a requirement for any such communication reduces the scalability and robustness of the system by introducing unnecessary dependencies between sites. Distributed collectors that achieve asynchrony and decoupling often interact with mutators in a similar way to concurrent uniprocessor collectors; in each case the assumption is that the graph is being mutated concurrently with collection and in the case of a distributed collector, those mutations may be occurring remotely.

A common technique for safely performing local work within a distributed garbage collector is to partition the graph into small pieces and introduce the **remembered set**: a data structure for recording the presence of inter-partition (including remote, or inter-site) pointers and thereby decoupling collection at one site from mutations at another. Such collectors exhibit the desirable properties of asynchrony and scalability but they are not complete because they use only local knowledge of connectivity. There exists an unavoidable tension between completeness on the one hand and asynchrony and scalability on the other because reachability is a distributed predicate: evaluating it exactly requires knowledge of state that is distributed across multiple sites. The evaluation only of local predicates on the object graph requires no communication and is by definition asynchronous and scalable but approaches using only local predicates cannot exactly detect reachability. For example, a collector constructed using only remembered sets will not be complete in the face of distributed cycles of garbage.

Asynchronous, scalable and complete garbage collection therefore requires some additional process for detecting unreachable regions of the graph: the process should be able to proceed without mutator interruption and terminate when such isolated regions are found. Likewise, the termination of that process should ideally be detectable without negative effects on the collector's scalability, i.e. with minimal synchronisation. Attempts at solving the distributed

GC problem have resulted in a large quantity of significant research with Jones' online garbage collection bibliography [55] referring to several thousand published articles.

There are a number of approaches to complete distributed garbage collection presented in the literature, with a wide range of complexity and performance. Some examples are distributed marking [57], controlled migration [67], back tracing [68], Hughes' algorithm [52] and the train algorithm [49, 51, 65, 64, 82]. Each of these systems has its benefits and its drawbacks and most have been implemented but each within different programming systems, making fair empirical comparisons of their performance difficult due to the requirement that test programs be translated between systems and due to other implementation details that differ between systems, such as the update model in use. Without implementing a number of collectors within a single programming system, an analytic approach to comparing the behaviour and performance of garbage collection algorithms would be fairer.

Because garbage collection is an exercise in graph labelling, the purpose of a GC is to gradually apply labels to objects according to their connectivity and at some point decide that the labelling has meaning, i.e. that one particular label is applied only to dead objects. This thesis defines the **essence** of a collector to be the means by which it makes progress in the application of labels to objects and also the means by which it decides when it has reached a labelling that accurately describes the presence of garbage. Abstract analysis of garbage collection algorithms require some means to describe the essence of each collector, preferably formally so that comparisons may be drawn.

Extracting the essence of a collector can be difficult because the published descriptions of some systems, notably DMOS [49], are highly mechanistic. For example, DMOS contains the train algorithm but also a mechanism for object substitution (migration support) and for the maintenance of train membership; separating this mechanism from the train algorithm for the purposes of comparing trains to other collectors such as Hughes' algorithm or back tracing is non-trivial. Namely, how does one decide where to draw a line through a collector and claim that certain components define the essential nature of the garbage collection algorithm in use and that other components are merely support mechanism that may be replaced without affecting the essence of the collector?

This thesis asserts that a rigorous way to compare GC algorithms is to use an **abstract model of garbage collection** that is capable of describing the essence of any particular garbage collector. The reason for this approach is that a model of garbage collection should contain an abstract description of the way in which

graphs are analysed to determine their connectivity (the "additional process" described above as being necessary for complete and scalable collection); if some part of a published garbage collector can be described by the abstract model, then it may be said that that component is the essence of the collection algorithm.

Comparing GC algorithms via their essence can reveal features such as correctness, time-complexity of operation and minimal requirements for synchronisation but because the comparison does not take into account the implementation details, specific measures of performance (e.g. throughput, timeliness, space overheads, etc) are not available. For example, mark-sweep and semi-space copying collectors [101] both make progress by a forward trace of the object graph from the root but the specific mechanisms for allocating objects and representing free space differs, resulting in differences in real-world performance, particularly with respect to space overheads. Therefore mark-sweep and semi-space-copy arguably are both the same essential garbage collection algorithm and they differ only in the mechanisms of their implementation. While the implementation differences between these collectors appear significant on the surface, they are both $O(live\,count)$, both apply one of two labels to each object, both traverse the live object graph in a wave from the root, both may use snapshot-at-the-beginning for concurrent operation, both operate in distinct phases and both require global synchronisation at a phase change.

A model that is capable of describing a large number of collection algorithms is defined here as **broad**, while a model that describes an algorithm in detail, e.g. containing a proof of correctness, is defined as **specific**. While it is possible that some models may merely be only broad or only specific, a model that is both broad and specific may be used to perform a detailed analytic comparison of multiple garbage collection algorithms.

## 1.4 Models of (Distributed) Garbage Collection

Bacon and Rajan [13] discovered similarities between forward-tracing and reference-counting uniprocessor collectors, noting that optimised versions of each collector behave similarly and have similar performance traits because they seem to be composed of the same underlying tracing actions; the approach is applicable only for uniprocessor systems. Their observation is not so much a model, but implies that there may exist a single model that is sufficiently broad to describe both forward tracing and reference-counting collectors.

Abdullahi and Ringwood [2] present a survey of distributed garbage collection as of 1998 which, in listing a brief description of a large number of collectors, highlights the fractured nature of descriptions of distributed garbage

collectors; each collector is described in its own terms, with different assumptions as to the nature of the underlying system and other implementation details.

Some research has been performed with respect to building generic models of distributed garbage collection, e.g. Lowry [64] defines a formal model of the Train Algorithm using wave-based Distributed Termination Detection Algorithms (DTDAs), Norcross [82] describes the construction of Train-based collectors by the composition of DTDAs and arbitrary local collectors and Zigman [102] describes the creation of compound collectors by composing multiple collectors to operate on sub-graphs. These approaches all provide models of modern, high-performance (asynchronous, decoupled and scalable) approaches to distributed garbage collection, the limiting factor being that the most formal — and therefore specific — literature is relevant only to Train-like approaches to garbage collection, i.e. it is not broad. Likewise, the broadest model (Zigman's) is not very specific.

Moreau *et al* [74] present a formal proof of Birrell's Reference Listing [18, 19] by introducing a graphical representation of the algorithm's state space and permitted transitions therein. This proof will become more valuable as a model of GC if it can be extended to describe the whole of a complete garbage collector but it is not yet clear how this will be achieved, i.e. the model is highly specific but not broad. The proof also seems to be entirely manually derived, i.e. there is no mechanical process that will permit another proof to be derived for a different collector. Tel and Mattern [94] present an alternative approach to proving the completeness of pointer tracking algorithms by showing that the issue is equivalent to Distributed Termination Detection (DTD) [93], a well-understood problem.

Tel's "proof by association" seems a better approach because it does not require the manual derivation of all states within a system and the discovery of order and meaning within those states, rather it uses existing proofs of well-known DTD algorithms to prove the correctness of a collector by showing congruence between the collector and a solution to the DTD problem. The limitation to this approach so far is that it is capable only of describing collectors that detect trivial unreachability (lack of pointers directly to an object) and are therefore incomplete. In other words, Tel's model of GC formalises the approach wherein collection decisions are made using only local knowledge of graph connectivity; the model contains no process to discover distributed regions of garbage.

The current state of the art therefore is that there are a number of known distributed garbage collectors, each not necessarily having a rigorous proof of correctness. Currently, each collector must be analysed and understood on its

own terms and it is a non-trivial task to invent new garbage collectors by building on the knowledge gained from previous efforts. While it is possible to manually prove the correctness of collectors by traditional means (e.g. the proof of Birrell's reference counting above), a more automated approach is desirable, i.e. the use of an abstract model.

An ideal model should be broad, specific and formal: it should be possible to instantiate a wide range of garbage collectors, prove their correctness using the model and perform comparative analysis between all collectors that may be described by the model. Abstract models of GC exist and are discussed in the following chapter but none exhibits the combination of all desirable properties.

## 1.5 Contributions & Structure of Thesis

The primary contribution of this thesis is that it presents the **Surf** abstract model of distributed garbage collection that is both general enough to describe a wide range of collectors and specific enough that its proof may be applied to collectors instantiated from the model. Due to the detail present in the model, describing a collector in terms of the model provides insight into some properties of the collector: where synchronisation is required for safety and how much this impacts concurrency, the expected scalability of the means by which the collector makes progress and/or detects isolation and broad generalisations regarding the collector's computing overhead and expected timeliness. A proof of correctness for Surf is supplied that defines a list of requirements that an implementation must fulfil and these requirements are the means by which this analysis occurs. The Surf model provides a design space within which each collector represents a single point; the process of choosing a point in the design space and fulfilling the requirements for completeness constitutes a concrete instantiation process. The existence of a well defined process that a designer can follow means that the creation of correct new garbage collectors is now more mechanistic and less error-prone than it was previously: by analogy, garbage collectors deny programmers the opportunity to incorrectly free memory and a model of garbage collection makes it difficult to design an incorrect collector.

The limitations of the Surf model arise primarily from the system model within which it is defined: sites and communications are assumed to be reliable (no crashing or restarting, no byzantine behaviour, no packet loss or corruption) and the collection process defined by the model is valid only for a shared-nothing storage model with no object migration. The model in its current state is therefore incapable of describing fault-tolerant systems because it does not permit the replication of objects and therefore has no model of coherency. The formality

of the model requires that it be aware of every pointer, therefore it is applicable only to closed systems: no federated stores [79], persistent [10, 8] or otherwise open systems may be described by the model without extending it. Use of the Surf model to design a new collector is no guarantee that that collector will exhibit high performance because the model is capable of describing collectors with a broad range of performance characteristics. A related limitation is that the model describes only the essence of a collection algorithm, i.e. the means by which the collector makes progress in labelling objects and deciding when the labelling implies the presence of garbage. An implementation contains not only this essential garbage collection algorithm but also a number of support mechanisms which may have significant effects on the collector's performance.

The thesis begins by examining the published history of distributed garbage collectors and models thereof in Chapter 2. The result of this examination is the discovery of a need for models of distributed garbage collection that are both general and formal; existing models are either general or formal but not both.

Having identified the need for a new model of distributed GC, Chapter 3 provides a formal definition of a distributed system using an event/transition model. The safety and completeness requirements of a collector are then defined in terms of the event/transition model and reachability. The system model is used to define all the components of the Surf abstract model of distributed GC then a proof of correctness is built on the system model and a mapping between the Distributed Termination Detection [93] abstraction and the isolation of graph regions. A summary of the model is provided as a definition of the design-space that the model represents and a list of requirements that a collector must fulfil for the model's proof to apply; this design space and list of requirements constitute a concrete process for analysing the performance of collectors and instantiating new collectors.

To verify the descriptive power of the model and its accuracy, a number of existing garbage collection algorithms are described in Chapter 4 in terms of the Surf model by applying the analysis process from the previous chapter. For each collector so analysed, some behaviour is predicted and in each case, this behaviour matches what is already known about these collectors. Demonstrating that the Surf model's predictions match reality is the first step in validating the Surf model and the accuracy of its analytical power; admittedly these predictions are post-hoc but by definition they must be if they are to be compared with the known properties of existing collectors.

The next step in validating Surf's analytical power is to make entirely new and untested predictions and test those predictions empirically. Where progress of collection is tied by performance-driven implementation choices to garbage

collection activity within partitions, the partition selection policy is critical to enabling progress. The Surf model of progress is therefore applied in Chapter 5 to a distributed persistent garbage collector that is an instance of the train algorithm; Surf provides insight into where progress is available and this information is used to define the partition selection policy. The Surf model predicts that with the chosen policy, approximately linear complexity (collector invocations required to reach completeness) is attainable, in contrast to the quadratic seen with naive policies previously published or the lack of progress seen with policies designed to detect acyclic garbage. The predictions of linear performance are then confirmed in experimentation using an implementation constructed for the purpose of measuring and analysing collector performance; this collector is also believed to be the first implementation of the train algorithm in a distributed persistent store. Having made new predictions and verified them experimentally, the Surf model's predictive value is thereby further validated.

The Surf model provides a design space within which collectors may be instantiated; previously published and implemented collectors that fit within the model (explored in Chapters 4 & 5) are heavily biased to one corner of the design space. In an attempt to explore the as-yet-unknown regions of design space, Chapter 6 describes a new GC algorithm that the Surf model predicts to have a behaviour unlike any other published collector: the ability to detect topology in the object graph other than disconnectedness and thereby detect garbage with excellent timeliness in certain circumstances.

In conclusion, abstract formal models of distributed garbage collection are a rigorous way of designing, proving, analysing and comparing distributed garbage collection algorithms. This thesis presents such a model, proves its correctness and verifies its predictions against the known behaviour of existing collectors and a new implementation. Exploration of the model's design space results in predictions of the existence of a new class of garbage collection algorithms.

# Chapter 2

# Distributed Storage Management

This thesis is primarily concerned with the presentation of a new abstract model of garbage collection, therefore this literature review chapter presents an overview of existing techniques for, and abstract models of, garbage collection.

Fundamental to any discussion of distributed garbage collection is a system model and a formalised language for describing the operations that may occur within that system; such a model is introduced here so that it may be used to describe models of garbage collection throughout this thesis.

Unreachability of an object is a stable property, i.e. once an object becomes garbage it cannot become live again. The Distributed Termination Detection (DTD) abstraction is a formal description of a class of computation over which stable properties may be detected and a DTD Algorithm (DTDA) is a solution to the DTD problem, i.e. a DTDA is a concrete algorithm that is capable of detecting stable properties. The DTD abstraction is therefore critical to distributed garbage collection because the fundamental requirement of a garbage collector is to detect a stable property of the object graph. For this reason, DTD is introduced in this chapter and two approaches to the description of DTD Algorithms are presented. DTD Algorithms are a core component of some existing models of distributed garbage collection and likewise they are a core component of the Surf model presented in this thesis.

Individual garbage collection techniques as represented by published collectors are reviewed here, leading to the conclusion that models of garbage collection are desirable because they provide a framework within which collectors can be constructed, proven and compared. The literature contains incorrect published collectors (errors noted in [42, 43, 87, 102]), in each case it seems the algorithms are published as mechanistic descriptions of implementations. A more rigorous approach to the design of collectors is via formal models of garbage collection. Two existing models are presented in this chapter — the Train Algorithm and Hierarchical Collectors — and each is

analysed in light of how well it achieves the desirable properties of an abstract model of garbage collection, namely:

- the model should have a proof of safety and completeness that may be applied to collectors instantiated therefrom,
- the model should be capable of describing a wide range of garbage collectors,
- the instantiation process should be as mechanistic as possible, leading to rigour in the application of the correctness proof, i.e.
  - the model should define a design space within which each instantiation represents a point,
  - the model should provide a concrete list of constraints that the instantiation must fulfil in order to satisfy the proof,
- the model should provide analytical insight into the behaviour of its instantiated collectors, and
- the model should be easy to understand and analyse.

Note that there are no properties here relating to the performance of individual collectors such as throughput or asynchrony; such details will depend on the details of a particular instantiation. Predictions regarding collector performance are one form of insight that it is desirable a model may provide into individual collectors.

## 2.1 Distributed Computation

Computation in a distributed system may be represented in a number of ways but a common factor to all representations is that they can express concurrency in the system, a necessary property due to the concurrency inherent in having multiple computing resources present in the system. Examples of such methods include Communicating Sequential Processes (CSP) [47] that use synchronous message passing and no shared storage, Distributed Processes [46] use asynchronous message passing and have no shared storage while Synchronizing Resources [5] use shared resources (storage). Java [6, 41] permits communication by remote procedure call (RPC) [100].

For the purposes of describing distributed algorithms within this thesis, an approach is taken similar to that of Distributed Processes. Each site executes at an arbitrary rate and communication is by the transmission and receipt of

asynchronous messages. The model is described in detail in Section 3.1 and summarised here as follows:

- computation at each site is represented as a chain of **states** that a site assumes in sequence,
- the atomic transition between two states is referred to as an **event,**
- events may occur at an arbitrary rate, with no bound on relative execution rates between sites,
- the transmission of a message to some other site is an event,
- the receipt of a message is an event, and
- causality exists only between subsequent events at the same site and between the send/receive event pair of each message.

The lack of shared storage and the strong definition of causality in this model make it a lowest common denominator approach to describing distributed systems.

This computation model and further discussion in this thesis do not include any concept of fault tolerance, i.e. communications and sites are assumed to be reliable. Packets are never lost, sites do not crash or restart and do not exhibit Byzantine behaviour.

## 2.2 Distributed Termination Detection

The Distributed Termination Detection (DTD) problem [39, 36] is fundamental to distributed computation because solutions to it may be adapted to the detection of **stable** properties within a computation, for example detecting that a system has deadlocked or that some region of a graph has become unreachable. A stable property [86] is a condition within a distributed system wherein some predicate evaluated over the states on every site within the system is true and will remain so regardless of the events that the computation may perform in the future. Clearly it is possible to safely evaluate only stable properties within a distributed system due to the lack of instantaneous knowledge; a global predicate that is true only momentarily may not necessarily be detected and if detected, may be detected after it is no longer true.

The **classic model** of the DTD problem concerns a distributed computation and the detection of its termination with the following assumptions as to its nature:

1. a site may be **active** or **passive**,
2. only active sites may transmit messages,
3. an active site may spontaneously become passive, and
4. a passive site may become active only on receipt of a message.

The system is considered **terminated** when every site is passive and there are no messages in flight. Termination is a stable property: rule 2 ensures that no messages may be sent and rule 4 ensures that no site may become active.

A DTD Algorithm (DTDA) is a distributed computation that observes some other distributed computation with the aim of detecting when it has terminated; surveys and classifications of such algorithms are presented by Tel [93] and Matocha & Camp [70].

Livesey *et al* [63] recently presented Doomsday, which contains two distinct concepts: the **Doomsday model** of DTD and the **Doomsday protocol** for constructing DTDAs. The **Doomsday model** of DTD expresses the problem slightly differently from the classic model:

1. A **task** is an active computation.
2. A task may migrate between sites.
3. A new task may be created at a site if it is **witnessed** by an existing task.
4. A task may spontaneously die and cease to exist.
5. A **job** is a collection of tasks with its origin in the birth of one or more tasks at a nominated home site.

Under Doomsday, the purpose of a DTDA is to detect termination of a job, i.e the lack of existence of any tasks within that job.

It is shown later in this thesis that the two models of DTD are substantially identical with the exception that under the Doomsday model, the nominated home site may spontaneously create tasks, therefore a Doomsday DTDA may detect termination only at the home site while classic DTDAs may safely detect termination at any site. The Doomsday system model is presented here because its job/task notation can describe the application of multiple instances of a DTD algorithm within a single computation, i.e. it explicitly permits the observation of finer-grained stable states than termination of an entire computation. The Doomsday system model notation is used for reasons of clarity later in this thesis without implying that the Doomsday protocol must be used to implement the DTDAs.

DTDAs may be broadly classified in two dimensions: synchronous or asynchronous, and wave-like or Doomsday.

A synchronous DTDA is one that requires message transmission to be synchronous, i.e. sites may perform no computation during the time in which a message is in transit or, alternatively phrased, message delivery is instantaneous; such a DTDA will not function correctly given the system model of Section 2.1. An asynchronous DTDA is one which makes no assumptions regarding message delivery times with respect to computation rates, i.e. messages may be arbitrarily delayed though some DTDAs require FIFO (in-order) delivery of messages on a given channel.

A wave-like algorithm is one which polls the system with a wave, visiting all sites and determining whether termination has been reached. A Doomsday algorithm using the Doomsday protocol of [63], i.e. correct operation requires only that task birth is **witnessed** by an existing task at the same site and that birth and death messages are delivered to a home site where termination is be decided. The Doomsday protocol appears to unify all non-wave-like DTDAs.

DTD may be extended from the detection of computation termination to the detection of arbitrary stable properties merely by a change in terminology as long as the same underlying constraints on the computation are observed, i.e. the property being detected is stable due to the same constraints that make termination a stable property. For example, Chandy & Misra [73] show that deadlock detection is congruent with termination detection and therefore that a DTDA may be used to implement deadlock detection.

Because unreachability is a stable property, the Surf model of distributed garbage collection presented in this thesis relies heavily on the DTD abstraction, as does much of the existing garbage collection literature as described below.

## 2.3 Distributed Garbage Collection

Garbage collection is the process of detecting the reachability of an object from a nominated **root**. If an object is directly or indirectly reachable, it is considered live, otherwise it is garbage and may be reclaimed. The garbage collection problem is well understood in the uniprocessor context where instantaneous knowledge of the entire object graph is available; see Wilson [101] for a survey of techniques.

A distributed store implementing a single address space implies both that there are objects at multiple sites and that they can contain pointers to arbitrary other objects, including objects at remote sites, in turn implying the presence of inter-site pointers. The presence of inter-site pointers is what makes the creation

of a complete and scalable collector difficult: the garbage collector must have global knowledge of the state of the graph but must be able to make progress in the detection of garbage without synchronising with other sites or interrupting the mutator of the graph that it is observing.

The distributed GC problem is soluble because unreachability is a stable property and may therefore be detected in a distributed system via use of DTD algorithms.

The problem with much existing literature is that garbage collectors are constructed in an ad-hoc fashion from designs that contain elements specific to particular applications and without a clear view as to how the correctness of the design may be proved, if it is in fact correct at all. For example, the original DMOS [49] publication includes an optimisation by combining multiple pointer tracking messages into a single message instead of merely presenting the pointer tracking algorithm as implemented and proving its correctness. It is therefore desirable to have a **model of garbage collection** that describes clearly how a class of collectors operates and proves their correctness. Creation of a new collector should therefore occur as an instantiation of the model, reusing the model's proof of correctness by showing how the new instantiation conforms to the constraints specified by the model.

By making the construction of a collector a mechanistic process that starts from a model, it may be possible to reduce the probability of the resulting collector being incorrect in some way. Conversely, it is undesirable to restrict the design of collectors more than necessary, i.e. permitting a model to instantiate a wider range of collection algorithms is an improvement to that model if rigour is not compromised.

Jones & Lins [56] and Abdullahi & Ringwood [2] review a number of approaches to distributed garbage collection. Uniprocessor garbage collection is not considered here in detail except where necessary in the application of a particular model or approach.

This section proceeds by describing approaches to detecting **trivial unreachability** (the lack of pointers to an object), culminating in its equivalence to the DTD problem. The review then moves onto globally complete collectors, presenting a range of published approaches and the issues that they have been found to face; previously these issues were thought to be specific to certain collectors but later chapters of this thesis will generalise the problems across a wide range of collection approaches and give specific reasons as to exactly why each collector does or does not suffer from each specific problem. Finally, hybrid and compound collectors are presented, each being composed of two or more other collectors for performance reasons; for example, a hybrid may

contain both a high performance collector using only local information (e.g. reference counting) to reclaim acyclic garbage and a complete collector with poorer timeliness to detect distributed cyclic garbage.

The diversity of the literature described here leads to difficulty in applying lessons learned from the design of prior algorithms to the design of newer algorithms therefore mistakes and discoveries are repeated and wheels are reinvented.

### 2.3.1 Trivial Unreachability

Trivial unreachability is defined as the lack of pointers to a particular object; the detection of this circumstance via reference counting [30] constituted the first attempt at garbage collection in a uniprocessor system. In a distributed context, the class of algorithms that detect trivial unreachability are referred to collectively as **pointer tracking algorithms**.

The nature of pointer tracking, wherein metadata (the **remembered set**[1]) is attached to an object and updated on receipt of messages from other sites where pointer creations and deletions have occurred, leads to a collector with good decoupling and scalability, though care must be taken with correctness where a pointer is transmitted between two remote sites. A single integer counter with simple increment/decrement messages at pointer creation/destruction events is not correct, as illustrated in Figure 4 where a naive and incorrect implementation of reference counting is shown. Site $A$ contains a pointer to an object on site $H$, $A$ sends a copy of the pointer to $B$ and then erases its local copy. There is no way to guarantee the ordering of increment and decrement message arrival in this implementation, therefore it is possible that the reference count will unsafely reach zero as shown in Figure 4. Though it seems that a simple change such as having the transmitting site send an increment instead of the receiving site, Figure 5 shows that this also is unsafe.

The lack of safety is inherent in the fact that increment and decrement messages arrive from different sites and therefore an ordering on their transmission order does not imply an ordering on their receipt order. Lermen and Maurer [60] rearrange the pattern of message passing to ensure safety through the FIFO properties of channels used in their system; Birrell's Algorithm [18] separately lists each remote reference to each object; Watson & Watson's weighted reference counting [99] permits each reference to hold a different weight and divides that weight when the reference is duplicated; DMOS [49] uses vectorised reference counts in its pointer tracking algorithm [50]. In proving Birrell's

[1]The term "remembered set" is used here to describe any form of metadata used by a pointer tracking algorithm in the determination of trivial unreachability.

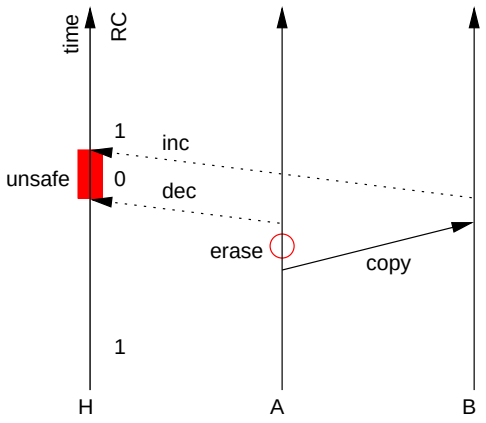


Figure 4: Erroneous Reference Count

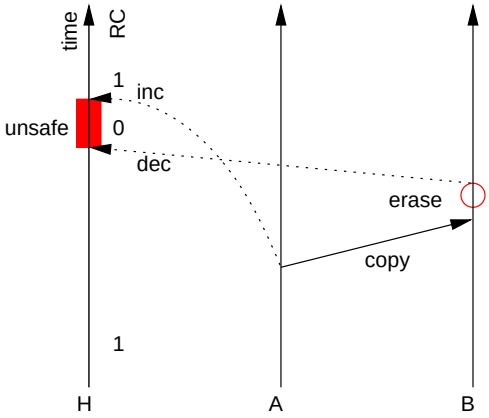


Figure 5: Erroneous Reference Count

Algorithm, Moreau *et al* [74] show diagrams of a number of published pointer tracking algorithms, each of which takes a different approach to ensuring safety.

Trivial unreachability is a stable property: once there exist no pointers to an object, no pointers to that object may exist in the future. Tel and Mattern [94] showed the equivalence of the pointer tracking and DTD abstractions, which means that a solution to one problem is a solution to the other and therefore that a correct DTDA is a solution to the pointer tracking problem. A garbage collector constructed using only a pointer tracking algorithm is incomplete with respect to cyclic garbage; though the whole cycle is unreachable from the root, no object in the cycle will be trivially unreachable.

The equivalence of pointer tracking and DTD is the path by which Doomsday came to existence: the DMOS pointer tracking algorithm was described first, evolved into the Task Balancing DTDA and then generalised into the Doomsday model of DTD, of which Task Balancing is one instance. Likewise, weighted reference counting is identical to the credit-recovery DTDA [71]: they are the same algorithm and solve the same problem but were described separately in different contexts.

### 2.3.2 Globally Complete Collection

Complete distributed garbage collectors may be broadly classified according to how they detect garbage:

- forward tracing algorithms,
- backward tracing algorithms, and
- region isolation-detecting algorithms.

Each of these approaches are elaborated below.

#### 2.3.2.1 Forward Tracing Collectors

The forward-tracing group of collectors are akin to the mark-sweep approach as seen in uniprocessor systems [72, 89, 17, 35]: the collector operates in phases, incrementally marking objects according to their reachability and starting at the root. At the end of a phase, all reachable objects are marked and the unmarked objects have been detected as garbage. Distributed collectors taking this approach include those of Hudak & Keller [48], Augusteijn [11] and Derbyshire [33]. This approach has scalability and timeliness issues since the marking wave must traverse the entire distributed heap before any garbage may be reclaimed.

Hughes [52] extends the distributed concurrent marking approach to permit multiple concurrent waves acting in pipeline fashion and in what seems a first for distributed garbage collection, makes explicit use of a distributed termination detection algorithm (Rana's [84]) to detect termination of each wave. Once a wave has terminated, every object marked with a wave number (epoch) lower than the terminated wave is detected as garbage. Hughes' collector is therefore the first algorithm to explicitly map the detection of unreachability (a distributed stable property) onto the DTD problem and thereby apply a DTDA to detect the unreachability of regions in the graph.

A forward tracing collector typically obtains safety in its interaction with concurrent mutators using a policy known as **snapshot at the beginning** (SATB): every object that is the target of an erased pointer will be marked and therefore considered live for the current mark phase. SATB is named as such because the policy results in every object that is reachable at the beginning of a phase not being reclaimed during that phase regardless of its reachability at the end of the phase. SATB implies that every pointer in the system constitutes marking work, even after the pointer is erased.

#### 2.3.2.2 Backward Tracing Collectors

Backward tracing collectors perform actions similar to forward tracing collectors in that they mark objects according to their connectivity, however the direction that the marking wave takes is reversed. At each step, the mark bit will progress from object $x$ to objects $y$ and $z$ if there exist pointers from $y$ and $z$ to $x$. The wave begins at a **suspected** object and progresses until no further progress is available or the root object is reached; in the former case, all marked objects have been detected as garbage.

An example of such a collector is Thor's completeness mechanism [68] which has the stated design goal of increasing locality of collection, it does so by attempting to constrain traversals to unreachable regions and thereby obviate the need for participation of sites not spanned by the garbage component being detected.

Thor further constrains the tracing region by having multiple levels of suspicion based on an estimate of distance [67, 66] from the root; objects closer than a certain threshold are assumed live with the aim of preventing the back trace repeatedly traversing live regions of the graph.

#### 2.3.2.3 Region Isolation-Detection: the Train Algorithm

MOS [51] (Mature Object Space) is a uniprocessor GC which divides the object graph into regions referred to as **trains** using what it refers to as a reassociation policy; progress of the collector is dependent on the correct operation of the reassociation policy to move objects between trains according to their connectivity. When there exists no inter-region pointers to a particular train, it is isolated and therefore detected as garbage. The MOS algorithm and its derivatives — the MOS family of collectors — are collectively referred to as the Train Algorithm.

PMOS [78, 81] extends the Train Algorithm for use in a uniprocessor orthogonally persistent [10, 8, 9, 28, 76] system by making the collector aware of hard-drive related input/output costs, specifically the high access latency.

DMOS [49] extends the Train Algorithm for use in a distributed store by the addition of a pointer tracking algorithm, Task Balancing [50], and the use of a DTDA to detect train isolation where trains span multiple sites, making it the second distributed GC to make explicit use of DTD to detect unreachability. Lowry [64] and Norcross [82] have subsequently provided more formal definitions of distributed instances of the train algorithm so the term DMOS is used in this thesis to refer to the original 1997 publication.

#### 2.3.2.4 Operation of DMOS

DMOS [49] partitions the object space into cars and trains; each object is entirely contained within a car and each car is entirely contained within a site and a train. Trains may span sites and their isolation is the mechanism whereby distributed garbage is detected and reclaimed. Metadata (remembered sets) is maintained to track the existence of inter-car pointers and a pointer tracking algorithm operates to ensure that this metadata is maintained safely. The means by which the collector makes progress is two-fold:

- acyclic garbage is reclaimed via repeated operation of a partition (car) collector at each site, and
- objects are reassociated between trains so that each cycle of garbage is collapsed into a train and all live objects are removed from that train.

The typical, though not necessary, arrangement is that the partition collector is a copying collector; while it is operating, any object in that partition is a candidate for reassociation to other trains. The reason for this is convenience: object reassociation policies are typically based on connectivity, i.e. objects are reassociated into a train from which they are reachable, and the operation of the partition collector is an efficient means to determine the set of objects within a partition that are reachable from some other train.

Object reassociation does not imply that objects migrate between sites[2], only that they become contained within a different train. Because objects are contained within cars and cars contained within trains, DMOS implementations use a copying partition collector that evacuates a car by copying every object therein to other cars, whether they be in the same or some other train; the space occupied by the evacuated car is reclaimed. To prevent infinite reassociation of garbage objects, trains are strongly ordered and objects may be reassociated only in one direction with respect to the ordering on trains.

Protocols are provided to permit sites to join a train when they have an object that the reassociation policy indicates should be reassociated to that particular train, and for sites to leave a train when they contain no remaining cars in that train. A DTD job exists for each train, with tasks of that job being objects within the train that are reachable from other trains. When the job is detected as terminated, it contains no tasks and therefore the train is isolated from all other trains and may be reclaimed.

Lowry [64] showed that cars are not necessary for the train algorithm; they exist only as a convenient implementation technique to hybridise trains with

[2]No known distributed train implementations support object migration though the published algorithm supports it.

a partition collector and thereby rapidly reclaim acyclic garbage. The train algorithm requires only that objects exist inside trains and are reassociated between trains where necessary to make progress. Likewise, the original DMOS publication specifies the collector to the point where trains exist in rings and ring-specific protocols are defined for train management; trains in rings are an example of implementation mechanism defined by DMOS that is not essential to the distributed train algorithm.

#### 2.3.2.5 Issues with Distributed Trains

Applying trains in a distributed context raises a number of correctness issues that have been discovered over time:

- various **race conditions** [64, 102], relating to concurrent reassociation, train maintenance and train isolation detection,
- the **unwanted relative problem** [49], related to isolated trains not remaining isolated, and
- a no-progress bug [87] wherein concurrent mutator activity may hide a reference from the reassociation process.

Race conditions between train joining/leaving and isolation detection exist in the DMOS paper and were discovered by Zigman [102]; the problem is that the DTD model constraints are not observed where train membership is being updated.

The unwanted relative problem occurs where the reassociation policy imports a live object into an unreachable train, thereby making the train reachable again. The original DMOS paper recognises the unwanted relative problem and solves it by introducing the concept of an **epoch**, i.e. the partitioning of a train by age of objects. Epochs are delimited by a consistent cut obtained by passing a token around all member sites of a train, which are arranged in a ring; when isolation is detected, it is detected per epoch. This solution means that the isolation detection DTD is running on a non-growing subset of the train and isolation of that subset is stable; the solution is similar to that proposed by Lowry [64] where a train is closed to all reassociation and object creation while the DTDA operates.

The unwanted relative race was described in more detail and more formally by both Lowry and Norcross, each solving the problem by requiring synchronisation between the reassociation process and train isolation DTDA. Both Lowry and Norcross define models of distributed garbage collection using the train algorithm and their contributions are discussed in more detail below; solving this particular race condition requires that reassociation fits the constraints of task creation within the region isolation DTDA.

The progress bug was discovered by Seligmann and Grarup [87] in the original MOS [51] and is due to the collector using discrete sampling of remembered sets at each site to discover inter-train pointers and therefore work for the reassociation process. Consider the case where train $A$ is reachable from train $B$ but the object in $A$ that is directly reachable is rapidly changing due to mutator activity in $B$; the sampling of remembered sets at different sites in $A$ means that there is no guarantee that any site in $A$ will observe the presence of the pointer. Discrete sampling means that the collector is aware of the presence of work at a given site only at certain points in time; if that work moves amongst the sites of a train due to mutator activity, it is possible that no site will observe the work and act upon it. The system remains safe because the pointer tracking algorithm (a DTDA job) for each object is safe and correctly integrated with the region isolation DTDA, but the collector may make no progress.

A solution to the progress bug is to consider pointers to represent reassociation work in certain circumstances even after those pointers have been erased. The extreme case is to consider all erased pointers to represent reassociation work, an approach that is equivalent to snapshot at the beginning both in its implementation and in its effect: all objects that were ever reachable from some other train will be reassociated out of their current train. A more constrained solution that considers fewer erased pointers as reassociation work is presented by [87] but it places undesirable constraints on other parts of the system, namely the partition selection policy. DMOS proposes the use of a **sticky bit** that represents reassociation work after a pointer has been erased.

While the train maintenance race conditions are specific to DMOS because they are an artefact of the train maintenance protocols, the progress bug and unwanted relative problem do not seem specific to the train algorithm: they relate to how progress is found and executed and how the isolation of regions is detected, fundamental operations in any garbage collector. This raises the question of whether other distributed garbage collectors may or may not be subject to these problems and if not, what is the critical difference between those other collectors and the train algorithm that makes them immune from these problems that have been observed with distributed trains? It is therefore desirable that these problems be predicted by a formal model of GC when instantiated with the train algorithm; inspection of the difference between the train instantiation and other instantiations should provide insight into why the train algorithm is subject to these issues and what properties of other collectors make them immune (or not) to the issues described in this subsection.

### 2.3.3 Hybrid and Compound Collectors

Hybrid collectors are an approach whereby multiple garbage collection algorithms are combined to produce a collector exhibiting the better properties of all the collectors from which it was formed.

Generational GC [61, 97] is a technique for composing multiple collectors in a uniprocessor system and frequently inspecting the newest region of the graph in the assumption that the probability of an object becoming garbage within any particular timespan reduces with the object's age. It is a form of incremental GC in that only a small portion of the store is inspected during each collector invocation; MOS [51] was first described as an algorithm for performing incremental GC within the (relatively large) mature space, therefore the original application of the Train Algorithm was as one GC within a system that contained at least two. This concept is extended by Beltway [21] for uniprocessor systems, a mechanism for dividing the heap by object age and performing incremental collection with high performance.

An early example of hybrid collection in a distributed system is [59], which combines pointer tracking with a distributed tracing algorithm; the result is a collector which will rapidly detect acyclic garbage due to the use of reference counting and occasionally reclaims cyclic garbage by invoking the tracing collector. In a similar vein, [85] combine Birrell's reference listing [18] with Derbyshire's distributed marking [33].

Implementations of the Train Algorithm [24, 78, 82, 87] generally contain two collectors: the Train Algorithm itself in addition to a copying partition collector. Each invocation of the partition collector copies all objects in a partition that are not trivially unreachable to some other partition; where the reassociation policy indicates that an object should be moved to another train, it is at this time that it occurs. The use of a pointer tracking algorithm to generate remembered sets for each partition means that acyclic garbage will be reclaimed by the partition collector; the Train Algorithm is necessary only to reclaim inter-partition (including distributed) cyclic garbage.

Zigman [102] describes a method for composing hierarchical garbage collectors by dividing the graph into regions, applying a separate collector within each region and specifying requirements for how objects must migrate between regions so that a higher-level collector may detect cycles of garbage. The resulting model encompasses the Train Algorithm and a variety of other collectors, permitting arbitrarily deep nesting of collectors; it is described in more detail below.

## 2.4 Models of Distributed Garbage Collection

A number of distributed garbage collectors are listed above, employing a wide range of approaches that seemingly do not fit into any one single model, though there exist some recently defined models which each encompass some part of the literature. The term "model" is used here to represent formalisms which encompass a range of garbage collectors or even approaches to garbage collection; each typically provides a **design space** within which specific algorithms exist. An **instantiation** of a model is a specification of all the parameters of the model's design space, sufficient to describe a particular garbage collection algorithm. A number of existing models are summarised here; the models of Norcross, Lowry and Zigman are explored in detail below.

The objectives of a model of garbage collection, restated, are:

- the model should have a proof of safety and completeness that may be applied to collectors instantiated therefrom,
- the model should be capable of describing a wide range of garbage collectors,
- the instantiation process should be as mechanistic as possible, leading to rigour in the application of the correctness proof, i.e.
  - the model should define a design space within which each instantiation represents a point,
  - the model should provide a concrete list of constraints that the instantiation must fulfil in order to satisfy the proof,
- the model should provide analytical insight into the behaviour of its instantiated collectors, and
- the model should be easy to understand and analyse.

The models analysed in this section vary in their fulfilment of each of these objectives and are analysed in terms of such.

The first model is the equivalence of DTD and pointer tracking as shown by Tel and Mattern [94]; it is important because it shows that a particular garbage collection issue is identical to a well understood distributed computing problem. In showing this congruence, it is now possible to solve the pointer tracking problem by application of a known, proven and well understood solution to the DTD problem, i.e. a DTDA.

The second model is implicit in Hughes collector [52] where a DTDA is used to detect termination of some **process** that identifies garbage. Tel *et al* [95]

construct an explicit model of similar forward-tracing distributed collectors, mapping termination of the marking process (a diffusing computation) to DTD. This approach is described and informally generalised in Starting with Termination [20], wherein uniprocessor collectors are extended to a distributed context by application of a DTDA and additional design work. The paper shows that garbage collection in a distributed context generally contains a process that conforms to the DTD system constraints and therefore the termination of this process may be detected by a DTDA; the specification of this garbage collection process is not formalised and therefore left to the designer.

Norcross [82] and Lowry [64] have both presented models of the Train Algorithm within a distributed context, both specifying that the isolation of a train may be detected using a DTDA; their theses are formal examples of the Starting with Termination approach that produce high performance hybrid garbage collectors. Norcross' specification of the Club Rules defines how a partition collector must operate in order for it to correctly interact with the train algorithm, thereby providing a formal separation between the complete collector (Trains) and the partition collectors.

Zigman [102] takes a slightly different and more general approach, permitting the use of arbitrary collection approaches at each layer of a hierarchy of subgraphs. Each collector is free to operate as it chooses, treating subgraphs as individual nodes of the graph that it can observe and the motion of objects (and therefore pointers) between subgraphs as mutator activity.

### 2.4.1 The Train Algorithm (Lowry)

Lowry [64] defines a formal event/transition model of the train algorithm, starting without cars and defining how progress must be made by reassociation to achieve separation of the live and dead objects and therefore complete garbage collection. The behaviour of the mutator is defined in terms of allowable events that it may execute and from there, the garbage collector is defined in terms of the events that it may execute and which events are required to achieve completeness. The model is then extended to permit asynchronous updates to metadata, i.e. stale knowledge of train membership at remote sites, then extended again to re-introduce cars as a performance optimisation.

Having presented a formal model of the collector events, safety and completeness are used to define when each reclamation event may occur. These rules are used to derive protocols that manage the train structures, i.e. for creating, joining, leaving, detecting the isolation of and finally destroying trains.

Lowry recognises the race condition present in the original DMOS paper and uses synchronisation to solve the problem, i.e. reassociation progress and

the train maintenance protocols are halted while the train isolation DTDA is executing. Lowry's trains use asynchronous wave-based (of unconstrained topology, e.g. ring or tree) DTDAs to detect isolation; while the wave is in progress, no reassociation is permitted. To achieve this synchronisation, a **closing wave** traverses the train, receipt of which indicates to member-sites that no reassociation into or object creation within that train should occur; once the closing wave has returned, DTDA execution begins. If the DTDA wave returns without detecting isolation, an **opening wave** traverses the train to indicate that reassociation is to begin again. By ensuring that the reassociation process may not occur concurrently with the DTDA, the unwanted relative problem is no longer dangerous and the system retains its safety. Because closure prevents sites joining a train, creating objects in a train and reassociation into a train, train closure introduces race conditions whereby those three activities are indefinitely delayed because each attempt to perform the relevant activity is blocked by the closure of a train; the approach presented by Lowry is to have the collector remember such instances of blocked progress and perform them immediately that the train is reopened, thereby restoring liveness.

Lowry's formal description of the train algorithm improves on DMOS in a number of ways, largely relating to asynchrony, i.e. the ability of the collector to reclaim garbage without synchronising with other sites. For example, Lowry's trains segregate private objects (to which no external references exist) from public objects, permitting the reclamation of private garbage with no communication, whereas in DMOS, public and private garbage may be intermixed. Likewise, Lowry's trains permit concurrent collection within a site and there is less mutator disruption due to the explicit description of the algorithm in terms of a concurrent readers, exclusive write model of data access.

Lowry's thesis presents a convincing argument-by-analogy that it is safe: train isolation detection is mapped to distributed termination detection. For each concept within the DTD paradigm (jobs, tasks, messages, etc), a corresponding concept is found in the isolation of trains (train, reachable objects, pointer tracking messages, etc) and the collector is constructed so that the accuracy of the analogy is not violated. For example, by mapping a task of the DTDA to an object that is reachable from outside the train, the constraints on task behaviour, notably creation, must be observed by ensuring that the same constraints apply to the reachability of objects from outside the train.

Completeness in the train algorithm is proven via use of a finite set of constraints on the instantiated collector; the proof combines the effects of these constraints to show that a garbage object will be reclaimed within finite time. The presence of these constraints is beneficial in that it provides a mechanistic path to

the instantiation of a garbage collector: if an implementor fulfils the constraints, the result should be a safe and complete distributed garbage collector.

The downside to this level of rigour and detail is that the abstract model of trains can describe only a narrow class of garbage collectors, i.e. all instantiations are merely variations on the train algorithm. It may be argued that this is not a meaningful drawback: Lowry observes that the train algorithm exhibits all the desirable properties of a high quality garbage collector in terms of safety, completeness, asynchrony and concurrency. In some ways however, Lowry's model is unnecessarily restrictive because it describes operation only with a wave-based DTD: the closure of trains is disruptive to collection progress because all sites within a train must cease reassociation activity while isolation detection occurs. In contrast, the use of a Doomsday-class DTDA as described by Norcross contains sufficient synchronisation to make the system safe but it does not force synchronisation on all other sites within a train to achieve that safety.

Given that Lowry's thesis focuses on high quality (high performance) garbage collection through asynchrony and concurrency, much of the analysis that Lowry presents is aimed at constraining the development of a train-algorithm garbage collector to a high performance solution. As such, the derivation of the model and the stated goals in that derivation provide insight into the likely high performance of collectors instantiated from Lowry's model; conversely, the narrowness of the model means that it provides little in the way of analysis tools that may be used to compare the relative performance of different collectors that it may instantiate.

The model is readily comprehensible at an abstract level (if not the details of the proof) and relatively easy to apply to a new collector design.

### 2.4.2 The Train Algorithm (Norcross)

Norcross [82] focuses on the application of the Task Balancing (TB) distributed termination detection algorithm in the creation of distributed garbage collectors based on trains. Task Balancing was first defined as a pointer tracking algorithm [50] for DMOS and work performed on proving that algorithm was generalised into the Doomsday [63] approach to DTD (of which Task Balancing is an instance). Norcross advocates the use of TB in garbage collectors because it is asynchronous, concurrent (non-interrupting) and scalable. While Norcross' thesis uses TB to solve all DTD requirements in the collectors so derived, it does not appear that the model defined therein is restricted to the use of TB.

Norcross' major contribution is to formally define an interface between the train algorithm and the car collectors that reclaim acyclic garbage. Because reassociation (and therefore progress of the train algorithm) is tied by

convenience and efficiency to the car collector, there are requirements that a car collector must fulfil for it to work correctly within the train algorithm: these requirements are dubbed the **club rules**. The aim of the club rules is that they sufficiently specify the behaviour of partition garbage collection at each site so as to completely capture the interface between local collector and train algorithm. The next step is to note that local collectors do not interact except via train algorithm mechanisms, therefore homogeneity of local collection is not required; any heterogeneous selection of local collectors may be implemented within a system so long as every local collector conforms to the club rules.

To derive the club rules, the Starting With Termination [20] approach is used to define a mapping from a uniprocessor collector to a distributed collector, then one must decide which critical state has been made distributed and what predicate must be evaluated over that state. Once this is decided, a mapping of the termination predicate to the DTD paradigm is made and the club rules are derived from this mapping. The rules are chosen such that they constrain local collector behaviour to be conformant with the mapping to DTD, i.e. all information that the DTDA requires is made available and local sites are prevented from acting in ways that do not conform with the DTD model, e.g. spontaneous task creation.

A number of uniprocessor collectors (mark-sweep, generational and reference counting) are mapped to distributed systems and club rules defined for each using this process. Finally, an abstract informal definition of MOS is presented — Unordered MOS (UMOS) — which when transformed into a distributed system, yields a variant of the distributed train algorithm and a set of club rules. There are two mappings-to-DTD used: one for object isolation from other trains and one for train isolation. Because the Starting With Termination approach is taken, Norcross noted the existence of the unwanted relative problem in distributed trains.

By using a Doomsday-class DTDA to detect train isolation, the only synchronisation required for a site wishing to perform an unwanted-relative reassociation is to request a witness task of the train isolation job from the home site of the train. Once this witness task has been received, the reassociation (task creation) may go ahead. If the train was detected as isolated and therefore reclaimed while the witness request was in flight, a denial message will be sent back; in this case the train is about to be reclaimed and there is no longer any requirement to reassociate objects into it. This witness-request synchronisation introduces a small degree of centralisation but it avoids the problem present in Lowry's model of forcing synchronisation on all other sites that are members of the same train; the Doomsday synchronisation approach therefore exhibits more

concurrency and less interruption than Lowry's train closure and wave DTDAs.

Norcross' model is safe if it is proven that the derived club rules are sufficient to ensure that the DTD model requirements are met but such a proof is not given. On the assumption that the club rules are sufficient; the club rules provide a highly mechanistic approach to the design of local collectors within a distributed collector based on the train algorithm. Even if the club rules stated by Norcross were proved to be insufficient, the central idea of stating the requirements on a local collector in terms of club rules is valid. No proof of completeness is given.

The model is similar to Lowry's in its narrowness: it is capable of describing only a narrow range of collectors, i.e. those based on the train algorithm; the primary parameter that is permitted to vary within the model is the local collector. The construction of the model is based on the use of Task Balancing for both object isolation and train isolation, though it seems that since the club rules ensure satisfaction of the DTD model constraints that any asynchronous Doomsday-class DTDA would work. No means of analysing the collectors instantiated from the model are provided, nor is the approach tested with other DTDAs, i.e. an implementor gains no information from the model as to the relative merits of selecting particular local collectors or DTD algorithms.

The model is readily comprehensible and while lacking a formal proof, the concrete list of club rules means that it is relatively easy to apply to a new collector design.

### 2.4.3 Hierarchical Collectors (Zigman)

Zigman [102] presents a model of hierarchical collectors that permits a designer to compose multiple garbage collection algorithms into a single larger algorithm. The model involves dividing the object graph into subgraphs and operating a separate garbage collector in each subgraph. To ensure completeness at the top level, objects must be able to migrate between subgraphs and thereby isolate garbage within a subgraph. By recursively dividing the graph, a multi-level collector of arbitrary depth may be formed, though typical instantiations will contain only two or three levels. Since subgraphs appear to be single graph nodes to the upper levels, the migration of objects between the subgraphs will change the topology observable to the upper levels in the same way that mutator activity does; garbage collection activity at the lower levels is therefore represented as mutation to the upper levels.

The propagation of reachability information in a tracing collector is represented as work: where reachability of a graph node is known, there exists work for each pointer that it contains. The processing of that work may result in new work being generated; the process continues processing work until none

remains and the reachable portion of the graph is completely discovered. This model of work is applied to the growth of subgraphs, or rather the motion of subgraph boundaries across the object graph, resulting in **structure-based reorganisation** of the subgraphs. The model asserts that where there is an ordering on subgraphs, only simple isolation detection — reference counting — is required at the upper level of the collector to detect isolation of the lower level subgraphs; the reason for this is that the ordering on subgraphs will, in conjunction with liveness of reorganisation, result in every cycle of garbage eventually being entirely contained within a subgraph and all live objects removed from that subgraph. Conversely where no such ordering exists, the isolation of a given subgraph is not guaranteed but it is possible for cycles of subgraphs to become isolated; in that case, a complete collector is required at the upper level.

Zigman's model is very general and seems to encompass a very wide range of published garbage collection algorithms: the structural reorganisation process formulated as tracing work means that the model is capable of unifying tracing algorithms and explicit reorganisation approaches such as the Train Algorithm as different points within a single design space. The drawback to this level of generality is that there is no concrete process that permits the model to be used in the construction of a new collector.

The model does not explicitly consider distribution but instead is concerned with mutation of an abstract graph of nodes. However, Zigman describes DMOS in terms of the model, breaking it down into the components defined by the model: definition of subgraphs, reorganisation processes, etc. In doing so, it is noted that state has become distributed: what would have been a reference count in a uniprocessor system is now distributed and requires additional means to determine when it has reached zero. The description of DMOS notes the presence of race conditions in the original DMOS publication and observes that these are due to state being distributed without a formal means to account for its coherency. Zigman's model does not address these concerns, it merely notes that where state is to be distributed, a more formal approach than that taken originally with DMOS is likely advisable. Likewise, the hierarchical model does not prove safety or completeness of distributed trains but it does provide informal assurance that the MOS algorithm is correct in its most abstract formulation.

The primary value of the model seems not to be in constructing or proving particular garbage collection algorithms but rather an analysis of what components are necessary in the construction of a garbage identification algorithm.

## 2.5 Requirement for a New Model of Distributed GC

The previous section describes three models of garbage collection, each exhibiting a different mixture of desirable properties:

- Lowry's model contains a formal model and proof of the distributed train algorithm and uses it to instantiate a high quality collector; the drawback is that the model is highly specific in that it describes only the train algorithm with wave-like DTDAs.
- Norcross' model formally separates the operation of partition collection from the train algorithm, permitting heterogeneous local collection if the club rules are followed; it too is specific to the train algorithm.
- Zigman's model is extremely general but contains little in the way of specific detail; it provides no proof of correctness or safety and no guidance in the construction of a new collector.

There is therefore a need for a model to bridge the gap between these existing models. Ideally, this new model should fulfil all of the requirements stated earlier. Despite the respective drawbacks of each of these models, they have collectively provided great inspiration for and strongly influenced the Surf model that is presented in this thesis.

## 2.6 Summary

Distributed garbage collection has been a fertile research field for 30 years and in that time, some parts of the problem have become well understood while a number of incorrect collectors have been published because they were designed without the benefit of a formal framework that may be used to prove their correctness. The recent emergence of models of garbage collection that provide a somewhat mechanistic process for designing collectors is encouraging; the rigour of the derivation process leads to rigour in applying a proof constructed for the model to an instantiation of the model.

This chapter reviews three models of garbage collection at different levels of abstraction and therefore providing different sets of desirable properties. The most flexible models published do not provide a mechanistic means for deriving collectors from the model and thereby proving them, while the models concerning the Train Algorithm may provide a rigorous proof and a mechanistic process for arriving at an instantiation but they are limited to producing a narrow class of collectors. There is therefore a need for a new abstract model of garbage

collection which not only describes a wide range of garbage collection algorithms but also provides a proof that may be applied to any instantiation, a relatively simple process for ensuring that the proof is valid for an instantiation and some analysis that may be applied to an instantiation to gain some idea of its likely performance characteristics.

This thesis presents such a model.

# Chapter 3

# Unifying Distributed Garbage Collection

This chapter presents **Surf**[1], an abstract model of distributed garbage collection that is intended to capture the essence of all possible **fundamental garbage collection algorithms**, i.e. those that are not composed of other collectors. The aim of this model is to build a formal system for describing the operation of a garbage collector: the means by which it makes progress in applying labels to objects and how it decides when the labelling implies that certain objects are garbage. Surf provides a unified description of how distributed garbage collectors make progress and brings together a wide range of collectors within a single framework. According to the Surf model, a fundamental collector is composed of one or more relabelling processes and distributed termination detection algorithms that detect termination of relabelling. Relabelling processes are constructed so that work within each process corresponds to the existence of pointers into a particular graph region; termination of a process therefore permits inferences to be drawn regarding unreachability of that region.

In contrast, a composed collector contains one or more smaller garbage collection algorithms; for example, practical instances of the train algorithm contain an instance of the train algorithm [64] at the top level and partition collectors below that to reclaim acyclic garbage and perform some actions as required by the train algorithm [82]. The compound general case is described in Zigman's thesis [102]. One may not construct a compound collector unless one already has an existing fundamental collector therefore the description of fundamental collectors is a necessary step in the construction of compound collectors.

[1]Surf is so named because the typical execution pattern is of a number of waves passing concurrently over the object graph.

By using the Distributed Termination Detection (DTD) abstraction to make decisions as to the reachability of regions within the graph, the Surf model requires no extension for application in a distributed context. Because the DTD paradigm can detect stable states, it may be used to detect isolation since that is a stable property of garbage. The Surf model defines processes that a DTD algorithm may observe; these processes are constructed in such a way that their termination permits inferences regarding unreachability to be drawn.

The model is conceptually simple and the proof of its safety and completeness identifies a number of constraints on the ways in which it may be instantiated; these constraints define the bounds of a constructive approach to the creation of new garbage collectors. The model may be used as an aid in proving the correctness (or otherwise) of specific garbage collectors and it may be used to derive new garbage collectors.

Described in this chapter are the system model within which the abstract model of collection is described, a definition of safe and complete garbage collection and the Surf abstract model of garbage collection. The safety and completeness of the model are proven in this chapter; later chapters explore the descriptive power of the model by describing existing collectors using the model, investigate the performance of an instantiation of the model and then explore new areas of the design space provided by the model. A summary of the model is given as a description of the design space that the model defines as well as a list of the constraints that an individual collector must fulfil for the proof of correctness to apply.

## 3.1 System Model

### 3.1.1 Objects and Pointers

The system is composed of objects and pointers; each object can contain any number of pointers that refer to other objects. There exists a special **root object** which defines reachability: any object (in)directly reachable from the root is considered live. The collection of objects (nodes) and pointers (edges) forms a directed graph, the topology being defined by the application (mutator) that manipulates the object graph for its own ends.

Each pointer contains the **name** of an object, which is by definition enough information to locate the object and perform operations (see Allowable events, Section 3.1.4) thereon. Objects may contain non-pointer data which is irrelevant to any abstract discussion of garbage collection.

The system must provide referential integrity, i.e. there is no method to manufacture pointers and the system must be aware of every pointer contained therein. The only way of accessing an object is via a pointer to that object, obtained from some other object. Mutators only know the name of the root object, therefore to access an object in the system there must be a chain of pointers to it from the root, hence the definition of liveness.

Objects that are no longer reachable from the root are not accessible to the mutator; they are garbage and should be reclaimed by the system. A more rigorous definition is given in Section 3.2.

An object $y$ may contain a pointer to an object $x$ (which may or may not be distinct from y); the pointer inside $y$ is denoted $y.x$.

The term **component** is used throughout this thesis to describe a connected region of the graph. A **strongly connected component** is one where every object in the component is reachable from every other object in the component.

### 3.1.2 Distributed Computation

In distributed instantiations of the model, each object resides entirely on a single site and is not split across site boundaries or duplicated between sites[2]. For the purposes of the proof and generality thereof, each object may be considered as existing at its own site in the distributed system except where a specific instantiation of the model requires otherwise; this implies that objects and the computations accessing them operate asynchronously.

Computation at each site is modeled as a sequence of **events** that define atomic transitions between two subsequent **states** for that site. Individual states are denoted $S_P^k$ where $P$ defines the site ID (subscripts in general denote site IDs) and $k$ defines the sequence number of that state, a number which is local to, and with meaning only within, the site described by that state. The event at site $P$ leading to state $S_P^k$ is denoted $E_P^k$; its predecessor state is $S_P^{k-1}$. The happens-before relationship is denoted by the symbol $\prec$ and it is used to describe the presence of ordering between events and states.

States have identity and occur in order, i.e. $E_P^k \prec S_P^k \prec E_P^{k+1} \prec S_P^{k+1}$. Even though two distinct states may be functionally identical, i.e. be represented by the same bit pattern on the site in question, they are distinct states and have their own place in the total ordering on states. This implies that there is no concurrent computation within a site; where concurrency exists it is represented in the model as computation on separate sites.

[2]It may be possible to extend this model to consider objects that are themselves distributed by replication or splitting across site boundaries but no such extension is considered here.

### 3.1.3 Predicates

The expression of local predicates on a particular state is as follows and indicates that the stated predicate is true for that state:

$$S_P^k : \{predicate\}$$

The statement $predicate \rhd event$ is taken to mean that should $predicate$ become true, $event$ must be executed within finite time, likewise $predicate \widetilde{\rhd} event$ means that $event$ is permitted to occur after predicate becomes true. In other words, an event described in this way may only occur if a predicate that permits the event was true at some state preceding (by causal ordering) the event in question:

$$\exists event \Rightarrow \exists S_P^k \mid \left(S_P^k : \{predicate\} \wedge predicate \widetilde{\rhd} event \wedge S_P^k \prec event\right)$$

Communication between sites is by asynchronous message passing; not necessarily FIFO though some instantiations of the model may require that as an additional constraint. Fault tolerance is not considered in the system model: all messages are delivered in finite time without corruption or duplication, sites do not crash and do not exhibit Byzantine behaviour.

Message transmission and receipt are two separate events, typically occurring at different sites; for a message sent from site $P$ to $Q$:

$$send_P \prec recv_Q$$

This causal relationship between events is provided by the message traversing the network between sites; it is the only mechanism by which a causal ordering between states and events on different sites may be established. Computation speed at a given site is unbounded, i.e. message transmission provides an ordering only of the send and receive events, not any events following transmission or prior to reception.

### 3.1.4 Allowable Events

Events of interest to the collector involve the mutator modifying the graph structure and the transport of pointers between sites in messages. The allowable mutator events are:

$create(y.x \leftarrow (x = new))_P^k$

object $x$ created at site $P$ at timestep $k_P$ and pointer copied into $y$.
$S_P^{k-1} : \{\exists y \wedge \neg\exists x\} \wedge S_P^k : \{\exists x \wedge \exists y.x\}$
In other words, given the event $create_P^k$, $y$ exists at the prior state while $x$ does not, $S_P^{k-1}$ but $x$ does exist at the following state, $S_P^k$ and there

also exists a pointer to $x$ in $y$.
"$\leftarrow$" has no formal meaning as an operator outside of the $create$ event; informally it is syntactic sugar indicating that pointer assignment occurs as part of a $create$ event.

$send(y.x \rightarrow Q)^k_P$
site $P$ sends a copy of the pointer-to-$x$ in object $y$ to site $Q$. No change of graph state at $P$ but the event must be observable to the garbage collector.
$S^{k-1}_P : \{\exists y.x\}$
"$\rightarrow$" has no formal meaning as an operator outside the definition of the $send$, $recv$ and $plus$ events; informally it is syntactic sugar indicating the transmission of a pointer value in the direction of the $\rightarrow$ from one site to another occurring during one of these events.

$recv(P \rightarrow z.x)^k_Q$
site $Q$ receives a copy of the pointer-to-$x$ from site $P$ and stores it into object $z$
$S^k_Q : \{\exists z.x\}$

$minus(y.x)^k_P$
erasure of pointer to $x$ in object $y$ at site $P$. $x$ does not necessarily reside at $P$.
$S^{k-1}_P : \{\exists y.x\} \wedge S^k_P : \{\neg\exists y.x\}$

$plus(z.x \rightarrow y.x)^k_P$
This is not actually an event in its own right, it is the composition of a $send$ and $recv$ pair on the same site wherein $z$ sends a pointer-to-$x$ to $y$.

All mutator activity is defined in terms of sequences of the above events, one sequence executing at each site in the system.

## 3.2 Definition of Garbage Collection

Garbage collection (GC) is defined as an exercise in the determination of graph connectivity. Stated simply, an object is **live** if it is within the transitive closure of the root, otherwise **dead**. Since the mutator may access objects only via the root and therefore only live objects, **dead** is a stable property: once true, it remains true since there is no means for the mutator to obtain the name of a dead object. The purpose of a garbage collector is to detect objects that are dead so that the space

they occupy may be reclaimed but the Surf abstract model deals only with the detection of garbage; allocation and reclamation of objects are mechanistic details dependent on the store architecture, irrelevant to detection and not considered here.

Two necessary properties for a correct garbage collector are **safety** and **completeness**. Safety refers to the property that no live object will ever be detected as garbage and completeness refers to the property that dead objects will be detected as garbage within finite time. There are other desirable performance-related properties of garbage collectors (throughput, timeliness of reclamation, scalability and low space overhead) that are not considered here; this chapter is entirely concerned with the correctness (safety and completeness) of a garbage collector as expressible by the abstract model. Different instantiations of the model will exhibit more or less desirable performance characteristics according to the details of each particular instantiation.

A common-sense constraint on garbage collection is that it must be non-interfering, i.e. must not mutate any live portion of the object graph. It exists only to observe topological changes wrought by the mutator.

The GC has a single event that it may invoke on an object $x$, $Reclaim(x)$. The execution of this event indicates that the GC has determined that $x$ is unreachable and its space should be reclaimed.

### 3.2.1 Reachability

Consider objects $a$ and $b$; $a$ contains a pointer to $b$ which is denoted $a.b$; the root object is denoted $r$ and its identity is known *a priori*. Each object, including the root, may contain an arbitrary number of pointers. Referential integrity in the store implies that $\exists a.b \Rightarrow \exists a \wedge \exists b$; as a matter of practicality this requirement may be violated in dead regions of the graph because reclamation of distributed regions that have been detected as unreachable will not be atomic. The exception to referential integrity is not observable to the mutator nor to the algorithms that detect garbage and is therefore ignored for the purposes of this thesis because we are concerned only with the detection (not reclamation) of garbage.

Liveness of an object $x$ is denoted by the predicate $\mathbf{Live}(x)$ and unreachability as $\mathbf{Dead}(x)$

$$\mathbf{Live}(x) \equiv \mathbf{Reachable}(r, x)$$
$$\mathbf{Reachable}(z, x) \equiv (x = z) \vee (\exists z.x) \vee (\exists y \mid (\mathbf{Reachable}(z, y) \wedge \exists y.x))$$
$$\mathbf{Dead}(x) \equiv \neg\mathbf{Live}(x)$$

Furthermore, **trivial unreachability** is defined as the lack of pointers to an object; clearly an object that is trivially unreachable is garbage but the converse does not hold:

$$\mathbf{TriviallyUnreachable}(x) \equiv \neg\exists y.x$$
$$\mathbf{TriviallyUnreachable}(x) \Rightarrow \mathbf{Dead}(x)$$

### 3.2.2 Safety and Completeness

Safety implies that at no time shall a live object be reclaimed; it is stated formally $\forall x$ as:

$$\exists \mathbf{Reclaim}(x)^k_P \Rightarrow S^{k-1}_P : \{\mathbf{Dead}(x)\}$$

i.e., if the reclaim event is invoked on an object, the objects must be dead in the state before the reclamation event.

Completeness states that all dead objects will be eventually reclaimed; formally, the $Reclaim(x)$ event must be executed for every dead object $x$:

$$\mathbf{Dead}(x) \rhd Reclaim(x)$$

### 3.2.3 Mutator

Mutators are modelled as processes that manipulate the graph topology via the allowable events of Section 3.1.4. A mutator may invoke an event on an object if it holds a pointer (inside the root) to that object:

$$\exists event(x)_P \Rightarrow \exists r.x$$

The copying of pointers to and from the root is governed by the allowable events.

### 3.2.4 Summary of GC

Liveness or otherwise of an object is defined in terms of its reachability from the root object and the purpose of a garbage collector is to detect dead objects. A garbage collector should be both safe and complete; these properties are defined here formally and may be evaluated over any GC algorithm, regardless of its construction. The definitions of safety and completeness provided here are the goals of the abstract model's proof, i.e. the proof operates by showing that these properties are true where the model has been correctly instantiated.

## 3.3 Surf: the Abstract Model of GC

The tests above define what a garbage collector must do, while the Surf abstract model defines a constructive approach to GC; it lists a number of components that are combined within certain constraints to produce a correct GC. Given that this model is capable of expressing a wide range of existing GCs, there is no guarantee that the collector created will have desirable performance characteristics: the decisions made during composition will define the new collector's performance. The model provides only a guarantee of correctness; to specify further would narrow the context of the model to a small subset of all possible fundamental collectors.

A range of additional mechanisms (see Sections 3.5 and 3.5.10) are required for each collector but these mechanisms are not part of the essence of a garbage collector, i.e. they do not define the means by which the collector makes progress. Specifying the design of any of these mechanisms would unnecessarily restrict the model to that particular design whereas the model encompasses collectors with every possible implementation of those mechanisms. Likewise, the model defines a design space wherein it appears that certain areas of the design space result in collectors with more desirable performance characteristics than other areas (see Chapter 4 for examples); while it would be possible to narrow the model to describe only high performance collectors, that would be an unnecessary reduction of the model's descriptive power. By taking the broad approach, the model is also capable of describing the lower-performance collectors — they are, after all, complete collectors and worthy of description and proofs of correctness — and the model provides insight into why, even at the abstract level, some collectors will have better performance than others.

The abstract model is composed of three important concepts: **labels** defining **regions**, a **relabelling process** that changes the labels on objects and a **distributed termination detection algorithm** that detects relabelling processes terminate and therefore that regions are isolated. This section proceeds by explaining each of these terms and how they cooperate within the model to form a safe and complete garbage collector. Having shown the abstract process by which progress is made and isolation of regions are detected, Remembered Sets are presented as an implementation technique for detecting trivial isolation of objects and the presence of inter-region pointers.

### 3.3.1 Labelling and Regions

Fundamental to the task of garbage collection is the process of **labelling** objects: since a garbage collector must reclaim objects, it must decide which ones to

reclaim. Objects suspected of being garbage are given one label while objects suspected of being live are given (a) different label(s). The process of applying labels according to connectivity continues until it is known that no object with a particular label, $L$ is reachable from any object with a different label; if the label on the root is not $L$ then every object with that label is garbage by the definition given in the previous section. Constructing a working garbage collector requires a relabelling process that will result in isolated regions (a **useful relabelling process**) and the means by which the such processes are constructed is described below; this subsection merely explores the reasoning available with respect to reachability that follows from a particular labelling of the graph.

Labels are described here in the most general sense in that they contain information that the collector knows about a particular object. This information may be implicit in the execution of the GC, it may be encoded in each object's name (address), it may be represented by a collection of tag bits associated with each object yet invisible to the mutator; it may be any combination of this information. A GC may use any number of labels and the meaning of any given label may change with time. The label of an object is referred to by the function $label(x)$ that may be evaluated at any time (i.e. in any state) at the site containing $x$. Labels are not observable by the mutator.

The set of objects with a particular label is defined as a **region**, the region named by a $\mathfrak{Gothic}$ version of the label in question. Regions may span many sites within the system.

$$\mathfrak{L} \equiv \{x \mid label(x) = L\}$$
$$(label(x) = L) \equiv x \in \mathfrak{L}$$

It is now necessary to define a **useful labelling**, i.e. one wherein the labels convey information about the reachability of objects. For a labelling to be useful, at some point in time the GC must be able to decide that certain regions contain only unreachable objects and therefore that those regions may be reclaimed. We extend the **Dead** predicate to regions and define it to be true when every object in a region is dead:

$$\mathbf{Dead}(\mathfrak{L}) \equiv (x \in \mathfrak{L} \Rightarrow \mathbf{Dead}(x))$$

For the purposes of detecting that a region is dead, we define a subset of the dead regions referred to as the **usefully-dead regions**; these regions are not only dead but there exist no pointers from other regions into a usefully-dead region. A usefully dead region does not contain the root and for every object in the region, there are no pointers to that object from objects in other regions:

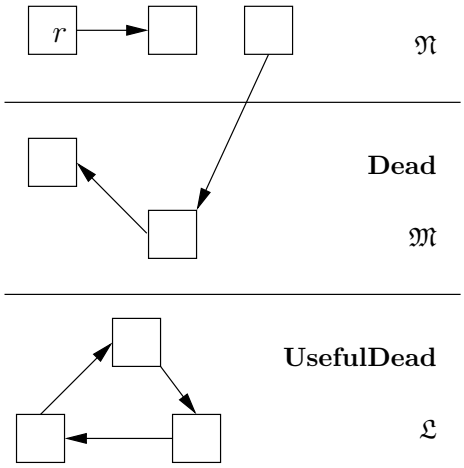


Figure 6: (Usefully) Dead Regions

$$\mathbf{UsefulDead}(\mathfrak{L}) \equiv r \notin \mathfrak{L} \wedge x \in \mathfrak{L} \Rightarrow (\neg \exists y.x \mid y \notin \mathfrak{L})$$
$$\mathbf{UsefulDead}(\mathfrak{L}) \Rightarrow \mathbf{Dead}(\mathfrak{L})$$

Since a usefully-dead region may not contain the root and there are no pointers to it from other regions, it must also be a dead region. These regions are usefully dead since it is possible to detect the lack of pointers into a region and therefore detect that the region is dead. The abstract model operates on the basis of constructing usefully dead regions through relabelling and then detecting them as such; merely creating (non-useful) dead regions would not permit the step of detecting that a region is dead. The detection of usefully dead regions (lack of pointers to a region) is bound up in the definition of a useful relabelling process and the termination thereof, described below: a useful relabelling process terminates when there exist no inter-region pointers for some combination of regions that the process is operating over.

The concepts described above are illustrated in Figure 6. Region $\mathfrak{L}$ is usefully dead since there are no pointers into the region, Region $\mathfrak{M}$ is dead but not usefully dead because it contains only dead objects but it is the target of an inter-region pointer and Region $\mathfrak{N}$ is not dead because it contains live objects.

Because a Surf-derived GC operates by reclaiming whole regions, only dead regions may be reclaimed otherwise safety will be violated. Completeness requires that every dead object is labelled as part of a usefully dead region within finite time and may therefore be reclaimed; this requirement is the core definition of a useful labelling scheme that will result in a safe and complete collector:

$$\mathbf{Dead}(x) \rhd (x \in \mathfrak{L} \wedge \mathbf{UsefulDead}(\mathfrak{L}))$$

In other words, every dead object must be relabelled to a usefully dead region or the region that contains it must become usefully dead within finite time.

This collection-by-labelling approach requires two algorithmic components: a means to apply labels to objects and a means to decide when a region is isolated; these components are described below.

### 3.3.2 Relabelling Process

The **relabelling process** is driven by a set of rules defining how labels are applied to objects; the design of this component provides a useful labelling, which is necessary to have liveness and therefore completeness of collection. Since the goal is a useful labelling, implying a lack of inter-region pointers, the purpose of the relabelling process is to change the labels on objects so that inter-region pointers become intra-region pointers. Applying a different label to an object is an event under the system model, i.e. it is instantaneous. For historic reasons, the event of relabelling may be referred to as **reassociation** or **reorganisation** — terms used by existing collectors to describe the application of a new label. The relabelling process only changes the labels on objects (its only permitted event is $relabel$); it does not mutate the graph and therefore is invisible to the mutator. At least one relabelling process is required, though a collector may have several.

The relabelling event of object $x$ at site $P$, timestep $k$ to label $L$ is denoted $relabel(x \rightarrow L)^k_P$

To form usefully dead regions, each relabelling process must discover the locations where a pointer exists between two regions and then relabel either the source or destination of that pointer so that both source and target objects are within the same region.

The **candidate** region of a particular relabelling process is the region for which an inference may be drawn regarding its being usefully dead at the termination of that relabelling process. Because relabelling processes are defined differently for each instantiation of the model, the way in which this inference is drawn varies between collectors; the heart of the definition though lies in the definition of work for a relabelling process being equivalent to the existence of certain inter-region pointers. Termination of a relabelling process therefore implies a lack of pointers to a certain region, perhaps from a subset of other regions. If there are no pointers from other regions then the region in question is usefully dead. Where the definition of a relabelling job includes only pointers from some subset of other regions, external logic is required to infer region deadness from one or more relabelling jobs terminating and external conditions that are dependent on the relabelling job definitions.

The region containing the root is not a candidate region of any relabelling process; other regions not containing the root may or may not be candidates of different relabelling processes.

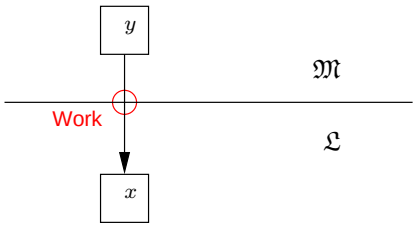


Figure 7: Inter-Region Pointers are Work

$$candidate(\mathfrak{L}) \Rightarrow r \notin \mathfrak{L}$$

Given two objects in different regions, the presence of a pointer between those regions implies there is work for the relabelling process:

$$label(x) = L, label(y) = M$$
$$(\exists y.x \mid (L \neq M \wedge candidate(\mathfrak{L}))) \equiv work(\mathfrak{L})$$

The presence of work means that an object must be relabelled across the region boundary to eliminate the work, i.e. make progress:

$$work(\mathfrak{L}) \quad \widetilde{\triangleright} \quad [relabel(y \rightarrow L) \quad or \quad relabel(x \rightarrow M)]$$

Note that the above defines work only for $\mathfrak{L}$ because $\mathfrak{L}$ is the target region so the processing of this work will make progress towards the lack of inter-region pointers to $\mathfrak{L}$. Processing this work may involve relabelling an object in $\mathfrak{L}$ or $\mathfrak{M}$.

The above expression defines **work** for the relabelling process; there are often a number of relabellings possible since objects may contain an arbitrary number of pointers and be the target of an arbitrary number of pointers. Where work is available for the relabelling process, it must be processed within finite time to guarantee liveness. The situation is illustrated in Figure 7; the presence of an inter-region pointer from $y$ to $x$ implies that there is work for the relabelling process for which $\mathfrak{L}$ is candidate.

By the definition of work, the lack of work means that there exist no inter-partition pointers to objects in candidate regions of the relabelling process for which there is no work. The lack of any such inter-region pointers in combination with a region not containing the root is identical to the definition of a usefully dead region, therefore the lack of work on a candidate region implies that the region is usefully dead.

$$\begin{aligned} \neg work(\mathfrak{L}) \wedge candidate(\mathfrak{L}) &\equiv r \notin \mathfrak{L} \wedge \forall y.x \mid (x \in \mathfrak{L}),\ y \in \mathfrak{L} \\ &\equiv r \notin \mathfrak{L} \wedge \neg\exists y.x \mid (x \in \mathfrak{L} \wedge y \notin \mathfrak{L}) \\ &\equiv \mathbf{UsefulDead}(\mathfrak{L}) \end{aligned}$$

The above definition of work for a relabelling process implies that termination of relabelling (no remaining work) is equivalent to isolation of a region. A mapping of this problem to Distributed Termination Detection is provided in the Proof (Section 3.4) which applies equally to the detection of isolation and the termination of relabelling.

Relabelling is a diffusing process since the performance of a unit of work — the removal of an inter-region pointer by relabelling either the source or target of that pointer — may result in the creation of more work at adjacent objects in the graph. Consider a pointer $y.x$ that constitutes work, i.e. is inter-region, the only ways to process the work are to relabel either $x$ or $y$ to make the pointer intra-region; doing so may change the inter-region status of other pointers associated with the object that is relabelled. There are four cases to consider: pointers to and from the relabelled object and whether the relabelled object is the source of target of the pointer that caused relabelling.

Where the source of the pointer ($y$ for $y.x$) is relabelled, intra-region pointers to the source ($z.y \mid label(z) = M$) will become inter-region pointers ($z.y \mid label(z) \neq L$) to the destination region, $L$. Intra-region pointers contained in the source ($y.z \mid label(z) = M$) will become inter-region pointers ($y.z \mid label(z) \neq L$) to the source region, $M$. Inter-region pointers related to $y$ may remain as such if their other endpoint is not in $L$, however the nature of the work they represent will change.

Where the target of the pointer ($x$ for $y.x$) is relabelled, intra-region pointers to the target ($z.x \mid label(z) = L$) will become inter-region pointers ($z.x \mid label(z) \neq M$) to the source region, $M$. Intra-region pointers contained in the target ($x.z \mid label(z) = L$) will become inter-region ($x.z \mid label(z) \neq M$) pointers to the target region, $L$. Inter-region pointers related to $x$ may remain as such if their other endpoint is not in $M$ and the nature of the work they represent will change.

Therefore the processing of a unit of work (inter-region pointer) may result in the creation of new units of work at adjacent portions of the graph; this creation of work (and by implication, the processing of work) does not cease until regions become isolated. Safety requires that this potential creation of work tasks is correctly fitted into the DTD model, see below. Liveness of the collector depends on the relabelling process selecting appropriate units of work to process from the many available at any given time so as to ensure that objects do not move cyclically amongst regions; one approach to preventing cyclic relabelling is to impose an ordering on regions and permit relabelling in a single direction only though it is by no means the only approach permitted by the model. An ordering on regions is by necessity global so careful design is required to avoid global synchronisation either in the creation of new regions or the evaluation of the

ordering between regions.

Preventing cyclic relabelling is necessary for liveness of the relabelling process: it ensures that for a finite graph size and a bounded region creation rate, the relabelling process will successfully terminate, i.e. achieve usefully dead regions. Proving completeness requires not only liveness of relabelling but a guarantee that all unreachable objects will eventually be moved into usefully dead regions; different collectors ensure this property in different ways and it forms one of the constraints on collector construction.

Having shown how the relabelling processes can form usefully dead regions, the only remaining component in the creation of a garbage collector is some mechanism to detect termination of relabelling and therefore isolation of regions: Termination Detection.

### 3.3.3 Distributed Termination Detection

Having determined that lack of work available for a relabelling process implies the isolation of a region, a method is required to detect the termination of the relabelling process. The Surf model uses the concept of Distributed Termination Detection (DTD) to detect termination of the relabelling process, which requires that the relabelling process fit the DTD system model.

Before presenting a description of how DTD fits into the abstract model, two useful published definitions of DTD, the classic [93] and Doomsday [63], are reviewed so that the terminology and concepts therein may be used.

#### 3.3.3.1 Classic Distributed Termination Detection

This section describes the classic DTD problem, wherein the DTD algorithm (DTDA) exists to determine when a computation has terminated. It contains the following concepts:

**Process** Each physical site contains a process, which may be passive or active: $\mathbf{Passive}(S_P^k)$ or $\mathbf{Active}(S_P^k)$ each being a predicate on the state of the process at that site,

**Comms** Processes communicate via message passing which may be synchronous or asynchronous, depending on the DTDA used

**Active** Active processes may perform an application event (ignored by DTD model), send a message, receive a message or become passive: $\mathbf{Active}(S_P^{k-1}) \Rightarrow E_P^k \in \{send, recv, passive\}$

**Message** The means by which an active process can make a previously-passive process active.

**Passive** Execution of the $passive$ event by the computation makes a process passive; this is the only way a process may become passive:
$\left(E_P^k = passive\right) \Rightarrow \mathbf{Passive}(S_P^k)$
The only permitted event for a passive process is the receipt of a message, i.e. there exists no other permissible events that are not $recv$:
$\mathbf{Passive}(S_P^{k-1}) \Rightarrow \left(\neg\exists E_P^k \mid E_P^k \neq recv\right)$
and receipt of a message makes a process active:
$\left(E_P^k = recv\right) \Rightarrow \mathbf{Active}(S_P^k)$

**Term** The computation is defined as terminated (a global stable property) when all processes are passive and there are no messages in flight and therefore no more possible events in the system:
$\mathbf{Term}(S_*^{k*}) \equiv \forall P, \left(\mathbf{Passive}(S_P^k) \wedge \neg\exists E_P^{k+1}\right)$

The above defines the Distributed Termination Detection Problem using the system model and syntax of Section 3.1; i.e. a class of application behaviour. A valid Distributed Termination Detection Algorithm (DTDA) is an algorithm which can detect termination of a computation within finite time. It does so by executing a $decide$ event on any site within the system within finite time and must not execute $decide$ before termination is reached.

$$\mathbf{Term}(S_*^{k*}) \rhd decide_P$$

i.e., once the system is terminated, a decide event must be executed somewhere and it must be executed after the state in which termination is reached.

#### 3.3.3.2 Doomsday

The Doomsday model of DTD is presented here because the terminology it contains is used to describe and prove the correctness of the Surf abstract model of garbage collection. The use of Doomsday terminology does not require that Doomsday style DTDAs must be used when instantiating the model; wave-based DTDAs are equally permissible.

Doomsday expresses the DTD problem explicitly as a diffusing computation instead of communicating processes and provides a constructive approach to solving the problem. This approach generates only non-wave-like DTDAs but the diffusing approach is sufficient to describe the behaviour of the relabelling process. In Doomsday:

**Jobs** Each computation is referred to as a Job, $J$; Jobs may be spontaneously created at any site, making that the Job's home site, $H = \mathbf{Home}(J)$

**Tasks** Each Job consists of multiple Tasks, spread across a number of sites:
$T_P^n \in J$

**Location** At any given time (state), each task has a location site; the task state is part of that site's state:
$\mathbf{Site}(T_P^n) = P$
or a task may be in transit, contained in a message:
$\mathbf{Site}(T^n) = \circ$

**Birth** A Job's home site may spontaneously create tasks of that Job, i.e. there exist events at the home site wherein a task that does not exist before the event will exist after the event:
$\exists E_H^k = birth(T^n)_H \mid \left(S_H^{k-1} : \{\neg\exists T^n\} \wedge S_H^k : \{\exists T_H^n\}\right)$
Tasks may create new tasks at their current site
$\exists E_P^k = birth(T^m)_P \mid \left(S_P^{k-1} : \{\exists T_P^n \wedge \neg\exists T^m\} \wedge S_P^k : \{\exists T_P^m\}\right)$

**Death** Tasks may spontaneously cease to exist:
$\exists E_P^k = death(T^n)_P \mid \left(S_P^{k-1} : \{\exists T_P^n\} \wedge S_P^k : \{\neg\exists T^n\}\right)$

**Migration** Tasks may migrate, which is a two-step process involving leaving one site then arriving at another:
$\left(\exists E_P^{k_P} = send(T^n)_P\right) \wedge \left(\exists E_Q^{k_Q} = recv(T^n)_Q\right) \wedge$
$(send(T^n)_P \prec recv(T^n)_Q)$
$\left(S_P^{k_P-1} : \{\exists T_P^n\} \wedge S_P^{k_P} : \{\neg\exists T_P^n\}\right) \wedge \left(S_Q^{k_Q-1} : \{\neg\exists T_Q^n\} \wedge S_Q^{k_Q} : \{\exists T_Q^n\}\right)$
This is not directly part of the Doomsday model, rather it is part of the Doomsday system model

**Decide** If a Job has no extant tasks, the Home site may $decide(J)$ that the Job is terminated:
$\neg\exists T^n \in J \quad \widetilde{\rhd} \quad decide(J)_H$
Safety requires that decide only occur if there is a causal relationship between all tasks ceasing to exist and the state in which decide occurs:
$\exists S_H^k \mid S_H^k \prec decide(J)_H \Rightarrow \forall T^n \in J,\, death(T^n) \prec S_H^k$
That final causal relationship between task deaths and decision of termination is provided by the transmission of control messages that establish causal relationships between states that would not exist due to the computation alone. The DTDA may send such messages in response to the $birth$ and $death$ events of each Task.

The Doomsday Model of DTD requires that the creation of a remote task is witnessed by an existing task, i.e. the Birth has **cover**. As long as the DTDA can observe Birth and Death events and it can be guaranteed that no Birth events

occur spontaneously on a site other than the Home site then any Doomsday-derived DTDA may be used. The Doomsday system model specifies the use of message passing and states that tasks must migrate inside messages therefore when mapping GC tasks onto Doomsday below, it is necessary to show not only that the Doomsday requirements (Birth, Death, Cover) are met but also that the motion of tasks conforms to the Doomsday system model. To this end, a Migration entry is added to each mapping to show how the motion of a Task is performed by an underlying message of the system model of Section 3.1. Since the system model used here is compatible with Doomsday, showing that Task Migration is performed by messages is a necessary step in showing that the mapping to Doomsday is correct.

#### 3.3.3.3 Comparison of DTD Models

This section shows how the two DTD models are congruent: they solve the same problem though they use different terminology. The Doomsday approach to constructing DTDAs results only in non-wave-based algorithms but that does not reduce its descriptive power with respect to mapping computations onto the DTD problem.

The two models, though expressed differently are substantially similar because any application that can be mapped into one description can be mapped into the other, as follows:

| **Classic** | **Doomsday** |
|---|---|
| Computation | Job |
| Active Process | Site with non-zero tasks |
| Idle Process | Site with zero tasks |
| Message | Migrating Task |

The primary difference between the two models is that Doomsday explicitly includes the concept of a Home site and permits the Home site to spontaneously create tasks at any time, thereby allowing termination to be detected safely only at the Home site. The Classic model does not include this restriction and permits an additional class of solutions to the Termination Detection problem based on wave algorithms.

For the purposes of this model, the Job abstraction permits the description of multiple overlapping DTDA executions (Jobs) within a single system. This aids clarity since it explicitly states that the scope of DTDA execution may be smaller than the entire system and it permits detection of the termination of multiple concurrent computations (Jobs) within the system.

#### 3.3.3.4 Application of Termination Detection

By defining a Job for some combination of regions (typically just one region) and appropriate inter-region pointers as Tasks of that job, termination of the job is equivalent to isolation of the region(s). A mapping of the relabelling process into the Doomsday model of DTD is provided in the Proof section below, showing how termination of relabelling may be detected using the DTD framework. Also provided here is a mapping of the pointer tracking problem (remembered sets) into the DTD model

### 3.3.4 Remembered Sets & Pointer Tracking

A Remembered Set (remset) is a mechanism whereby each object keeps track of the pointers that refer to that object. Reference Counting is a degenerate form of remsets suitable only for uniprocessor systems in that the total number of references to each object is recorded. For the purposes of this model, the reference counts for each object must be broken down by the labels applied to the object containing the references, in other words the reference counts are a compound structure, indexed by source region.

Remsets therefore detect trivial unreachability of objects and also serve as a source of information as to the presence of inter-region pointers. The message passing mechanism used to correctly maintain a remembered set is referred to as a **pointer tracking protocol**. Table 1 summarises the mapping of the remset / pointer tracking problem to the DTD abstraction; this mapping means that any DTDA may be used to solve the remset problem and thereby determine when an object is trivially unreachable, i.e. there exists no pointers to that object.

For the purposes of the Surf model, remsets are used to detect not only trivial unreachability of objects but also the existence, or lack thereof, of references from particular regions. The data structures maintained at the home site (the location of the object in question) contain safe estimates of the existence or otherwise of tasks (pointers) in other parts of the system; it is necessary to partition these data structures by the region in which those tasks exist. By partitioning the data structures, it is now possible to determine which regions contain pointers to particular objects and therefore the existence (or otherwise) of inter-region pointers; this knowledge is useful (but not sufficient, see the Proof) in discovering the presence of relabelling work and it also defines the reachability of each object from other regions.

The reachability of an object from another region implies that the region is not usefully dead. Surf operates by forming and detecting usefully dead regions; therefore pointer tracking is used as a source of information as to the existence

| **DTD Concept** | **Meaning wrt Remembered Sets** |
|---|---|
| Job | Direct reachability of each object is a Job; the home site of the Job is the location ($P$) of the object in question: $J \models x_P$ |
| Task | Each pointer to the object of interest is a Task of that object's Job: $T^n_P \in J \models y.x$ |
| Birth | The first pointer to an object is created when the object is created (definition of object creation), all subsequent pointers to that object may be created only as copies of existing pointers to the object: $birth(T^n)^k_P \models send(y.x \rightarrow Q)^k_P$ |
| Migration | Pointers may travel across the network in messages, which is modelled by the migration of tasks: $send(T^n)_P \models send(y.x \rightarrow Q)^k_P$ <br> $recv(T^n)_Q \models recv(P \rightarrow z.x)^k_Q$ |
| Death | Each task ends when the relevant pointer is erased: $death(T^n)_P \models minus(y.x)^k_P$ |
| Termination | The lack of pointers to an object is equivalent to the lack of Tasks in (termination of) the Job: $\neg\exists T^n \in J \models \neg\exists y.x$ |

Table 1: Remembered Sets mapped to DTD

or otherwise of inter-region pointers and therefore permits determination of whether a region is usefully dead.

Doomsday-derived algorithms are particularly suitable for solving the remset problem because the nature of pointers being copied throughout the store resembles a diffusing process. Safety of the DTD implies that an object will never erroneously decide that it is not the target of any pointers and liveness of the DTD implies that every trivially dead object ($\mathbf{TrivialDead}(x) \equiv \neg\exists y.x$) will decide within finite time that it is not reachable from other objects.

It should be noted that remembered sets are not a necessary component of an instantiation of the model, merely that they provide a mechanism for detecting trivial isolation and the presence of inter-region pointers. Remembered sets provide information as to the reachability of a single object from other regions; this reachability information is considered in aggregate to determine when a region becomes usefully dead. Some instantiations of the model use degenerate forms of remembered sets: constraining remembered set operation makes an implementation simpler at the cost of greater conservatism and therefore poorer timeliness.

### 3.3.5 Summary of Abstract Model

Having shown that a relabelling process can form usefully dead regions and asserted that the problem of relabelling termination is solvable with a DTDA, the conclusion is that the only necessary components for construction of a safe and complete garbage collector are:

- a useful Relabelling Process to form usefully dead regions,
- a DTDA to detect the presence of inter-region pointers,
- a DTDA to detect termination of the relabelling process and therefore region isolation, and
- some constraints on their combination and application.

The constraints are derived below in the Proof; they are the conditions which must hold true for the proofs of safety and completeness to be correct. An instantiation of the model that fulfils the constraints stated below will result in a safe and complete garbage collector.

It should be noted very carefully that there are typically **two** DTDAs present in a given instantiation of the model: one performing remembered set / pointer tracking functionality (the **lower level**) and one detecting the isolation of regions (the **upper level**). It is important not to confuse the two DTD mappings as

they have different job scopes and detect different properties of the graph. The two DTDs interact: the lower level acts in many cases as the task-migration mechanism (see Section 3.3.3.2) for the upper level and this is described further below. An unterminated job (object reachable from another region) of the lower level DTDA is a task of the upper level DTDA; when no such tasks exist, there are no inter-region pointers to that region.

There is an important property of DTDAs as applied to pointer tracking (lower level) that enables the use of a DTDA to detect region isolation (upper level): when a pointer is copied, a birth message is sent — perhaps indirectly — to the home site of the referred-to object. This message conveys the notion of liveness from the task-creation site to the home site and is the mechanism of task migration in the upper level. Mapping the isolation problem (upper level) to Doomsday requires that tasks may migrate only in messages; this behaviour of the lower level DTDA is the means by which tasks of the upper level are guaranteed to be transported in messages and thereby attain compatibility with the Doomsday system model.

It must be carefully noted that the Surf model describes only the abstract process whereby collectors make progress and detect termination of that progress. The model does not specify exactly how the collector is implemented and requires a range of additional protocols and mechanisms outside the essence of the collection algorithm described by Surf. Surf defines requirements on these mechanisms in order that they be correct but different implementation choices will have an impact on collector performance; these requirements are derived in the Proof (Section 3.4 immediately below) and a summary of the additional protocols is provided in Section 3.5.

## 3.4 Proving Safety and Completeness of Surf

The correctness proof for Surf is founded on formally describing the actions of the model in terms of DTD and using the knowledge that a particular DTDA is correct to show that an instantiation of the model is correct. The bulk of this proof consists of showing how the abstract model fits the DTD description of how a computation must behave and deriving constraints on instantiations to ensure that the validity of the mapping to DTD remains valid.

This proof is in multiple parts for clarity: the **relabel-source** and **relabel-target** approaches are considered separately; each is described first without a concurrent mutator then the permissible events due to mutator activity are introduced and their effects analysed. The reason for introducing concurrent mutator activity later is that the proofs are clearer in the case where the object graph is static;

the concurrent proofs are built on the static proofs by admitting the possibility of mutator events. It is possible to use the static proofs to construct garbage collectors that do not require interaction with a mutator, i.e. stop-the-world collectors.

Safety is proven first for each case (relabel-source and relabel-target) and then a set of requirements for providing completeness are derived from the fundamental requirement of completeness that dead objects eventually enter a usefully dead region.

Having shown how Surf forms safe and complete collectors and derived a set of requirements that ensure the proof is valid with respect to an instantiation, some approaches present in the published literature are investigated as to whether they fulfil the model requirements.

### 3.4.1 Termination Detection and Relabel-Target

This section describes the mapping of the region isolation / relabelling termination problem into the Doomsday DTD framework on the assumption that **target** objects of inter-region pointers ($x$ in $y.x$) are relabelled. This mapping constitutes the core of the proof of correctness for relabel-target instantiations: fulfilling the requirements derived in this section ensures that an instantiation represents a safe mapping to DTD and therefore correct operation of the resulting garbage collector.

The first step in the mapping is to have knowledge of which objects are the targets of inter-region pointers and this task is performed by the pointer tracking/remset DTD as described above. Having constructed accurate remsets and described some necessary properties of their behaviour, it becomes possible to map objects with non-zero inter-region remsets to DTD tasks and thereby determine isolation of a region. Note that the mapping here uses target objects as tasks of the relabelling-termination DTD; that choice means that termination will allow the algorithm to infer isolation of a region. The opposite approach of using source objects as tasks would permit the algorithm to infer only when a region contains no pointers to another, which is not useful result in the general case.

The approach of relabelling the target of inter-partition pointers implies that this represents the class of **forward tracing** collectors, i.e. collectors where the progress of relabelling activity travels in the same direction as the pointers in the graph. If cyclic relabelling (i.e. object $x$ goes from region $\mathfrak{L}$ to $\mathfrak{M}$ to $\mathfrak{L}$), then the relabelling process will not be cyclic in the face of cyclic data structures because an object can enter or leave a particular region only once. Under the relabel-target approach, acting on a task in a job results in a reduction of the number of objects in that job because the object representing the task is moved to a different region.

| DTD Concept | Meaning wrt Relabel-Target |
|---|---|
| Job | The reachability of each region is a Job:<br>$J \models \mathfrak{L} \equiv \{x \mid label(x) = L\}$ |
| Task | Each object reachable via an inter-region pointer constitutes a Task of the region (Job) in which it exists:<br>$T_P^n \in J \models x_P \mid \exists y.x \wedge label(y) \neq L$ |
| Birth | Object becoming reachable from another region by relabelling (mutator activity disallowed); can be as a result of an object entering the region of interest while being reachable from other regions:<br>$birth(T^n) \models relabel(x \rightarrow L) \wedge \exists y.x \mid label(y) \neq L$<br>or as a result of an object leaving the region of interest while containing what were intra-region pointers:<br>$birth(T^n) \models relabel(x \rightarrow M) \wedge L \neq M \wedge \exists x.y \mid$<br>$label(y) = L$<br>These are two of the four cases discussed in Section 3.3.2; the other two cases result in creation of tasks for other regions and are the transpose of these two cases, i.e. are described by applying this mapping to other regions. |
| Death | Object becoming unreachable from another region; either by relabelling or (later) by a mutator erasing a pointer |
| Migration | Birth messages of pointer tracking mechanism. |
| Termination | Region becomes usefully dead |

Table 2: Relabel-Target mapped to DTD

The approximate process is that a candidate region will shrink until it contains no live objects, at which point it is reclaimed.

#### 3.4.1.1 Static Proof

Given the presence of accurate remsets, each object is aware of the presence of inter-region pointers to itself and will decide the point in time when no such pointers exist. This information provides the local knowledge required to determine when a region is usefully dead, since this occurs when no objects in the region are the target of pointers from outside the region. What remains is to map the determination of isolation of a whole region (not merely a single object as performed by Remembered Sets) onto the DTD problem; this mapping is listed in Table 2.

To make this mapping correct, it is necessary to ensure that allowable events at any given time do not violate the primary constraint of the DTD model, namely that tasks may not be spontaneously created at sites other than the home site. Ignoring the mutator for this portion of the proof, it remains to be shown that the Birth events may only occur in the presence of an existing Task; in the second case (object leaving region) this holds true but in the first case (object entering region) it does not.

Therefore the mapping is not correct without further modification because a reachable object can be imported into an otherwise unreachable region; this is known as the **unwanted relative problem**. The "unwanted relative problem" term, though defined in the DMOS literature, is generalised by this thesis to all relabel-target collectors conforming to the Surf model that permit objects to be relabelled into candidate regions; the term will be further generalised to the relabel-source approach in Section 3.4.2.1. To solve the unwanted relative problem, a number of approaches may be taken, each of which amounts to **making the DTDA safely aware of the newly created task**. To do this requires synchronisation between the DTDA and relabelling process, thereby destroying the non-interfering properties of DTDAs but this loss seems unavoidable.

The synchronisation required will depend on the nature of the DTDA chosen for a particular instantiation of the model and two example approaches are described briefly here for illustrative purposes; they do not define the model. The first approach applies to use of a probing wave DTDA wherein relabelling is halted while the wave DTDA is executing. Should the wave not find termination, relabelling is re-started.

The second approach applies to the use of any Doomsday DTDA; what is required is to satisfy the cover rule by providing a witness to the birth. The relabelling process sends a message to the region's home site requesting a witness, the witness is created at the home site, travels to the site of relabelling, observes the task creation and then dies or migrates back to the home site. Should termination be detected before the request for witness arrives, the request will be denied and that information conveyed to the site requesting relabelling; termination implies isolation therefore reclamation of the region and so no further need to relabel an object into that region.

Where an object leaves the region of interest, it will do so only where an inter-region pointer to that object existed and therefore the object in question constitutes a task of the DTDA. When the object is relabelled, any intra-region pointers it contains will cause other objects in the region to become tasks of the region DTD; these new tasks should be considered as being created at the relabelling site (satisfying the Doomsday cover rule) and then migrating to the

newly-externally-reachable objects in the form of pointer tracking messages. This is an instance of the DTD control messages of the lower layer (pointer tracking) acting as task-bearing agents in the higher layer (region isolation) DTD as described above for Remembered Sets. The region isolation DTDA is not aware of the migration; this explanation involving migration serves only to show how the cover rule is satisfied and that task migration is performed by the transmission and receipt of a message.

Therefore, with the synchronisation between relabelling and termination detection described above, the task of detecting region isolation can be safely mapped to the DTD problem: a collector constructed in this manner with a useful relabelling process and no concurrent mutator will be safe and complete.

#### 3.4.1.2 Concurrent Proof: Safety

The proof of relabel-target garbage collection safety with concurrent mutator(s) is identical to the static proof above with the exception that additional events are permitted: the mutator may copy, erase and transmit pointers between sites. Remset behaviour was shown to be correct with concurrent mutator activity above (Section 3.3.4); what remains is to show the correctness of the region isolation DTD with mutator activity, thereby guaranteeing that regions will not be erroneously detected as isolated due to mutator activity.

Pointer erasure ($minus(y.x)_P$) may occur at any time and may result in an object becoming unreachable from other regions and therefore the death of a region-isolation task. Since a task may die at any time, this is permissible.

A pointer may be duplicated for transmission: $send(z.x \rightarrow Q)_P$; if the pointer is inter-region ($label(x) \neq label(z)$) then $x$ already represents a region-isolation task so no new task is created, otherwise the pointer is intra-region. For the event to be allowed, the mutator must hold a pointer to $z$, which represents a task ($\exists r.z \mid label(r) \neq label(z)$) or the region is not a candidate ($label(r) = label(z)$). The duplication of the pointer-to-$x$ at $z$ represents task birth at $z$ and is witnessed/covered by the task already at $z$; the newly created task of the upper level DTDA is carried to $x$ by the pointer tracking (lower level) DTDA control messages.

The reception of a message containing a pointer ($recv(P \rightarrow y.x)_Q$) is merely the completion of the pointer duplication process that began with $send$. The pointer so received will be stored in an object and may result in a new inter-region pointer-to-$x$ but this was accounted for during the $send$ event.

Object creation results in an object labelled either in the same region as the object that refers to it, a region of its own or a non-candidate region. The new object is directly reachable from some object that is reachable from a root:

$create(y.x)^k_P \Rightarrow S^{k-1}_P : \{\exists r.y\}$. Reachability is unchanged for all existing candidate regions; no tasks are created or destroyed.

Having shown that all allowable mutator events conform to the DTD model as it relates to the region-isolation DTD mapping, it is clear that relabel-target garbage collection is safe with the following constraint: non-candidate regions may not become candidates without synchronisation with the DTD since task creation in non-candidate regions need not conform to the DTD rules.

#### 3.4.1.3 Concurrent Proof: Completeness

Completeness requires that mutator events do not interact with the relabelling process in such as way as to prevent progress towards a state wherein live and dead objects are separated along region boundaries. More specifically, all unreachable objects must eventually be labelled as members of a usefully dead region, which defines the **primary completeness requirement**:

$$\mathbf{Dead}(x) \rhd x \in \mathfrak{L} \wedge \mathbf{UsefullyDead}(\mathfrak{L})$$

For garbage to be reclaimed, garbage objects must be inside usefully dead regions because the reclamation of such regions is the mechanism whereby garbage is detected: a matter of definition regarding collector operation under this abstract model. Fulfilling this condition requires that:

- the system must have at least two regions: one for live and one for dead objects,
- initial conditions are such that dead objects may not perpetually remain in a live region, i.e.
  - dead objects must eventually not be in the same region as the root, and
- the relabelling process must retain liveness in the face of mutator interaction.

The conditions above are justified as follows:

**Two Regions**

A minimum of two regions are required since any fewer will require that the root is a member of the only region, making it a non-candidate and not usefully dead,
$r \in \mathfrak{L} \Rightarrow \neg candidate(\mathfrak{L})$
therefore with a single region, no candidate regions are available to become usefully dead and reclaim garbage.

**Initial Conditions**

Should an unreachable portion of the graph reside entirely in a region containing the root, no relabelling work will exist to separate the live and dead objects. Therefore, any region containing garbage objects must eventually become a candidate region to permit the reclamation of that garbage since the garbage will never be removed from the region in question, the reachable objects must be removed instead. This requirement is effectively a restatement of the primary completeness requirement above but taking into account the case where an entire garbage component may lie entirely within a region and therefore never leave the region.

$\mathbf{Dead}(x) \wedge x \in \mathfrak{L} \rhd candidate(\mathfrak{L})$

**Relabelling Liveness**

This requirement is trivial in meaning; should liveness of relabelling be lost, the system may not ever generate usefully dead regions. Proving that a given collector exhibits this property is more difficult and is covered below.

$liveness \Rightarrow (candidate(\mathfrak{L}) \rhd \mathbf{UsefullyDead}(\mathfrak{L}))$

Combining the definition of all three of these conditions results in a proof that completeness is achieved:

$$\begin{array}{l} \left( \begin{array}{c} (\mathbf{Dead}(x) \wedge x \in \mathfrak{L}) \rhd candidate(\mathfrak{L}) \\ \wedge \\ candidate(\mathfrak{L}) \rhd \mathbf{UsefullyDead}(\mathfrak{L}) \end{array} \right) \\ \Rightarrow \mathbf{Dead}(x) \rhd (x \in \mathfrak{L} \wedge \mathbf{UsefullyDead}(\mathfrak{L})) \end{array}$$

The initial conditions requirement implies that all regions in which garbage exists must eventually become candidate regions. This in turn implies the repeated creation of new regions as old regions are reclaimed so as to fulfil the two-regions requirement. In a system with exactly two regions, the creation of a new region on the reclamation of an old one implies that the root will be relabelled into the newest region, leaving all other objects initially in the surviving old region, making it a candidate region. Safety requires that this operation be synchronised with the DTD (Section 3.4.1.2), thereby implying that the collector operates in distinct *phases*.

Where more than two regions are present, it is possible to avoid changing non-candidate regions into candidates and thereby avoid synchronisation.

Maintaining liveness of relabelling requires that mutator activity not prevent the discovery of inter-region pointers; the exact manner in which this is ensured

depends on whether the system inspects pointers or remsets to determine the presence of work for the relabelling process. Since the mutator by definition cannot access dead objects, it is not possible for the mutator to affect the relabelling of an unreachable component into a single region; the only possible effects of the mutator pertain to the relabelling of live objects. Liveness also implies the prohibition of infinite cyclic relabelling.

### 3.4.2 Termination Detection and Relabel-Source

This is the second section to describe a mapping of the abstract model to DTD; it serves exactly the same purpose within the proof as Section 3.4.1 insofar as showing correctness in mapping the abstract model to DTD provides correctness of each instantiation.

Described here is the back-tracing approach to collection: the existence of an inter-region pointer implies that the **source** of the pointer (the object containing the pointer, i.e. $y$ in $y.x$) will be relabelled into the region that contains the target of the pointer. These are so-called back-tracing collectors because the relabelling activity progresses in the opposite direction to the pointers in the system, which is the converse of the approach described above (Section 3.4.1) wherein the targets of inter-region pointers are relabelled.

A very similar mapping of the problem onto the Doomsday system model is provided to that above, namely that the isolation of each region is a Job and pointers into the region in question are represented by Tasks. The primary difference between the two is due to the direction of relabelling: relabel-target requires synchronisation when an object enters the region while relabel-source requires synchronisation when an object leaves the region.

Under this approach, acting on a task of a region results in an increase in the number of objects in that region; instead of a region shrinking until it contains only garbage objects, the common case is that it grows until it contains the entirety of all components that are partially within that region. Should the root object be relabelled into the region, it is no longer a candidate region and may not be reclaimed.

#### 3.4.2.1 Static Proof

The general approach to collection is the same as for relabel-target: detect the isolation of a region using a DTD. For the same reasons as the relabel-target collectors, tasks model inter-region pointers into a region of interest instead of pointers out of the region. Once again, Remembered Sets (Section 3.3.4) provide a mechanism whereby each object is made aware of the inter-region pointers of

| DTD Concept | Meaning wrt Relabel-Source |
|---|---|
| Job | The reachability of a region is a Job<br>$J \models \mathfrak{L} \equiv \{x \mid label(x) = L\}$ |
| Task | Each object reachable via an inter-region pointer constitutes a Task of the Job (region) in which it exists<br>$T_P^n \in J \models x_P \mid \exists y.x \wedge label(y) \neq L$ |
| Birth | Object becoming reachable from another region by relabelling; can be as a result of an object entering the region of interest while being reachable from other regions:<br>$birth(T^n) \models relabel(x \rightarrow L) \wedge \exists y.x \mid label(y) \neq L$<br>or as a result of an object leaving the region of interest while containing what were intra-region pointers:<br>$birth(T^n) \models relabel(x \rightarrow M) \wedge L \neq M \wedge \exists x.y \mid label(y) = L$<br>These are two of the four cases of Section 3.3.2; the other two cases result in creation of tasks for other regions and are the transpose of these two cases, i.e. are described by applying this mapping to other regions. |
| Death | Object becoming unreachable from another region; either by relabelling or (later) by a mutator erasing a pointer |
| Migration | Tasks in messages |
| Termination | Region becomes usefully dead |

Table 3: Relabel-Source mapped to DTD

which it is a target. The mapping to the Doomsday system model is listed in Table 3.

It should be noted that this mapping is so far identical to that of the relabel-target approach of Section 3.4.1; the only differences are in the justification for correctness of the mapping (ensuring that the cover rule is satisfied) and the synchronisation required between relabelling processes and DTDAs. Where an object enters a region, a task exists for the region due to the inter-region pointer from the object to be relabelled. This task spawns and migrates (in a network message) to the relabelled object; if that object is the target of inter-region pointers after relabelling then the task it received remains. The cover rule is therefore satisfied for the job that the object is relabelled into.

Where an object is relabelled out of a region, there is no guarantee that the object represents a task. Should it contain intra-region pointers, tasks must be

created in the job to deal with these pointers that now exist in another region and the cover rule is not automatically satisfied in this case. Therefore, a safe GC requires synchronisation between the relabelling process and the DTD of the Job which an object leaves.

This mapping shows a symmetry between the relabel-target and relabel-source approaches:

- relabel-target requires synchronisation for the job that an object enters (unwanted relative), and
- relabel-source requires synchronisation for the job that an object leaves (victimised relative).

By describing the Surf model in terms of a mapping between relabelling process termination and the DTDA system model, specific cases are identified where synchronisation is required. One of these specific cases is the unwanted relative problem as described by DMOS, therefore the Surf model contains a generalisation of the unwanted problem across relabel-target collectors and defines the symmetric case for relabel-source collectors.

The location where synchronisation is required depends on the direction of tracing over the graph and this synchronisation may be avoided in either case by careful design of the relabelling process to ensure that cases requiring synchronisation never occur.

The mapping provided here shows that isolation detection of a region under a relabel-source relabelling scheme is safe insofar as the use of a DTDA to detect region isolation will not erroneously claim a (live) region to be isolated if relabelling is synchronised with the job that an object is leaving due to relabelling.

#### 3.4.2.2 Concurrent Proof: Safety

The proof of safety of the relabel-source approach is identical to that given for the relabel-target approach (Section 3.4.1.2) because the mappings are identical. The static proof above shows that the mapping is safe in the face of relabelling events; the concurrent proof of safety of relabel-target shows how the identical mapping is safe in the face of mutator events.

#### 3.4.2.3 Concurrent Proof: Completeness

Proving completeness in the face of mutator activity requires a different approach than given for relabel-target because the collector proceeds across the object graph in the opposite direction. Instead of pulling potentially live objects out of a region until it is usefully dead, the relabel-source approach grows regions from

a suspected garbage object until all other objects that can reach the originally-suspected object are included in the region. Should the region still remain a candidate at the completion of this growth process, the region is garbage.

The primary completeness requirement remains unchanged from relabel-target: all garbage objects must eventually reside in a usefully dead region so that they may be reclaimed:

$$\mathbf{Dead}(x) \rhd x \in \mathfrak{L} \wedge \mathbf{UsefullyDead}(\mathfrak{L})$$

Satisfying the primary completeness requirement implies that the following conditions must hold:

- the system must have at least two regions: one for live and one for dead objects,
- the relabelling process must retain liveness in the face of mutator interaction,
- garbage must be suspected as such, and
- candidate regions must be permitted to grow to completion before objects are removed from them.

The meaning of these requirements is defined as follows:

**Two Regions**

The first requirement is obvious: if only a single region is present, it is not a candidate and cannot be reclaimed so no garbage may be reclaimed. There may also be no progress (relabelling) with only a single region. This is the same situation as relabel-target.

**Relabelling Liveness**

The second requirement is the same as given for relabel-target: mutator activity may not be permitted to hide pointers from the relabelling process. The proof of safety above shows that a region will not be erroneously reclaimed while reachable; liveness of relabelling and therefore completeness require that the relabelling process be capable of observing the pointer(s) constituting tasks of any given region.

**Suspicion**

Completeness requires that every dead object or some other dead object reachable therefrom must eventually be suspected after they

become dead:

$$\mathbf{Dead}(x) \rhd \left( \begin{array}{l} suspected(x) \vee \\ (suspected(y) \wedge \mathbf{Dead}(y) \wedge \mathbf{Reachable}(x, y)) \end{array} \right)$$

A stronger and simpler to evaluate version is the **strong suspicion guarantee**, i.e. ensure that every dead object is suspected:

$$\mathbf{Dead}(x) \rhd suspected(x)$$

**Region Growth**

If a dead object is the suspect, the region that grows from it must be dead because

$$\mathbf{Dead}(x) \Rightarrow \neg\exists y.x \mid \mathbf{Live}(y)$$

Growth of the region without bound (i.e. no new regions are created until no relabelling progress is available) will result in a dead region forming around $x$ once the extent of that garbage component has been discovered because

$$\mathbf{Reachable}(x, suspect)$$

The reason for requiring that candidate regions grow to completion is to ensure that dead regions become usefully dead. A dead yet not usefully dead region is referred to only by pointers from dead objects:

$$\mathbf{Dead}(\mathfrak{L}) \wedge \neg\mathbf{UsefullyDead}(\mathfrak{L}) \Rightarrow (\forall z.x,\ (x \in \mathfrak{L} \wedge z \notin \mathfrak{L}) \Rightarrow \mathbf{Dead}(z))$$

therefore a dead region will always grow into a usefully dead region if given the opportunity.

The detection of isolation and hence reclamation of regions implies that regions must be created to satisfy the first ($\geq 2$ regions) requirement.

The combination of the suspicion guarantee and a guarantee that a region is permitted to grow to completion will ensure that every dead object is eventually a member of a usefully dead region and therefore that the collector is complete. The suspicion guarantee ensures that every dead object will eventually be in a dead region:

$$\mathbf{Dead}(x) \rhd x \in \mathfrak{L} \wedge \mathbf{Dead}(\mathfrak{L})$$

While unlimited growth of a region ensures that dead regions become usefully dead:

$$\mathbf{Dead}(\mathfrak{L}) \rhd \mathbf{UsefullyDead}(\mathfrak{L})$$

Therefore the collector fulfils the primary completeness requirement:

$$\mathbf{Dead}(x) \rhd x \in \mathfrak{L} \wedge \mathbf{UsefullyDead}(\mathfrak{L})$$

As a counter-example to show the necessity of the region-growth requirement, consider the case where a dead object contains a pointer to other dead objects and also a live object:

$$(\exists z \mid \mathbf{Dead}(z)) \wedge (\exists z.y \mid \mathbf{Live}(y)) \wedge (\exists z.x \mid \mathbf{Dead}(x))$$

Should $z$ be relabelled into the region containing $y$ from the region containing $x$, it will prevent the isolation of the region containing $x$. Therefore the region containing $x$ must be permitted the opportunity to grow to completeness and be reclaimed before $z$ is removed from that region. This requirement stems from the assumption that cyclic relabelling is prohibited; the prohibition implies that $z$ may not rejoin $x$'s region.

Permitting finite cyclic relabelling behaviour removes the growth requirement but it becomes much more difficult to prove liveness, let alone ensure efficiency. The model admits the possibility of finite cyclic relabelling but the constraints on its application to satisfy liveness are not explored in this thesis.

### 3.4.3 Approaches to Fulfilling Requirements

This section examines published techniques in terms of the requirements for safety and completeness derived in the proof above.

One requirement for completeness is that mutator activity not be permitted to hide work from the garbage collector, therefore some pointers must be considered work after they are erased if the GC is to terminate. One approach to ensuring liveness of relabelling in the face of mutator activity is known as **snapshot at the beginning** (SATB), wherein relabelling work (an inter-region pointer) that exists at some point in time (state) is considered to continue to exist for all subsequent states despite the mutator's erasure of the pointer in question. It is so named for its use in collectors exhibiting phases of operation wherein any object live at the beginning of a phase will survive that phase and objects becoming unreachable in a phase will not be reclaimed until the next phase. Generalising SATB to a non-phased collector, i.e. one with three or more regions results in every object that is reachable from outside a region at any time being relabelled out of that

region; the region that reclaims an object is the region that the object entered after becoming unreachable:

$$S_P^k : \{\mathbf{Live}(x) \wedge x \in \mathfrak{L}\} \Rightarrow S_P^k \prec relabel(x \rightarrow M)_P^l$$
$$S_P^{l-1} : \{\mathbf{Dead}(x)\} \wedge \exists relabel(x \rightarrow M)_P^l$$
$$\Rightarrow S_P^m : \{x \in \mathfrak{M} \wedge \mathbf{UsefulDead}(\mathfrak{M})\} \wedge S_P^l \prec S_P^m$$

SATB is merely one approach to guaranteeing relabelling liveness; it is simple to implement but overconstrains the solution somewhat by ensuring that every inter-region pointer representing work is preserved as work despite the erasure of that pointer. It is necessary only to guarantee that at least one inter-region pointer to each component is considered to represent relabelling work.

An **incorrect approach** is to inspect remsets in search of inter-region pointers; the reason for incorrectness is that remsets discard a significant quantity of information. Consider a reachable cycle being continuously traversed by the mutator: a pointer to each object will be created and destroyed for each traversal of the cycle and isolation of the region containing the objects will not be detected due to the presence of these pointers. A discrete-time sampling of any object's remset is not guaranteed to observe that there is work available, therefore the repeated discrete sampling of all remsets in a region is no guarantee that the presence of work will be visible. Liveness of relabelling is therefore lost.

The fault in the remset approach is that after $send/recv$ then $minus$ events have occurred, the remset returns to its original zero-state: all knowledge that the object was reachable from outside the region is lost. The same problem applies when pointers out of a region are sampled in a discrete-time manner: there is no guarantee that the pointers will be observed if they are subject to frequent mutation.

## 3.5 Instantiating the Model

Having defined all of the components required to build a fundamental garbage collector in previous sections, this section shows a concrete process that an algorithm designer may follow to either analyse an existing collector or construct an entirely new garbage collector.

In summary, instantiating the model requires five major decisions:

- direction of relabelling progress: relabel-target (forward) or relabel-source (backward),
- configuration of regions: two or many,

- definition of the relabelling job(s),
- choice of the relabelling termination DTDA(s), and
- choice of the remset DTDA.

These five choices define a five-dimensional space in which a collector may exist; the totality of the space represents the universe of collectors represented by the Surf model and each collector occupies a point in that space. Having selected the fundamental components that define the collector, a number of requirements must be fulfilled to ensure that the safety and completeness proofs defined above are valid for the collector:

- If the collector is subject to the unwanted relative or victimised relative problem, synchronisation is required between relabelling and the region isolation DTDA.
- Relabelling liveness must be preserved in the face of mutator activity.
- The system must have at least two regions.
- For relabel-target collectors:
  - The initial conditions at the creation of a region must provide completeness.
- For relabel-source collectors:
  - The suspicion algorithm should support the strong suspicion guarantee, though other approaches are feasible, they are unproven here, and
  - Dead candidate regions must be permitted to grow to usefully dead regions.

The proof provided in this chapter is applicable to any collector that satisfies the requirements stated above, therefore a designer can use them as a checklist to quickly determine if a particular garbage collector is correct or not and to draw some broad generalisations as to the relative performance of the collector. Having defined how the collector functions in terms of the model, the collector still requires some further design work since there may be functionality and/or protocols required that are not covered by the mode.

Having shown how to fit a collector into the model, it becomes possible to analyse the behaviour of a collector in terms of how it fits into the model. The

point in design-space that the collector occupies will define its behaviour to a large extent and the means by which the safety and completeness requirements are satisfied will also have a bearing on its performance. The effects of each of the choices are discussed briefly here; examples of this analysis are presented in Chapter 4.

### 3.5.1 Direction of Progress

Relabel-target algorithms start with a seed of knowledge — the known reachability of the root — and grow that knowledge as a diffusion across the object graph. They therefore have a defined complexity and will reliably reclaim garbage within a finite number of relabelling steps.

Conversely, relabel-source algorithms begin with a hypothesis and then diffuse the hypothesis across the object graph to determine if it remains true or not. If the hypothesis was false, the work done in testing that hypothesis is likely to be lost, therefore progress of a relabel-source algorithm is dependent upon the presence of a suspicion algorithm which can reliably pick dead objects as suspects, i.e. tends to form correct hypotheses. A suspicion algorithm that always begins from a correct hypothesis will cause a relabel-source collector to always make progress but this may not be optimal progress. If the suspicion algorithm never forms a correct hypothesis, the collector will make no progress. However, Chapter 6 hypothesises that relabel-source collectors may have excellent timeliness if constructed carefully.

### 3.5.2 Configuration of Regions

A collector with only two regions implies that one region is the candidate and the other contains the root. This means that once the candidate is reclaimed, the non-candidate region must become the candidate region and for this, synchronisation is required. The synchronisation overhead is a cost not borne by multiple-region systems. Likewise, all relabel-target collectors with only two regions must operate in phases with global synchronisation between phases and they may not reclaim any garbage until the end of the phase. The reason for this is the combination of two requirements: that all garbage objects be in the candidate region at the beginning of a phase and that all live objects be in the other region at the end of the phase, therefore the algorithm must process every live object within every phase.

Conversely, collectors with multiple regions can avoid the synchronisation overhead by making the root a member of a special non-candidate region and all other regions are candidates. This means that no synchronisation is required

for non-candidates to become candidates, no phases are implied and the collector does not need to operate over the entirety of the graph to reclaim garbage. Such collectors are therefore more scalable than two-region collectors.

### 3.5.3 Definition of Relabelling Job

A relabelling termination job is defined by which inter-region pointers constitute work for that job. For example, the train algorithm defines all pointers into a region to constitute work for that region; termination of that region's job is therefore isolation of the region. Hughes Algorithm [52] defines work to be every pointer from a younger region, i.e. regions in some direction according to an ordering on the regions; termination of the job implies that the region is unreachable from younger but not older regions and therefore the region is isolated if it is the oldest region.

The choice of relabelling job definition will define what isolation may be inferred from the termination of any particular group of jobs. It also defines whether or not the algorithm is subject to the unwanted or victimised relative problems. The unwanted relative problem occurs when an object is relabelled into a candidate region of a relabel-target collector and there exist pointers to it from other regions; the symmetric case is the victimised relative problem where an object containing intra-region pointers is relabelled out of a candidate region of a relabel-source collector. If the system is subject to either of these problems, safety requires synchronisation between the relabelling process and relabelling-termination DTDA.

### 3.5.4 Choice of Relabelling Termination DTDA

Having defined each relabelling process, it is necessary to determine when each has terminated so that inferences about the isolation of graph regions may be drawn. Any correct asynchronous DTDA should be a valid choice in terms of correctness, but the choice has performance implications.

Systems using a wave-like DTDA may incur synchronisation overhead for a greater number of sites than a Doomsday DTDA. A wave-like DTDA appears to require synchronisation against an entire region (all sites in the region cease relabelling while the DTDA executes), whereas a Doomsday DTDA requires synchronisation only between the region's home site and the site performing an unwanted/victimised relative relabelling. However, a Doomsday DTDA requires synchronisation for every relabelling event that would imply unsafe task creation according to the definition of the unwanted- and victimised-relative problems, whereas the wave-like DTDA synchronises only when the train is

closing and therefore possibly isolated. Doomsday DTDAs therefore seem to incur more frequent synchronisation overheads over a smaller number of sites while wave-like DTDAs incur synchronisation over a greater number of sites but less frequently.

### 3.5.5 Choice of Remset DTDA

Reachability of a single object is equivalent to the DTD problem, so in theory, any DTDA may be used to solve that problem. However, the diffusing nature of how pointers to a particular object are copied throughout the object graph make Doomsday-style (diffusing) DTDAs a more sensible approach. Conversely, a wave-like DTDA may be used but there is a requirement to determine the scope of the wave and permitting the wave to visit every site within the system would be an unscalable approach.

### 3.5.6 Unwanted and Victimised Relatives

Where synchronisation between relabelling and the relabelling-termination DTDA is required due to the unwanted or victimised relative problems, the type of synchronisation will depend on the DTDA that was chosen. As stated above, wave-like DTDAs require broader synchronisation than Doomsday DTDAs and therefore may offer reduced concurrency.

In the case of a wave-like DTDA, Lowry [64] defines a train-closure protocol that may be used in conjunction with wave-like DTDAs.

In the case of a Doomsday-protocol DTDA, Norcross [82] defines a protocol wherein the site wishing to relabel an object sends a **request for witness** to the home site of the region in question; see Section 3.4.1.1.

### 3.5.7 Relabelling Liveness

Ensuring labelling liveness in the face of mutator activity means that at least some inter-region pointers must continue to constitute relabelling work even after they are erased; discrete-time sampling of the remembered sets of a region is not sufficient to ensure progress since it discards all previous history. The selection of work that is retained and how work is represented will have an effect on the conservatism of the collector and therefore its garbage loading and throughput.

The extreme approach is snapshot at the beginning which preserves all work after pointers are erased: any object that is reachable from outside a region will be relabelled out of that region. A minimal approach would retain sufficient work to ensure that every component reachable from outside the region is migrated

out of that region; the determination of the minimal subset may be difficult. One published approach [87] is to begin retaining pointers only when isolation detection has failed yet no work is observable, though the implementation described there had negative effects such as constraining the partition selection policy.

The designer of an algorithm must also prove that the implemented preservation of work is sufficient to provide completeness, i.e. a component spread across two regions must eventually collapse into a single region.

### 3.5.8 Relabel-Target Completeness

Completeness of the relabel-target requires not only liveness of relabelling but requires that every region that does not contain the root is eventually a candidate and its relabelling job is permitted to run to termination. This requirement is one reason that two-region collectors operate in phases with synchronisation, it also has fairness implications for multiple-region collectors. For example, the requirement that jobs be permitted to terminate means that jobs (and regions) must not be created so rapidly that older jobs never perform any work.

### 3.5.9 Relabel-Source Completeness

Completeness within relabel-source collectors has been proven in this thesis only for the case where there is a total ordering on regions and cyclic relabelling is not permitted; a more relaxed definition of regions may be possible but this is not proven here and would imply a different set of completeness requirements.

The strong suspicion guarantee places some fairness requirements on the suspicion algorithm; it is possible to achieve completeness without strong suspicion, particularly if assumptions about region growth can be made and this is explored in Chapter 6.

The growth guarantee is required for completeness with the ordered-regions assumption; with different assumptions, this growth may not be required — see Zigman's [102] model where a complete collector is used as the upper layer instead of the DTD of Surf.

### 3.5.10 Support Algorithms & Additional Policies

Though this model describes fundamental collectors, it is likely that the algorithm so specified will be used within a compound collector. For example, implementations of the train algorithm typically contain trains at the upper level and some form of copying partition collector at a lower level; both Lowry [64]

and Norcross [82] describe this situation. Therefore the algorithm designer must ensure that the implementation of the lower level does not violate the requirements imposed by the upper level of collection. The use of a partition collector and binding relabelling to operation of that collector has the effect of batching the relabelling that is performed and the designer should analyse the effects that this batching has on progress within the top-level collector; such an analysis is presented in Chapter 5.

Algorithms are required in an implementation to perform support tasks such as the creation and destruction of regions, synchronisation where defined by the above requirements, etc. Not every site need participate in every region, therefore protocols permitting sites to join and leave regions are required; these are not considered here but the designer must ensure their correctness and that there exist no race conditions between those protocols and the components defined by the model. The Surf model does not in general define how these protocols operate, it merely defines them to exist and provides some constraints on how they must operate. These protocols are examined below; each derives from an assumption of the system model or a safety or completeness requirement stated by Surf.

Leaving the internal operation of these support algorithms undefined is one means by which Surf achieves broad coverage over a wide range of garbage collection algorithms. Clearly it would be possible to define solutions to all of these requirements here, but that would make the Surf model specific to those solutions. Instead, the approach of defining the existence of and requirements on these algorithms permits the Surf model to be applied to many more collectors yet still provides specific information to implementors in terms of what requirements their implementations must fulfil in order to be correct.

#### 3.5.10.1 System Support

The system model implies reliable (lossless, duplication-free) message passing, non-crashing sites, etc. Mechanisms must be in place to fulfil these assumptions.

#### 3.5.10.2 Free Space Management

The ability to create new objects and to return their space to the free region when they are detected as dead implies the existence of mechanisms for managing free space. Surf places no constraints on this mechanism. In other words, Surf defines a GC and a GC may not exist without an object store to operate over.

#### 3.5.10.3 Region Membership Protocol

Regions are a distributed construct and assuming that they are not global for scalability reasons, there must be some means to define membership of a region. Specifically, there must be means for a site to join a region when it is to relabel an object into that region and to leave a region when it contains no more objects in that region. The joining and leaving of regions must not interfere with the relabelling termination detection DTDA, otherwise an unsafe collector may be produced. It is permissible that relabelling be paused or deferred on a site while the join protocol requests membership for that site.

#### 3.5.10.4 Region Creation Policy

The existence of regions in conjunction with regions being destroyed implies that there must be mechanism to occasionally create new regions so as to satisfy the completeness requirements.

In the case of a relabel-target collector, new regions must be created into which live objects are relabelled. If the collector has only two regions, the policy is simple: at the end of a phase, one region is discarded, an empty region is created and the root is moved into that region. For a many-region collector, regions must be periodically created to ensure that there are sufficient regions to satisfy the completeness requirements; the exact creation rate for optimum performance is dependent on mutator behaviour and not defined by Surf.

In the case of a relabel-source collector, new regions are created at suspect objects. Therefore, an implementation must contain a suspicion algorithm that provides the suspicion guarantee defined in the proof of completeness for relabel-source collectors. A simple implementation might be to fairly suspect every object, thereby providing the strong suspicion guarantee but more efficient approaches are possible. Because garbage will be reclaimed only when the suspect is dead, progress in a relabel-source collector is critically dependent on the design of the suspicion algorithm.

#### 3.5.10.5 Unbounded Region Growth

Another requirement for completeness in relabel-source collection is the guarantee that a dead region be permitted to grow to be usefully dead. Creation of new regions via suspicion is therefore permitted only when there exists no relabelling work for any existing regions and an implementation must provide a protocol to determine when this occurs. Because region growth is a diffusing process that is congruent with relabelling progress, the termination of region

growth may be detected by running a DTDA job for each region wherein intra-region pointers to the region in question are tasks of that job. Termination of that DTDA therefore implies that no further relabelling is available and that a new region may be created.

It is not necessary to make the region-growth DTDA correct with respect to concurrent mutator activity because completeness requires unbounded growth only for dead regions. Should mutator activity falsely cause an early declaration of termination, the region so affected was live and its growth is unnecessary for completeness.

#### 3.5.10.6 Work Ordering

In a typical implementation, work is performed serially: since there may be thousands or millions of objects available for relabelling at any one moment, it is not feasible to relabel them all concurrently. A policy is therefore required to decide the order in which work is processed. At a minimum, it must be starvation-free, i.e. every unit of available work must eventually be processed.

## 3.6 Limitations of the Model

An important restriction of this model is that objects may exist only on a single site: they may not migrate or be replicated, which would introduce questions of consistency. Garbage collectors supporting object replication are not described directly by this model.

The model describes a methodology for constructing garbage collectors from a set of components that are themselves not garbage collectors, i.e. **fundamental collectors**. It does not consider compositional techniques for combining multiple collectors into a larger, more complex and perhaps higher-performance whole, e.g. Hierarchical Garbage Collectors [102] and more specifically, generational collection [61, 97], the use of car-collectors in the Train Algorithm (see Section 5.2), the use of partition collectors and site-marking in Thor [69], the Starting With Termination [20] approach, etc. All of these compositional techniques require one or more existing garbage collectors, they do not describe the construction of a collector where none previously existed whereas the Surf model constructs garbage collectors from first principles.

The model as described and proven here is a **single-stage** approach to garbage collection: the collectors described by this model attempt only to generate usefully dead regions. It is possible to extend the model so that regions are generated for which other predicates are true and then compose multiple stages of algorithm to find intersections, differences, etc of regions, finally resulting in a

dead region, which is the approach taken by Trial Deletion [12] and described in Chapter 4.

The Surf model applies to closed-world collectors, i.e. those which have full control of their heap and do not need to interact with other heaps or collectors. Orthogonally persistent systems [8, 10] and federated stores [79] require cooperation between multiple heaps in the system and therefore the ability to track pointers in other heaps. This problem may be solved in a number of ways that amount to implementation techniques that provide multiple root objects; the maintenance of such roots sets is entirely outside the scope of the model but addressed in Section 5.2.4.

The completeness proof presented above prohibits infinite cyclic relabelling of objects to ensure that every cycle of garbage is eventually entirely contained within a single usefully dead region; the typical means of ensuring this is to place an ordering on regions. Zigman states that where no such ordering exists, completeness may still be achieved if a more complex isolation detection algorithm is used, i.e. one that may detect isolation of a group of regions. That approach requires an additional complete collector, i.e. one requires an existing collector to build a new collector; the Surf model in contrast can build a collector from components that are themselves not collectors. Zigman's hierarchical model relies on the existence of a Surf-derived collector at the uppermost layer.

The Surf model leaves several important protocols and support mechanisms within a collector undefined. This is necessary to achieve coverage of a broad range of approaches to collection but it means that an implementor must prove the correctness of these support protocols. Though the exact operation of these protocols is undefined by Surf, the model does provide constraints on most of these protocols that can aid in their design and proof. Decisions made in the design of these support mechanisms will have an impact on the performance of a collector.

## 3.7 Conclusion

Garbage collectors have previously been described without any formal or universal framework, thereby making it difficult to compare different algorithms published at different times or to prove the correctness of any given algorithm.

This chapter presents a novel model of garbage collection that permits the description and construction of garbage collectors from Distributed Termination Detection Algorithms and some other components and provides a framework for proving their correctness. The model:

- is a constructive approach to building garbage collectors,

- describes a wide variety of existing **fundamental** (not composed) garbage collectors,
- may be used to analyse the safety and completeness of existing collectors,
- may be used to construct entirely new garbage collectors, and
- may be applied to analyse the behaviour of a collector and provide insight into its operation.

Safety and completeness are defined formally in terms of object reachability and the determination thereof in finite time. A formal proof of the model's correctness is presented using a system model founded on atomic state transitions and a list of permissible events describing mutator activity. An analysis of the interaction of mutator events and events internal to the garbage collector produces a set of constraints that, if fulfilled, will result in the safety and completeness properties holding; fulfilment of these constraints provides a simple process whereby the correctness of a collector may be (dis)proved.

Inspection of existing collectors in terms of these constraints sheds light on the correctness (or otherwise) of said collectors and some aspects of their operation. The application of the constraints to an arbitrary selection of DTDA and useful relabelling process should result in the creation of a correct garbage collector, though the performance of the resulting garbage collector will depend heavily on choices made in its construction that are not addressed by this model.

Chapter 4 illustrates the mapping of existing fundamental collectors into the Surf model and thereby demonstrates the descriptive and analytic power of the model; Chapter 5 describes the extension of the model for use in a partitioned distributed persistent heap, analyses progress using the concept of work defined in the model and verifies this analysis through experimentation; Chapter 6 explores a new area of the design space defined by the model to instantiate a new garbage collector that is predicted to have unique and interesting properties.

# Chapter 4

# Applying the Surf Model

This chapter uses the Surf model of garbage collection to describe a number of existing collectors by mapping each into the model and describing the collector as an instantiation of the model according to the process presented in the previous chapter (Section 3.5). Describing a number of existing collectors and using the model to arrive at insights into those collectors is an illustration of the descriptive power of the model; it is a demonstration that the model has the power to describe a wide range of collectors and that describing them reveals important information about the correctness, operation and performance characteristics of those collectors.

Because Surf is an abstract description of how garbage is detected by the application of labels to objects, implementation techniques and details such as addressing and allocation are irrelevant to the mappings. However, an analysis of each instantiation in terms of the type of distributed termination detection algorithm (DTDA) chosen for isolation detection and the manner in which progress is made can provide insight into policy decisions that may be required, such as partition selection which is explored in Chapter 5.

It is also possible to map some incomplete collectors into the model; degeneracy in the mapping shows why those collectors are incomplete and is an indication, though not proof that every component of the model is necessary to implement a complete distributed garbage collector, i.e. that the model is minimal in the list of components that it requires a collector to possess. In other words, explicitly attempting to capture the essence of how a garbage collector makes progress in applying labels to objects and detects termination of that progress can provide insight into how a collector operates and where a collector is incomplete, the model shows that the collector is missing some component that the model deems critical to complete garbage collection.

The complete collectors described in this chapter are: Distributed Marking [57, 69], the Train Algorithm [49, 64], Hughes Algorithm [52], Back Tracing [68] and

Trial Deletion [12]. For the purposes of illustrating a counter-example, Reference Counting [30] is mapped into the model and its incompleteness explained in terms of the model. Trial Deletion is not a simple fit for the model but the critical part of its operation is found to match the model and it also turns out that certain necessary preconditions are fulfilled by a second instantiation of the model; Trial Deletion is therefore an amalgam of two cooperating instantiations of the model that together form a safe and complete collector.

In mapping each collector into the model, the following elements are considered:

- the classic definition of the collection algorithm,
- the definition of regions and their meaning,
- the direction of relabelling,
- the definition of the relabelling jobs,
- the approach to distributed termination detection for relabelling and trivial reachability,
- whether the algorithm is subject to the unwanted or victimised relative problem and if so, how it solves it,
- guarantees of liveness in the face of concurrency, where necessary, and
- any further inferences that the model allows to be drawn from how mapping.

In each case, it is necessary to show that the combination of relabelling process and region definitions will result in usefully dead regions, that the safety requirements of DTD are met and that safety and liveness of relabelling are preserved. Where all of those requirements have been met, the proof of Section 3.4 is applicable to the mapping and shows the correctness of the collector.

## 4.1 Distributed Marking

Distributed tri-colour marking, similar to that seen in Emerald [57] and first described for uniprocessor systems by Baker [14], is a concurrent collector implementable in uniprocessor or distributed systems.

### 4.1.1 Classic Definition

The classic definition of the algorithm is that each object may be marked as white, grey or black. All objects begin a phase coloured white except for the root, which is grey. The collector proceeds by scanning the pointers in a grey object, marking any white targets as grey and then changing the source to black. The invariant so maintained is that white objects may be referred to only by white or grey objects; once there remain no grey objects then all white objects are considered dead because the root has become black. For safety in the face of concurrency, snapshot-at-the-beginning (SATB) is implemented by makeing grey any white object that is the target of a pointer erased by the mutator.

### 4.1.2 Mapping

The table below shows the correspondence between each component of the model and the respective component in Distributed Marking. The Safety and Completeness section at the bottom of the table indicates how the collector fulfils the requirements listed as the second part of the mapping process.

| **Model Concept** | **Mapping to Distributed Marking** |
|---|---|
| Direction | Progress is by relabel-target because the marking wave travels in the same direction as the pointers. |
| Regions | There are two regions, Black and White.<br>Grey objects are members of the white region. |
| Relabelling job | Pointers from black to white objects constitute relabelling work.<br>Black may not become White hence regions are ordered, thereby avoiding infinite relabelling. |
| Relabelling DTDA | Relabelling-termination DTDA is Dijkstra-Scholten [93]: tasks diffuse in a tree fashion, following the mark bits, termination occurs when no Grey objects remain.<br>There is one job, termination of which models termination of the marking wave.<br>Each grey object is a task.<br>Each task spawns further tasks as a grey object's pointers are scanned (cover rule satisfied).<br>The transition to black marks the death of a task. |
| Remset DTDA | Single bit (grey bit), conservative with SATB; see Analysis below. |

| Model Concept | Mapping to Distributed Marking |
|---|---|
| Safety & Completeness Requirements | There are always two regions, which implies operation in distinct phases with global synchronisation between phases.<br>The unwanted relative problem is avoided since objects are always relabelled into the non-candidate (Black) region; synchronisation is required only when relabelling into a candidate region.<br>All live objects become white at phase-change, therefore all garbage objects will become members of a candidate region (initial conditions satisfied).<br>SATB provides correctness in the face of concurrent mutation. |

### 4.1.3 Analysis

Distributed marking illustrates a corner case of the model: there are exactly two regions and remembered sets (implemented with grey bits) are a very conservative approximation.

Because there are only two regions, synchronisation is required to move all objects from the black (non-candidate) region into the white (candidate) region when the old white region has been reclaimed; the reason being that a non-candidate region is becoming a candidate and this requires interaction with the DTD as described in Section 3.4. As a matter of implementation, white objects are reclaimed during this synchronisation period, logically if not physically. Because there are only two regions, no garbage may be reclaimed until a relabelling job completes, which requires traversal of the entire live graph, in turn implying that the collector exhibits poor scalability.

The "remembered set" functionality in this collector is the grey bit on each object; the approach is conservative because it indicates that an object is reachable from outside the region if it was ever so reachable within the current phase. If one starts with an accurate remembered set for trivial reachability (as defined in Section 3.3.4) and:

- discards all remembered set information for objects in the non-candidate (black) region,
- discards all remembered set information describing intra-region pointers,
- applies SATB to remove the possibility of remset counters decreasing, i.e. ignore all pointer erasures by the mutator, and

- reduces the precision of the remset to one bit (it cannot be decremented, so this remains safe),

then all that remains is a single bit of state which is set when the trace reaches an object and can never be cleared, though it is later discarded. Grey bits may therefore be described in terms of the DTD abstraction as follows:

- all white objects (the candidate region) have a remset job; a task is created when that object:
  - becomes directly reachable from the black region by relabelling another object, or
  - is discovered as reachable at phase-start due to mutator activity erasing a pointer to it (SATB);
- the Job has one task, representing the first discovered pointer-from-black,
- the task cannot be destroyed because of SATB, so
- no further tasks need be created, and
- the Job is discarded when the object is scanned and therefore enters the non-candidate region, i.e. is blackened.

The above shows that although the classic description of the algorithm does not include remsets, it includes similar yet degenerate and conservative functionality in the form of the grey bit.

Since all White objects reachable from Black objects are relabelling tasks, at termination of relabelling the White region will be unreachable from the Black region. The collector therefore forms usefully dead regions (white) because the root is in the black region. The DTD safety requirements (cover rule) are fulfilled. Since all objects except the root are made White at the beginning of a phase, all garbage objects will eventually enter a White region and have the opportunity to be reclaimed, so the initial conditions requirement is fulfilled. This mapping therefore demonstrates that the collector is safe and complete.

In summary, this algorithm can perform concurrent collection — concurrency between multiple instances of collection work and the mutator — but lacks scalability due to the phase synchronisation. It should be noted that this mapping describes a number of collectors often thought of as separate algorithms, namely semi-space copying [14] and mark-sweep; the differences between them lie only in how regions are represented and the order in which work is processed. For example, marking collectors have an explicit tag bit to represent the region and

relabelling involves setting the mark bit, whereas copying collectors represent regions in the addresses of objects and relabelling involves physically moving (copying) the object between regions.

The model also does not define the order in which work is processed, only that progress is made as a wave across the object graph. The shape of this wave will depend on the order in which work is processed, e.g. depth-first, breadth-first or some other ordering. Examples of published collectors conforming to this mapping are:

- McCarthy's mark-sweep collector [72], which operates in depth-first order,
- Cheney's collector [27], which operates in breath-first order and
- Ben-Ari's collector [16], which operates in lexical order.

## 4.2 Train Algorithm

The Train Algorithm has been instantiated in a number of contexts: uniprocessor mature-space [51], persistent storage: PMOS [78, 81], distributed heaps [49, 65, 64, 82] and combinations thereof: DPMOS [24]. The classic description of the algorithm is a very close match with the abstract model presented in the previous chapter; in fact it served as a starting point for creation of the abstract model.

### 4.2.1 Classic Definition

The system contains a number of **cars** (partitions) and **trains** (regions), the latter composed of many of the former. Each car resides entirely within one physical site while trains span multiple sites and provide the mechanism to reclaim distributed cycles of garbage. Collection proceeds independently and concurrently inside the cars, reclaiming garbage that is locally isolated. When collecting a car (by copying objects reachable from the car's remembered set to other cars), the collector may choose to *reassociate* (relabel) an object from the train it is in to another train, according to a number of rules designed to ensure completeness.

Reassociation proceeds until reachable and unreachable objects are separated along train boundaries, at which point a train may be detected as isolated and discarded.

### 4.2.2 Mapping

The collection of garbage inside cars is a throughput optimisation as far as the abstract model is concerned and may be discarded from the model: the

train algorithm is perfectly functional without it, though it will take longer to detect acyclic garbage structures. Performing additional GC within cars is a compositional technique therefore outside the scope of the model and not considered further here; cars are not necessary for the mapping.

The mapping is summarised as follows:

| **Model Concept** | **Mapping to Trains** |
|---|---|
| Direction | Progress is by relabel-target. |
| Regions | Regions are trains and are identified by integer names.<br>Regions are ordered, preventing cyclic relabelling.<br>Each object has a train number attached to it, perhaps indirectly; e.g. each partition may have a train number, if the store is partitioned. |
| Relabelling job | Relabelling is referred to as **reassociation** in the previous literature.<br>An object is relabelled by changing its train number, e.g. by moving it to a different partition if partitions there be.<br>Pointers from the root constitute relabelling work into the youngest candidate region.<br>Pointers from a younger region to an older region constitute relabelling work into the younger. |
| Isolation DTDA | Isolation may be detected by any asynchronous DTDA.<br>The isolation of every candidate region is represented by a job; each inter-region remset entry constitutes a task of the region isolation job. |
| Remset DTDA | Any valid asynchronous DTDA may be used to implement remembered sets. |

| Model Concept | Mapping to Trains |
|---|---|
| Safety & Completeness Requirements | There are always at least three regions: the mutator train plus two others; all non-mutator trains are candidates.<br>The unwanted relative problem exists because objects are relabelled into candidate regions; two synchronisation options have been described in the literature.<br>All non-root objects are always in candidate regions, so the initial-conditions requirement is fulfilled if regions are created to fulfil the three-region minimum.<br>Relabelling work is discovered by sampling remembered sets, therefore a subset of remset entries must be preserved to maintain liveness of relabelling in the face of mutator activity; the exact requirements for preserving relabelling work are derived in the Analysis below. |

### 4.2.3 Analysis

There is very close correspondence between the abstract model and traditional descriptions of the train algorithm; similar concepts are present in each. The original uniprocessor description of trains had no concept of distribution; DMOS [49] brought distribution to the system and this distribution causes two additional complications over the uniprocessor case:

- without synchronisation, live objects could be reclaimed due to the unwanted relative problem, and
- progress could fail due to mutator interaction.

The unwanted relative problem has been solved for the distributed train algorithm in DMOS and then more formally by both Lowry [65, 64] and Norcross [82]. Lowry's approach involves wave-based DTD algorithms and explicit synchronisation by preventing relabelling while the DTD is in operation. Norcross takes the Doomsday approach, sending a coverage request to the home site where necessary; a witness task is created at the home site and sent to the relabelling site to satisfy the Doomsday cover rule. The Doomsday approach appears to have minimal interruption: relabelling is halted at a single site when an unwanted-relative relabel occurs and only until the cover request is fulfilled (one message round-trip time); conversely, the wave-DTDA approach prevents

relabelling at all sites in a train while the DTDA is in operation, regardless of whether or not unwanted-relative operations are occurring.

Seligmann & Grarup [87] described the progress bug and propose a solution that is a restricted form of SATB. Their solution is tied to a naive partition-selection (for local collection) policy that requires that cars of a train be collected in order so as to reduce the amount of remset data; when collection has been performed on every car in a train and no progress made, it indicates that the progress-bug is occurring. At this time, all pointers into the train are recorded in a special set and forced to be relabelled in the next GC pass over the train.

Without the naive partition-selection policy and the ability to perform local collection in arbitrary order, some other mechanism is required to ensure that relabelling work is recognised without relying upon remset entries, as per Section 3.4. This may take the form of SATB, i.e. considering all pointers to represent relabelling even when they have been erased, or perhaps a more subtle algorithm: DMOS defines the concept of a **sticky bit** that preserves a remset entry as relabelling work after the erasure of a pointer. The minimum requirement for correctness is that at least one inter-train pointer to every component in a train that is reachable from younger trains is retained as relabelling work. Consider a component $x$ in train $\mathfrak{L}$ that is reachable from train $\mathfrak{M}$ due to pointers in objects in the set $rr$; if this set is non-empty (the component is reachable from outside the train) then at least one inter-train pointer to $x$ is retained as work:

$$(\mathbf{Reachable}(z, x) \wedge z \in \mathfrak{M} \wedge x \in \mathfrak{L}) \Rightarrow z \in rr$$
$$rr \neq \varnothing \Rightarrow \exists y.x \mid y \in rr \wedge retained\_as\_work(y.x)$$

Because the Train Algorithm may have an arbitrary number of trains (regions), it is possible to avoid the end-of-phase synchronisation problem present in Distributed Marking; the reason for this is that at no time does a non-candidate region become a candidate region and thereby violate DTD correctness constraints. When relabelling an object, it will typically be moved to the region containing the source of the pointer representing the work, the exception being when the pointer is from the root, which resides alone in an infinitely-young non-candidate train to which no other objects may be added. Objects that are directly referred to by the root are moved to some candidate younger **active train** which will not typically be the focus of collection effort but which does possess an operating isolation DTDA job.

Completeness requires that live objects be separated from garbage components that have coalesced into a single train, which in turn requires the regular creation of new trains that are made active; this is a corollary of the requirement that there be at least two or three regions in conjunction with the reclamation of older regions.

Trains may be global or (for asynchrony and robustness reasons) span some subset of the sites in the system; in either case the implementation must contain mechanism related to the safe creation, growth, shrinkage and destruction of trains. Train maintenance mechanism is outside the scope of the mapping and its correctness must be separately proven by an implementor. The reason for this is that Surf defines only the existence of regions and not the process of their management because there clearly exists a range of approaches to keeping track of which sites each region spans; specifying one approach would overconstrain the model to one specific approach. Section 3.5.10 lists a number of such mechanisms that are undefined by Surf and the reasoning for leaving them undefined: Surf attempts to capture only the essence of a garbage collector.

Since trains need not be global, it is possible to reclaim distributed components of garbage by involving only the sites containing the garbage. Since detection of train isolation is performed by DTD, no synchronisation with other trains or the mutator is necessary. The collector therefore offers scalability because global cooperation is not required to reclaim a component of garbage contained within some small subset of sites.

Therefore the Surf model shows that the train algorithm is safe, complete, asynchronous and scalable: it does not interfere with the mutator, it can reclaim garbage with the participation of a minimal number of sites and collection can proceed concurrently on a number of sites without any form of synchronisation other than between relabelling and region-isolation jobs as required by the unwanted relative problem. Reclamation of garbage does not require the collector to perform operations on the entire heap, e.g. all live objects. This mapping is an illustration of how the Surf model, via the analysis process of Section 3.5 can result in a designer drawing a range of conclusions about the performance of a particular garbage collector before implementing it.

## 4.3 Hughes' Algorithm

Hughes' Algorithm [52] is very similar to both Distributed Marking and the train algorithm: it is a generalisation of distributed marking that uses multiple regions to increase concurrency, reduce synchronisation (no phase swaps) and improve timeliness of collection. The way that the isolation DTD is mapped in this collector avoids the unwanted relative problem at the cost of preventing reclamation of an epoch until all older epochs are isolated, thereby trading off scalability and timeliness in return for greater throughput via less synchronisation overhead during operation compared to the train algorithm.

### 4.3.1 Classic Definition

Instead of a single black/white mark bit on each object as per simple marking algorithms, each object is marked with an **epoch** number. Where a simple marking algorithm has a single wave progressing across the live portion of the object graph, blackening objects, Hughes has multiple concurrent wavefronts, each representing an epoch. When a wave reaches an object, the object's epoch is set to the wave's epoch if the former was lower; each object therefore knows the latest wave to have visited it.

When a wave completes, all objects marked with a lower epoch number than that which completed are considered garbage and reclaimed. New epoch waves are started regularly so that a number of them are operating concurrently. Because there is a global ordering on regions (epochs) that may be evaluated locally, Hughes uses global synchronisation in the starting of new epochs to ensure that the beginning of an epoch is unique.

### 4.3.2 Mapping

The mapping of Hughes' Algorithm to the abstract model of GC is summarised as follows:

| **Model Concept** | **Mapping to Hughes' Algorithm** |
|---|---|
| Direction | Progress is by relabel-target. |
| Regions | Regions are identified by increasing integer epoch numbers<br>Regions are ordered, preventing cyclic relabelling. |
| Relabelling job | Pointers from younger regions to older regions constitute relabelling work into the younger. |
| Relabelling DTDA | Each job represents the progress of an epoch trace.<br>Tasks in a job represent objects in older epochs that are reachable from that epoch.<br>Termination of a job implies that all older epochs are unreachable from that epoch and are therefore garbage. |
| Remset DTDA | Remsets consist of a single bit per epoch, as per Distributed Marking. |

| **Model Concept** | **Mapping to Hughes' Algorithm** |
|---|---|
| Safety & Completeness Requirements | Choice of DTD mapping avoids the unwanted relative problem (see analysis below).<br>There are always more than two regions (epochs) because regions are regularly created.<br>The regular creation of regions ensures that the initial-conditions requirement is fulfilled.<br>SATB provides liveness of relabelling. |

### 4.3.3 Analysis

On the surface, this mapping is similar to that of the train algorithm: an ordered sequence of regions and objects relabelled from older regions to younger regions according to connectivity, eventually leaving garbage behind to be reclaimed in the oldest regions. There is, however, an important difference in how the DTD algorithm is mapped into the system which has implications for the synchronisation required and the scalability of the resultant garbage collector.

In the train algorithm, each train (region) has a job representing the reachability of that region. Any pointer into that region constitutes a task of the job, so a region will not be reclaimed until it is unreachable from all other regions.

In contrast, each DTD job in Hughes' Algorithm represents the reachability of an epoch and all older epochs only from younger epochs. Safety therefore requires that region $n$ must not be detected as isolated (though it may actually be isolated) until all regions $n - k \mid k \in \mathbf{Z}^+$ — older epochs — are also isolated. This restriction on the order in which regions may be reclaimed impacts the timeliness of collection in comparison to the Train Algorithm but it requires less synchronisation during operation because it avoids the unwanted relative problem. Reclamation of epochs out of order is unsafe since there may exist pointers from older epochs to younger epochs that are not tracked as work by any relabelling job.

This mapping demonstrates the case where termination of a relabelling job does not directly infer region isolation; relabelling job termination must be combined with an additional condition — all older epochs/regions reclaimed — to infer isolation of a region.

The Train Algorithm suffers the unwanted relative problem because any object reachable from another region constitutes a task of the region isolation job. In contrast, such objects are tasks only in Hughes' Algorithm when they

are reachable from a younger epoch. Since all relabelling occurs in the direction of older to younger, an imported object may have no unwanted relatives, i.e. it will never require the creation of a task of the isolation job for the region into which it is imported.

Since pointers from older epochs are ignored, only pointers to the newly imported object from younger epochs are of import; any object that is the target of pointers from younger epochs already constitutes a task. For this reason, no task creation and therefore no synchronisation is required during relabelling. The DTD safety requirements are fulfilled and the collector is safe.

Usefully dead regions are formed since the relabelling process will form region boundaries along boundaries of reachability: if some garbage objects are present in any non-youngest region, they will eventually be the only objects in that region and it will therefore be usefully dead. Any object that becomes garbage before the creation of a new epoch will be reclaimed; therefore with the continual creation of epochs, it can be guaranteed that all garbage objects will eventually be in a non-youngest region and therefore reclaimable. The collector is therefore complete.

The use of SATB in conjunction with a simple forward trace means that remsets degenerate to a single bit per epoch, using the same derivation provided for Distributed Marking.

In summary, Hughes' Algorithm represents an interesting point in the Surf design space between distributed marking and the Train Algorithm. Its choice of DTD mapping is similar to marking so avoids the unwanted relative problem and the synchronisation that that entails; the tradeoff for this reduction in synchronisation is poorer timeliness since it requires that live objects are migrated out of older regions before newer regions may be detected as isolated. Its use of more than two regions avoids the phase-change synchronisation present in tri-colour marking.

## 4.4 Back Tracing

Back Tracing differs from the other collectors presented here in that it takes the relabel-source approach instead of relabel-target. It provides complete garbage collection with similar robustness properties to the Train Algorithm, i.e. the only sites that need participate in the reclamation of garbage are those that the garbage component spans. It is presented here to show that the new model is capable of describing a wide variety of collectors that were previously thought to be disparate.

Back Tracing is presented in Thor [68] as a completeness mechanism that detects distributed cycles of garbage after other garbage collection algorithms have reclaimed all acyclic garbage and non-distributed cyclic garbage. The Thor system is a composition of three separate collectors: the partition collector, site-marking collector and completeness algorithm. Other completeness algorithms [67, 69] have been investigated in Thor but this section describes only Back Tracing.

This section presents an instantiation of the model that represents back tracing but not in exactly the same way implemented by Thor; the differences lie in the way that Thor handles pointers held by the mutator (this incurs some synchronisation cost), the batching of tracing into partition-sized chunks and the use of a suspicion algorithm to bound the graph region that the back tracing algorithm may operate over. These differences are explored later.

### 4.4.1 Classic Definition

Back Tracing begins with suspicion of an object: the suspicion algorithm hypothesises that a particular object may be garbage and tracing begins at that point. Pointers to the suspected object are discovered and followed back to their source; this process proceeds recursively until no further incoming pointers remain or the root is reached. If the root is reached, all objects so traced are considered live and the operation abandoned; otherwise they are garbage and are reclaimed.

Thor is a partitioned object store with per-partition remembered sets and a forward tracing partition collector (PGC) that considers remembered set entries to be roots. The PGC in conjunction with remembered set update will reclaim all acyclic garbage.

The discovery of pointers to an object is heavily dependent on the implementation of a heap and a number of techniques are available to fulfil this requirement; such mechanisms may be very expensive. Thor uses remembered sets and a pointer tracking protocol to keep track of inter-partition pointers, including remote pointers. Back tracing uses this information as well as connectivity within a partition to determine how to trace backwards over the object graph; this connectivity information is computed by a variation on Tarjan's Algorithm [92] embedded in the PGC.

### 4.4.2 Mapping

The mapping of Back Tracing into the new model is summarised as follows:

| Model Concept | Mapping to Back Tracing |
|---|---|
| Direction | Progress is by relabel-source because the progress wave travels in the opposite direction to pointers in the object graph. |
| Regions | There are two regions: those marked by the trace (candidate region) and those not yet marked (non-candidate). |
| Relabelling job | Pointers into the candidate region constitute work of the single relabelling process. |
| Relabelling DTDA | The DTDA in use is Dijkstra-Scholten: tasks diffuse in a tree-like fashion across the object graph, termination occurs when no further diffusion progress is available.<br>There is one DTD job: the reachability of the candidate region.<br>Each object reachable from the non-candidate region is a task.<br>Termination of the job implies that the marked region is isolated; if it did not absorb the root then it is a candidate and therefore usefully dead; otherwise the candidate region is abandoned. |
| Remset DTDA | Remsets are implemented using a DTDA that is similar to Lermen and Maurer's [60] pointer tracking in its use of FIFO channels and a carefully chosen message path to ensure safety. |
| Safety & Completeness Requirements | Choice of the DTD mapping avoids the victimised relative problem.<br>Synchronisation is required where the candidate region absorbs the root.<br>SATB or similar is required to ensure progress.<br>The strong suspicion guarantee is satisfied by the suspicion algorithm.<br>Candidate regions are permitted to grow without bound. |

### 4.4.3 Analysis

The mapping as presented is relatively straightforward; the only complication is ensuring that the suspicion algorithm is fair, i.e. that all garbage objects are suspected within finite time. Classic descriptions of the algorithm are expressed

as a depth-first-search, therefore the termination detection (without concurrent mutator activity) is identical to the approach taken by Dijkstra-Scholten.

As described in Section 3.4.2.1, the unwanted relative problem as described in DMOS [49] can be generalised to a requirement for synchronisation where relabelling implies the unsafe creation of region isolation detection tasks. In the case of a relabel-source collector, this can manifest as the **victimised relative problem** where objects containing intra-region pointers are relabelled out of a candidate region. The victimised relative problem is avoided in Back Tracing because objects are always relabelled from the non-candidate region into the candidate region. Because no reachability DTDA job operates for the non-candidate region, no unsafe task creation is required for objects leaving this region so no synchronisation is required, hence Back Tracing does not exhibit the victimised relative problem.

As ever, the collector must be aware of concurrent mutator activity: the creation of a pointer into the candidate region has consequences for the safety and progress of collection. If the DTDA is not made aware of the creation then safety is compromised; if the relabelling process is not made aware then progress will stall. Safety is simply provided by ensuring that pointers held by the mutator (i.e. in the root) are represented by tasks in the DTDA. Progress may be ensured by application of SATB or similar.

If the root should be relabelled into the candidate region, the region becomes a non-candidate and the DTDA is abandoned, which requires synchronisation. All sites must be notified of the region abandonment, all objects in the candidate region are relabelled into the non-candidate region and then a single new region is created at suspect object, chosen with the cooperation of all sites. This requirement for global synchronisation is predicted by the model because the instantiation presented here has only two regions and more than two regions are required if such synchronisation is to be avoided.

It should be observed that this algorithm is very expensive if live objects are suspected since it may repeatedly traverse significant portions of the live object graph and discover no garbage in each phase. Efficient operation is therefore critically dependent on the suspicion algorithm employed to select the object at which the trace begins. It would also be valuable to bound the graph region that the back tracing algorithm may operate over and prevent it from repeatedly traversing live portions of the graph.

Concurrent operation of multiple instances of this collector is permitted so long as each instance is unable to observe the operation of other instances, i.e. each object is independently a member of the candidate or non-candidate region of each collector instance; candidacy in one instance is no constraint on candidacy

in other instances.

### 4.4.4 Back Tracing in Thor

This subsection explores the differences between the simple mapping described above and the approach taken in Thor. There are three primary differences: the suspicion algorithm is defined, objects are divided into partitions and there is a lack of mutator activity awareness. As mentioned earlier, Thor is actually a composite of three collectors where back-tracing is required only to collect distributed cycles of garbage; the structure of the other collectors and efficiency reasons drive these differences from the mapping.

#### 4.4.4.1 Suspicion

The Thor suspicion algorithm is based on distance from the root measured in inter-site pointer spans. When performing a partition collection, distances of pointers to the partition are propagated through the partition to pointers out of the partition; the distance of each output pointer is the minimum of all incoming pointer distances that can reach that outgoing pointer. When an outgoing pointer's distance changes, this change is transmitted to the target object and remembered until the target object undergoes its own partition collection. Since all live objects have finite distance from the root, their distance value remains finite; in contrast, objects in a distributed garbage cycle will have ever-increasing distances. Figure 8 illustrates the suspicion distances implemented in Thor, wherein intra-site pointers have zero contribution to the distance.

Two distance thresholds provide suspicion: the threshold of suspicion, $T_s$ and the threshold of activity, $T_a$. Back tracing begins at objects where $distance \geq T_a$ and halts at any object where $distance < T_s$. The distance-suspicion algorithm therefore defines a starting point for back tracing and a bound on the graph region that it may operate over. Where a back trace fails (discovers an unsuspected object), all objects in that failed region have their threshold of activity increased to prevent repeated futile scanning.

Since all local garbage is reclaimed by the other collectors in the system, the distance metric is permissible. Local cycles of garbage will not have increasing distance but they are reclaimed by the site-marking collector. Likewise, acyclic garbage will not have increasing distances but it will be reclaimed by the partition collector.

The suspicion algorithm employed by Thor is therefore sufficient in that it will eventually nominate every garbage object within finite time that is not reclaimed by another collector in the system. The bound on back tracing provided by

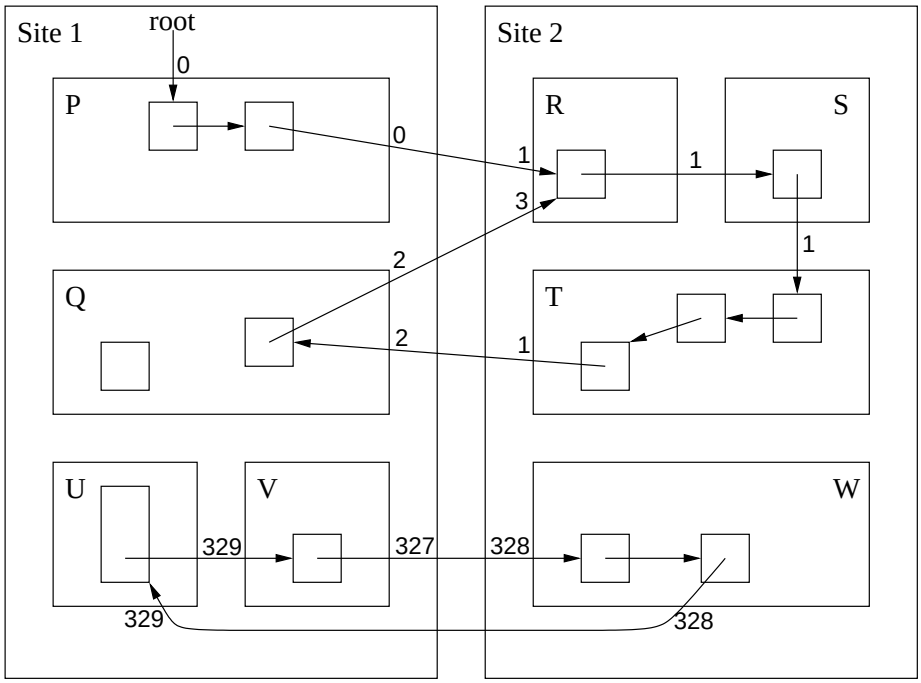


Figure 8: Distance Heuristic for Suspicion

$T_s$ is believed to improve efficiency by preventing repeated scanning of live objects but it may prove problematic where a back trace of a large circumference garbage cycle is abandoned repeatedly; this will occur where $circumference > T_a - T_s$. Circumference of a strongly connected component is defined here as the difference between the largest and smallest distances within the component; for a simple cycle is exactly equal to the circumference of the cycle in the distance metric, inter-site pointers.

#### 4.4.4.2 Partitions

The partition collector present in Thor is used to efficiently provide the connectivity information required for back tracing. Inter-partition connectivity is tracked using a variant of remembered sets (referred to as inrefs and outrefs); intra-partition connectivity is stored only for suspected ($distance \geq T_s$) objects and is computed during the forward trace of the partition collector. This removes the need to maintain connectivity information for the majority of live objects, thereby increasing efficiency somewhat. Tarjan's Algorithm [92] is used in a modified form to determine connectivity between the suspected ($distance > T_s$) incoming and outgoing pointers of each partition.

Back tracing therefore has two kinds of steps: local and remote. A local step uses the intra-partition connectivity information computed during the last partition collection and a remote step uses the remsets to determine the source of pointers.

This is not a change to the mapping *per se*, merely a batching of operations into partition granularity: the algorithm steps backwards a whole partition at a time rather than one object at a time.

#### 4.4.4.3 Mutator Ignorance

The most important difference between the mapping presented above and the implementation in Thor is that the latter is unaware of concurrent mutator activity. The reason is that it is designed for use in a transactional system and makes use of the client pointer tracking algorithm of Amsaleg *et al* [4] that permits it to be ignorant of many pointers held by the client, thereby reducing pointer tracking costs. Note that the Surf model is closed-world, i.e. it contains every pointer in the system, therefore it is incompatible with use in a persistent store such as Thor without extension of the model. Extension is required to make collection correct in the face of objects and pointers held by client caches and an example of such an extension is presented in Chapter 5; the extension requires that the tracking of pointers in client caches safely integrate with the Surf model.

The extension used in the Thor system does not provide sufficient information regarding mutator activity to retain safety of the Surf model, therefore Thor adds an additional safety-check synchronisation to ensure that unsafe mutations have not occurred.

Were back tracing to be implemented in conjunction with a different client pointer tracking algorithm, the safety assumptions of the model could be satisfied and the additional synchronisation step avoided.

## 4.5 Reference Counting

Reference Counting [30] is an incomplete garbage collection algorithm because it is incapable of reclaiming cycles of garbage. It is presented here to demonstrate that an algorithm that does not contain all of the components specified by the abstract model does not constitute a complete garbage collector and therefore that all such components are necessary.

### 4.5.1 Classic Definition

Each object has a reference count associated with it. When a pointer to the object is duplicated, the counter is incremented and when a pointer is erased, the counter is decremented. The counter therefore represents the total number of references to the object: when it reaches zero, the object is reclaimed. Reclamation

of an object requires that the pointers it contains be erased, possibly resulting in the cascading reclamation of a large number of objects.

Correctness in a distributed system requires the use of more than a simple counter due to the problems described in Section 2.3.1.

Note that this mapping does not apply to any particular published collector, it is merely the extension of uniprocessor reference counting to a distributed system.

### 4.5.2 Mapping

The degenerate mapping of reference counting into the new model is summarised as follows:

| **Model Concept** | **Mapping to Reference Counting** |
|---|---|
| Direction | There is no relabelling. |
| Regions | Each object resides in its own region. |
| Relabelling job | There is no relabelling job. |
| Isolation DTDA | There is one job representing isolation of each region (object). The object constitutes a task of its job if it is reachable. Termination Detection is a local decision since no region is distributed. |
| Remset DTDA | A DTDA similar to Task Balancing would be used to implement remembered sets, though any asynchronous DTDA is a valid choice. |
| Safety & Completeness Requirements | Without relabelling, there is no unwanted or victimised relative problem. Without relabelling, there is no liveness of relabelling, so the collector is not complete. |

### 4.5.3 Analysis

As shown in the description of Remembered Sets of Section 3.3.4, distributed reference counting is equivalent to a remset: the trivial reachability of each object is a job and the pointers that refer to the object are tasks of that job. Duplication of pointers is modelled by the creation of tasks and the erasure of a pointer is the

death of a task. Termination is reached when the object is trivially unreachable, i.e. there are no pointers to it.

Reference Counting collectors therefore have only the **lower level** of DTD, the remset. Without a relabelling process and accompanying DTDA to detect when relabelling has formed a usefully dead region, there can be no completeness. Considered alternately, complete collection requires that the scope of the region-isolation DTD (the one that detects usefully dead regions) to be at least as great as the largest possible strongly connected component of garbage; in the case of reference counting, the maximum DTD scope is a single object so any components larger than that will not be reclaimed.

## 4.6 Trial Deletion

Trial Deletion [12] is a technique for discovering cycles of garbage in conjunction with a reference counting collector that reclaims all acyclic garbage.

The core components of the model are present: regions, relabelling and (for the distributed case) the use of DTD to detect termination of the relabelling process. The difference is that two relabelling processes are used in sequence to find the difference of two sets, **reachable from suspect** and **reachable from live**. The difference so found comprises all of the cyclic garbage in the system.

Trial Deletion does not fit neatly into the model as stated in Chapter 3, rather it implies an extension of the model to multiple phases and analysis of external conditions to show correctness. Specifically, the collector operates in two phases, each of which may be described in terms of the Surf model but neither alone constitutes a correct garbage collector; additional logic not defined by the model is required to combine the inferences drawn at the conclusion of each phase, resulting in an indication of garbage. This extension to the model is examined further in Section 4.6.4.

### 4.6.1 Classic Definition

Briefly, Trial Deletion is composed of two tracing phases:

- a first trace from suspect objects to discover the maximum possible extent of all garbage cycles (Discovery), and
- a second trace to rescue objects that are reachable from outside the extent discovered in the first trace (Rescue).

The correctness of the collector is dependent on the assumption that there is no acyclic garbage in the system, which means that a forward trace from some point

in a cycle will reach every other object in the garbage component. Temporary decrementing of reference counts due to pointers discovered in such a trace will leave the reference counts at zero for every object not trivially reachable from outside the component — this is where the lack of acyclic garbage is assumed. Were acyclic garbage to be present and hold a reference to a garbage cycle, the cycle would appear to be reachable.

Objects begin a cycle-collection round as Black, unless they are suspect objects, in which case they are Purple and therefore members of a metadata set labelled Roots. There must be at least one Purple object in each cycle of garbage, a situation which is guaranteed by making objects Purple when they are the subject of a decrement resulting in a non-zero reference count.

The first trace begins at each object in Roots (i.e. all the Purple objects) and objects so discovered are marked Grey. Each time an object is marked as Grey, all objects it refers to have their pointer counts decremented and the trace proceeds to those child objects.

At the end of the first trace:

- all objects reachable from the suspects are marked as Grey and are potentially garbage,
- the Grey region contains all possible cyclic garbage since every cycle must have contained a Purple object,
- reference counts do not reflect the presence of any pointers within the Grey region,
- Grey objects with zero reference counts have no pointers to them from outside the Grey region, and
- Grey objects with non-zero reference counts are live due to the presence of a pointer from outside the Grey region, i.e. from a live object.

The purpose of the second trace is find the transitive closure of the live objects in the Grey region and mark them Black so that they are not reclaimed. All Grey objects not so marked as Black will become White and are detected as garbage. The algorithm therefore discovers two regions during its operation:

- Grey: reachable from a suspect and therefore containing all possible cyclic garbage, and
- Re-Blacked: Grey objects reachable from outside the Grey region and therefore live.

The difference between these two sets (the White region) is the entirety of cyclic garbage in the system.

### 4.6.2 Mapping

The original description of Trial Deletion was presented for a uniprocessor environment using scalar reference counts; it is extended here with the use of vector reference counts (remsets) to a distributed environment. Since a distributed reference-counting implementation is required as the basis of this algorithm, refer to the mapping presented in Section 4.5. Trial Deletion provides the mechanism (relabelling, region DTDA) that was degenerate in the reference counting mapping, finally resulting in a complete collector.

Since two traces are performed by Trial Deletion, two separate mappings to the new model are provided, one for each tracing phase. This description relies on an external suspicion algorithm to nominate all necessary Purple objects before a cycle-collecting phase begins.

The first phase, Discovery, is mapped as follows:

| **Model Concept** | **Mapping to Trial Deletion, Discovery Phase** |
|---|---|
| Direction | Progress is by relabel-target. |
| Regions | There are two regions: Grey (initialised from the suspected objects) and Black (all other objects). |
| Relabelling job | Every Black object reachable from a Grey object constitutes relabelling work.<br>Grey objects may not become black, preventing infinite relabelling. |
| Isolation DTDA | Termination detection is by Dijkstra-Scholten.<br>There is one job: reachability of the Black region from the Grey region.<br>Each Black object reachable from the Grey region is a task. |
| Remset DTDA | Remsets are single-bit as per the Distributed Marking mapping. |
| Safety & Completeness Requirements | This phase alone is neither safe nor complete; its purpose is to discover all objects reachable from some suspected garbage object, i.e. the total suspected region.<br>SATB is required for safety and liveness, implied by the use of single-bit remsets. |

The phase begins by initialising the DTD with a number of extant tasks, one per suspected (new Grey) object, and continues until no Black objects are

reachable from Grey objects. It alone is not a complete or safe collection algorithm since none of the remaining regions (Black or Grey) are usefully dead.

The second phase rescues Grey objects that are reachable from the Black region; thereby preventing live objects that are reachable from garbage from being erroneously reclaimed. The Rescue phase is mapped as follows:

| **Model Concept** | **Mapping to Trial Deletion, Rescue Phase** |
|---|---|
| Direction | Progress is by relabel-target. |
| Regions | There are two regions: Black (live; non-candidate) and Grey (candidate region). |
| Relabelling job | Pointers from Grey objects to Black objects constitute relabelling work.<br>Black objects may not become Grey, preventing infinite relabelling. |
| Isolation DTDA | Termination detection is by Dijkstra-Scholten.<br>There is one job: reachability of the Grey region from the Black region.<br>Each Grey object reachable from the Black region is a task.<br>Termination implies that the Grey region is usefully dead. |
| Remset DTDA | Remsets are single-bit as per the Distributed Marking mapping. |
| Safety & Completeness Requirements | Completeness is achieved only if the Grey region encompasses all garbage at the start of this phase (the initial conditions requirement).<br>Safety requires the use of SATB or similar to ensure that live objects are Black at the end of the phase.<br>The unwanted relative problem is avoided since objects are always entering a non-candidate region.<br>Synchronisation is required at the end of the phase because there are only two regions. |

At the end of this second phase, no Grey object is reachable from a Black object, therefore the Grey objects (now White under the classic definition of Trial Deletion) are garbage. As per the new model, the Grey region is usefully dead at the end of this phase. It should be noted that this mapping is substantially identical to that presented for Distributed Marking in Section 4.1 though the choice of colour in each region differs.

### 4.6.3 Analysis

It should be noted that the trial decrementing of reference counts in the Discovery phase and then subsequent re-incrementing in the Rescue phase has been elided from the mapping since this is merely a mechanism to detect the presence of pointers from Black objects to Grey objects.

Analysis of Trial Deletion initially appears much more complex than the other collectors because it is mapped into the model in two separate phases. Relating the second phase mapping (Rescue) to that of Distributed Marking is important since it shows that Trial Deletion is merely an optimised version of Distributed Marking that operates on a bounded region of the object graph instead of the entire graph. Distributed Marking assumes liveness initially only for the root; every other object is effectively suspect and must be rescued in the tracing operation whereas Trial Deletion suspects only a smaller region of the graph and then rescues live objects from that region.

Safety of Trial Deletion is provided by the fact that the second phase mapping, the one that actually discovers garbage, is identical to the Distributed Marking mapping.

Completeness of Trial Deletion requires additional reasoning; we must ensure that all garbage present in the system at the beginning of the Rescue phase is in the Grey region. Note that this requirement is practically identical to the requirement provided in the Proof (Section 3.4.1.3) that all garbage objects must eventually be in a candidate region. The need for this requirement is two-fold in this case: garbage cannot be reclaimed if it is not in a candidate region and also that garbage in the Black region may prevent the reclamation of garbage that is in the Grey region.

Trial Deletion ensures that this requirement is fulfilled in two ways: acyclic garbage is reclaimed by the reference counting mechanism and the Discovery phase will find all objects that are reachable from a garbage cycle. Therefore the only garbage in the system at the beginning of the Rescue phase is reachable from a garbage cycle and such objects will have been marked Grey in the Discovery phase.

### 4.6.4 Implications for Surf

The fact that Trial Deletion does not directly fit the Surf model as defined in Chapter 3 would seem to imply that the model does not have broad applicability, however, this section shows that Trial Deletion may be considered to fit within the model if it is extended to permit the use of multiple relabelling jobs to infer the unreachability of a region.

The Surf model in its simplest form **decides** the existence of garbage at the termination of a relabelling job; it uses the definition of that job to infer that there are no pointers to a region from some subset of other regions. If the definition of the relabelling job is such that its termination implies that there are no pointers to a region, the region is usefully dead when the job terminates. Such collectors are referred here to as **single stage**, i.e. they reclaim garbage through a single definition of a relabelling job. Though a single stage collector may have many candidate regions, only a single relabelling job is responsible for detecting when each region has become usefully dead and though Hughes' Algorithm (Section 4.3) applies external logic — an ordering on region reclamation — to infer that a region is usefully dead, it is still a single-stage collector.

Trial Deletion contains two stages that map to the Surf model, but neither stage alone is a correct garbage collector. By the definition of relabelling in the first stage, termination of that phase allows one to infer that no live objects are reachable from the suspected region. By the definition of relabelling in the second phase, termination of that phase allows one to infer that no suspect objects are reachable from the live region.

The second phase therefore appears to be a collector because the inference it draws is that a region is usefully dead. However, it is not a complete collector without the application of external logic (Section 4.6.3) to prove that the suspect region contains all dead objects.

Therefore Trial Deletion is a Surf-compatible collector because it may be described in terms of relabelling jobs, the detection of termination of these jobs and the application of logic (as Hughes' Algorithm does) to the definition of termination for each relabelling job to infer that a region is usefully dead. The shortcoming of the model with respect to Trial Deletion is that the instantiation process provided with the model (Section 3.5) will result only in the instantiation of single stage collectors and provides no guidance for the construction of multi-stage collectors similar to Trial Deletion.

Since it is possible that unions, intersections and other combinations of regions may be computed in multiple stages, Surf provides no formal framework for constructing or proving the correctness of multi-stage collectors. However, Surf does define that termination of a relabelling job implies the lack of a certain class of pointer, i.e. the lack of connectivity between some subset of regions. It is then up to the GC designer to combine multiple relabelling jobs so that the inferences drawn by each (lack of connectivity between certain regions, according to the definition of each relabelling job) may likewise be combined to show that a region is usefully dead.

## 4.7 Conclusion

In conclusion, this chapter shows how the Surf model of abstract garbage collection may be used to describe a number of existing collectors and thereby demonstrates the descriptive power of the model. A wide spectrum of collectors are described, spanning from the incomplete (Reference Counting) or unscalable (Distributed Marking) to more scalable, asynchronous and robust collectors (Train Algorithm).

Analysis of Trial Deletion in terms of the new model resulted in the description of a new garbage collector: Distributed Trial Deletion.

Analysis of Back Tracing in Thor reveals that safety is achieved in that system only by a final global synchronisation step when garbage is discovered; this synchronisation could in fact be avoided by treating mutator activity differently.

This chapter therefore shows that the Surf model has descriptive and analytical power: it can describe a wide range of collectors, provide insight into those collectors and lead to the development of new collectors. The continuation of this process is to experimentally confirm the predictions of the model (Chapter 5) then explore the design-space provided by the model to arrive at an entirely new garbage collector (Chapter 6).

# Chapter 5

# Experimenting With Trains

The purpose of this chapter is to demonstrate that the Surf model is an aid to the design and analysis of new garbage collectors and then to confirm the validity of that analysis through experimentation. Secondarily, the chapter also demonstrates that the formality of the Surf model may be extended for use in systems that do not exactly conform to the Surf system model.

The Surf abstract model of GC is applied to the design and analysis of a distributed instantiation of the Train Algorithm [49, 51, 65, 78, 81, 82] similar to that of Lowry [64] with the goal of providing insight into policy requirements. Specifically, the understanding of **progress** as defined by the abstract model is used to design a **partition selection policy** for the Train Algorithm. The empirical confirmation in this chapter of the predictions made by the model demonstrate that the model is capable of providing new insight into collectors in addition to the previously-known insights presented in Chapter 4.

To reduce remembered set costs, implementations of the Train Algorithm [24, 64, 82, 87] typically compose trains with a partition collector (PGC). This means that remembered sets need be stored only for inter-partition pointers rather than every single pointer, thereby reducing space overheads for metadata. The drawback is that the loss of per-object remsets means that progress of the train algorithm is made most efficiently by performing all available relabelling in a single partition batch-wise, i.e. during operation of the PGC. This arrangement means that relabelling progress in the Train Algorithm is tied to operation of the PGC and the partition selection policy which defines the order in which the PGC processes partitions.

Where partitions containing no relabelling work are processed, no progress is made. Likewise, if the partition selection policy exhibits starvation, the train algorithm will not be complete. The partition selection policy is therefore critically important to performance (throughput and timeliness) of the train algorithm and this chapter investigates a partition selection policy which

promotes relabelling progress, resulting in a predicted linear complexity to completeness instead of the quadratic or worse seen with other policies.

The collector presented in this chapter operates over a distributed, orthogonally persistent [10, 8] heap, DPMOS [24], designed for the purpose of performance experimentation. Implementing the lower layer of a persistent heap introduces the requirement of tracking pointers held in mutator caches; this chapter demonstrates how the Surf model and its proof of safety may be extended to fulfil this requirement.

In summary, this chapter presents a distributed persistent garbage collector as an instantiation of the Surf model, makes extensions where necessary to permit the caching of objects outside the system boundary, applies the Surf model to derive a partition selection policy and confirms the model's performance predictions regarding partition selection policies via experimentation. It is therefore demonstrated that the Surf model may be used to construct and analyse entirely new garbage collectors and that the model not only provides implementors with a concrete list of requirements that must be satisfied to achieve correctness, it may also be used to provide insight into the design of operational policies for such new collectors.

## 5.1 Mapping

The table below summarises the design of the DPMOS collector by presenting it as a point in the Surf design space, following the instantiation process of Section 3.5. Entries in the table below define the design decisions in each dimension of the model and specify how the implementation satisfies the proof requirements. Detailed descriptions of these design decisions follow the table.

| **Model Concept** | **Mapping to Trains** |
|---|---|
| Direction | Progress is by relabel-target. |
| Regions | Regions are trains and are identified by integer names<br>Regions are ordered, preventing cyclic relabelling. |
| Relabelling job | Pointers from the root constitute relabelling work into the youngest candidate region.<br>Pointers from a younger region to an older region constitute relabelling work into the younger. |

| Model Concept | Mapping to Trains |
|---|---|
| Isolation DTDA | The isolation of every candidate region is represented by a job; each inter-region remset entry constitutes a task of the region isolation job.<br>Isolation is detected by Safra's algorithm [34]. |
| Remset DTDA | Pointer tracking is performed by Task Balancing [50, 82]. |
| Safety & Completeness Requirements | There are always at least three regions: a virtual mutator-train plus two others; all non-mutator trains are candidates.<br>The unwanted relative problem exists because objects are relabelled into candidate regions; train closure is used to ensure safety.<br>All non-root objects are always in candidate regions, so the initial-conditions requirement is fulfilled if regions are created to fulfil the three-region minimum.<br>Relabelling work is discovered by sampling remembered sets, therefore some entries must be preserved to maintain liveness of relabelling. |

### 5.1.1 Direction of Progress

Progress in the train algorithm is by relabel-target, i.e. it makes progress with waves of relabelling travelling forwards — away from the root, in the same direction as pointers — through the object graph. It is a single-hypothesis collector, operating from the hypothesis (actually an axiom) that the root is reachable.

### 5.1.2 Definition of Regions

Regions are trains, they are identified by integers, on which there is a complete ordering. Because there are more than two regions, it is not necessary for a non-candidate region to become a candidate, therefore the collector need not operate with global phases and synchronisation. To achieve this, it is necessary that no object that may be garbage at some point ever be in a non-candidate train, therefore the root exists in a virtual **mutator train** which is by definition younger than every other train in the system. There is one **active train** at each site which

is the youngest train and objects reachable from the mutator train are relabelled into the active train.

The definition of younger is based on the numerical ordering of train identifiers; trains with a higher ID are considered younger. As trains are reclaimed due to isolation or having all objects removed from them, newer, younger trains are created. Each train has a defined home site which is responsible for the maintenance of that train; the train ID is a composition of the home site ID (lower bits) and a sequence number maintained at that site (upper bits).

Trains are organised into a ring topology and use the train-maintenance protocols defined by Lowry [64].

### 5.1.3 Relabelling Job

Pointers from younger trains to older trains constitute relabelling work and indicate that the target (in the older train) should be relabelled into the younger train. Objects are never relabelled into the mutator train, therefore where a pointer from the root (in its virtual train) exists, the relabelling work is into the active train.

Because relabelling always causes objects to enter trains that are candidate regions, the collector suffers from the unwanted relative problem and synchronisation is required.

Collection progress is made entirely by the relabelling job, therefore analysing how much relabelling is required to reach completeness will reveal the complexity of the collector. In the Train Algorithm, reaching completeness requires that garbage components do not span train boundaries, i.e. each component is collapsed into a single train though they need not be separate trains, and that all live objects are relabelled out of trains that contain garbage.

Consider a worst-case scenario where live and dead objects are intermixed across a number of trains and an empty active train exists into which all live objects will be relabelled. Each object can be reassociated at most once across each train boundary. Since there is garbage in all trains, each live object will be relabelled up to $O(train\,count)$ times to reach the active train. Assuming that all garbage is strongly connected, collapsing the garbage into a single train requires each garbage object to be relabelled up to $O(train\,count)$ times.

The net complexity for the collector to reach completeness is therefore $O(object\,count \times train\,count)$ relabelling events.

### 5.1.4 Isolation DTDA

Isolation of trains is detected by Safra's Algorithm, an asynchronous ring-wave DTDA. The mapping is such that each train has a DTD job and every object with a non-zero inter-train remset constitutes a task of the job. Termination of a train's job indicates that no objects within the train are reachable from without and the train is therefore isolated. See Section 3.4 for a more detailed explanation of how remembered-set jobs (the "lower level" DTDA) can act as tasks of the isolation ("upper level") DTDA.

### 5.1.5 Remembered Sets

Pointer tracking in the remembered sets is implemented by Task Balancing. Each object has a job wherein inter-region pointers to that object are tasks of the job, termination of the job implies that the object is not reachable from outside the train. Because the collector is extended for use as the lower layer in a persistent heap, the pointer tracking algorithm is extended for this purpose as described in Section 5.2.4.

### 5.1.6 Safety Requirements

Because the train algorithm suffers from the unwanted relative problem, synchronisation between the relabelling (reassociation) process and region isolation DTDA is required. Because this system uses a wave-like DTDA, Lowry's model of train-closure is used to implement the synchronisation and thereby obtain safety. The drawback of this approach is that synchronisation covers a greater scope (i.e. more sites are interrupted) than it would if Norcross' [82] Doomsday request-for-witness approach were applied. However, the synchronisation is required only when a train closes due to being isolated at at least one site, as opposed to when an otherwise-unsafe relabelling occurs; the ratio of the rates at which these events occur is not known.

The actual performance difference between these two approaches, if any, is beyond the scope of this thesis.

### 5.1.7 Completeness Requirements

Because remembered sets are sampled on a discrete-time basis as the source of relabelling work, it is possible that no particular site will observe the presence of an inter-train pointer rapidly moving amongst many objects in a cycle. To solve this problem, an approach similar to the sticky remembered sets of DMOS [49] is taken: where an isolation detection wave fails, an additional list of work is

maintained containing erased inter-train pointers to that train. This is a form of snapshot at the beginning that operates where the train shows as live yet no work is visible at the times where the remsets are sampled to determine work.

### 5.1.8 Support Protocols

Protocols are required to maintain train membership; given that trains have a ring topology in this collector, the protocols of Lowry are used.

### 5.1.9 Extensions beyond the Model

The Surf model is a closed-world approach, i.e. it assumes that reachability is defined only by pointers within the system and that no external items may influence the reachability of any particular object. In the case of a persistent store, this assumption is not valid because objects may be copied out into client caches and modified therein; a persistent collector must remain aware of objects and pointers in the caches if it is to remain safe and it must be aware of when those pointers and objects are purged if it is to remain complete. The Design section below describes the implementation architecture in more detail and provides a protocol designed to correctly track cache contents.

For efficiency reasons, DPMOS is a compound collector, composed of a partition collector and the train algorithm. The partition collector exists to rapidly reclaim acyclic garbage and its operation is co-opted to perform relabelling between regions. Each object exists within a partition and each partition is assigned to a region; relabelling of an object in DPMOS therefore requires that an object move between partitions. The partition collector is a copying collector; it copies every object reachable from that partition's remset into some other partition and reclaims the space previously occupied by the newly evacuated partition.

Objects not reachable from the remsets associated with that partition are discarded (not copied), objects reachable only from the same or older train are copied to a partition in the same train and objects reachable from younger trains are copied to the youngest train from which they are reachable, thereby satisfying the relabelling job definition. By binding relabelling to the partition collector, progress of the train algorithm is dependent on partition collector operation, i.e. it will make progress only where the partition collector is operating. The order in which partitions are selected for collection at each site will therefore define the progress that train algorithm makes and this is investigated in more detail below.

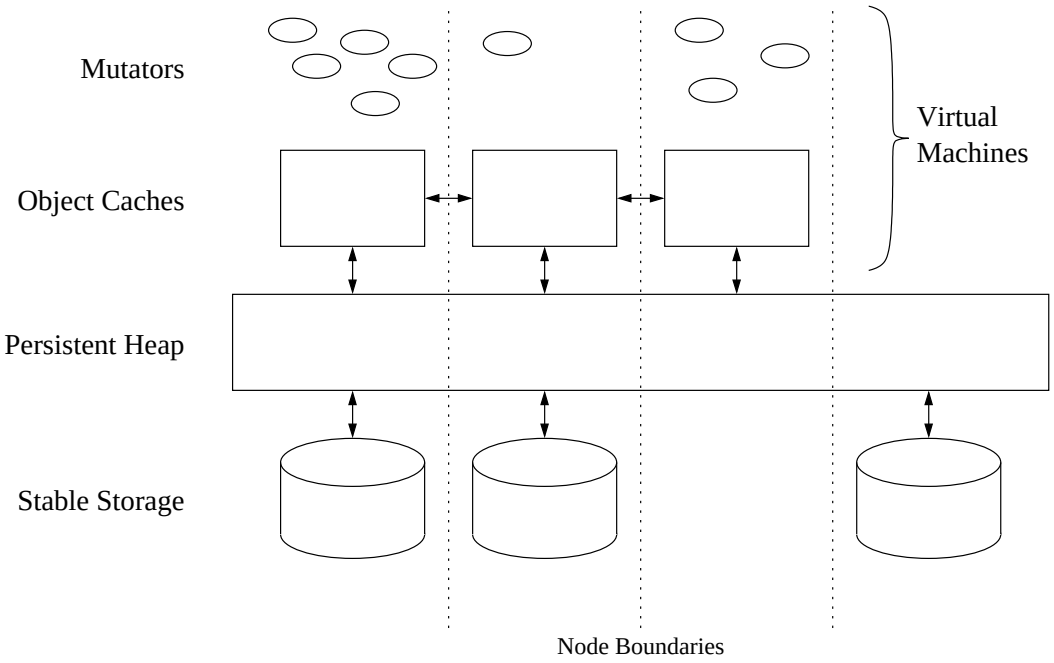


Figure 9: DPMOS Architecture

## 5.2 Design

This section describes the design of DPMOS as it relates to this thesis: instantiation of the abstract model, extending the mapping to account for pointers held by client caches (i.e. outside the system boundary as defined in the abstract model) and how the primary collector, trains, is composed with a partition collector for performance reasons.

### 5.2.1 System Architecture

The persistent garbage collector is one part of an orthogonally persistent system. The architecture of this system is broadly illustrated in Figure 9, showing the multiplicity of both stable storage sites and mutators in the system, all linked together by the distributed persistent heap.

Each mutator is a thread executing on a modified von Neumann machine where the storage abstraction is that of an object store rather than unstructured linear memory. The stable storage layer provides a checkpointing abstraction over linear memory at each site by applying shadow paging techniques. The persistent heap layer provides a heap abstraction by unifying all of the individual stable storage instances into a single distributed store; it is responsible for ensuring consistency between stable storage sites with respect to checkpoints. The data distribution model is a "shared-nothing" approach, i.e. objects exist on a single site and are not replicated within the persistent storage layer; the only replication within the system occurs at the client-cache level. The shared-nothing

approach was chosen because it conforms to the Surf system model and does not require additional functionality to ensure coherency of updates between replicas.

The distribution of the persistent heap layer requires distribution of the garbage collector in that layer; the design here follows the assumptions of the system model described in Section 3.1, namely that objects do not span site boundaries and do not migrate between sites. The system implements orthogonal persistence, i.e. persistence-identification is by reachability and any object may become persistent, regardless of its type. The update model is cooperative with checkpoints defining consistent states: mutators may create, read and write objects at any time via their caches but such changes do not become durable until a checkpoint occurs.

### 5.2.2 Mutators

DPMOS is programming-language agnostic, though it does enforce a particular object format with all pointers grouped at the front of the object. It should be possible to adapt the object format to any programming language by transforming objects when they are copied to and from the caches or modifying the compiler.

The implementation was tested using the ProcessBase [75] orthogonally persistent language; test applications implemented include OO7 [26] and Finite Element Analysis (FEA). The latter application was chosen because it generates large mesh-like data structures that are strongly connected and therefore require completeness of garbage collection is required if they are to be reclaimed.

### 5.2.3 Caches

Mutators operate on cached copies of objects: before an object may be read or written, it must be copied to a mutator cache. There may be any number of mutator caches in the system, they may be located at sites containing stable storage or not; therefore $n$ persistent storage sites serve objects to $m$ client/mutator sites. It is perfectly valid to instantiate this system with either or both of $n$ and $m$ being one though each must be greater than zero.

Cache coherency is the responsibility of the mutators and caching layer: they must cooperate to ensure data consistency. This simplifies construction of the persistent store while not constraining the cache-coherency implementation with requirements from the persistent store.

The assumption behind the use of client caches is that where a client modifies an object, it is possible that many such modifications will be made and it is desirable for performance reasons to hide these modifications from the persistent

heap while ensuring that the safety and completeness of garbage collection are not compromised. It is also possible that newly-created objects may reach end-of-lifetime, i.e. become unreachable, before their first checkpoint in which case it is desirable that such objects never reach the persistent storage layer. In addition to providing higher throughput and lower latency storage for mutators, client caches also reduce the load on the persistent heap layer, thereby permitting more computation time, network bandwidth and stable storage throughput to be used to perform garbage collection.

Guaranteeing correctness of garbage collection in the persistent heap in the face of objects and pointers being held in caches requires that the persistent garbage collector be aware of such pointers and objects held by client caches. A spectrum of approaches exists, varying in the level of information required from the objects caches:

- Naive: track all mutations.
- Transactional: exact knowledge of liveness.
- GC-based: track **presence of pointer** and **presence of object.**

The naive approach is to make the persistent heap aware of all mutations in the client caches and from this information, determine the reachability of objects in the client caches. This option is not considered further because the pointer tracking overheads it implies negate the performance benefits of the object caches; by making the persistent heap aware of all accesses, the cache does not reduce the latency of access to or load on the persistent heap.

The transaction approach presented in [4] presumes that modifications occur inside short-lived transactions and permits the exact determination of reachability from a transaction. Without ACID transactions [45], this approach is not applicable to the DPMOS system.

The GC-based approach is to have the caches inform the persistent heap when an object leaves the heap and when it no longer holds any pointers to a particular object; i.e. the cache summarises all mutation events and reports when the liveness of an object as observable from that particular cache changes. This results in a temporarily conservative estimate of liveness at low cost: it makes use of garbage collection inside the object caches to determine when an object is no longer held.

To achieve this summarisation effect, DPMOS object caches use **pointer swizzling** [77, 98], i.e. they selectively translate pointers from the persistent heap format (persistent identifiers, PIDs) to native machine addresses inside the cache. An additional aim of swizzling is to speed up object accesses: following

a machine pointer is faster than searching for the location of an object in some data structure from its PID. Swizzling pointers to frequently-accessed objects can therefore be beneficial to mutator performance because it reduces the cost of pointer dereference operations. Pointer tracking in DPMOS takes advantage of swizzling by not tracking swizzled pointer copies and erasures: it is necessary only to know when there are zero swizzled pointers to any given object and this information is provided by the cache collector. The details of the tracking algorithm are provided below.

### 5.2.4 New Cached-Pointer Tracking Algorithm

The DPMOS cached-pointer tracking algorithm tracks two things: presence of each object in each cache (a boolean) and the number of unswizzled pointers to each object in each cache (an integer). Where an object is present in a cache, the exact number of swizzled pointers to that object is immaterial since it may not be reclaimed by the persistent heap. Since the majority of pointer manipulations by mutators concern the copying and erasure of swizzled pointers, the majority of mutation events are of no interest to the persistent heap and therefore incur no pointer tracking overhead. Once the number of swizzled pointers reaches zero, the collector in the cache will discover this, purge the object and notify the persistent heap.

Each object has the following record associated with it (one per cache) where necessary:

```
typedef struct {
    int count;
    bool swizzled;
} cref;
```

A `std::vector<std::map<pid_t, cref> >` stores all necessary cache pointer tracking information at a given persistent heap site. The outer `vector` is indexed by the site ID of cache that refers to a given object, the inner `map` permits lookup of a `cref` entry from an object's PID. This data structure is stored inside and maintained by the persistent heap; entries are present only where their contents are non-zero. This data structure is physically separate from the persistent heap remembered sets which exist to record inter-partition pointers within the persistent heap. This data structure is volatile: in case of crash or shutdown, it is discarded because the cache contents are discarded.

Table 13 summarises the events that are of interest to the cached-pointer tracking algorithm and the operations that are performed for each event.

| Event | Actions |
|---|---|
| Read | **Object is copied to cache**, therefore:<br>`swizzled=true` for the object in question,<br>`++count` for every contained pointer, and<br>all required actions occur inside persistent heap, no additional communication required. |
| Swizzle | **Pointer in cache is dereferenced and converted to a machine address**, therefore:<br>will first require a **Read** event if the target is not already in the cache,<br>cache sends a "minus" message to persistent heap, and<br>persistent heap will `--count` for that PID. |
| Deswizzle | **Pointer in object is converted back to a PID**, therefore:<br>cache sends a "plus" message to persistent heap, and<br>persistent heap will `++count` for that PID. |
| Expunge | **Object is removed from the cache**, therefore:<br>**Deswizzle** (as necessary) then **Erase** events are performed for each contained pointer,<br>"expunged" message is send to persistent heap, and<br>persistent heap sets `swizzled=false` for the object expunged. |
| Copy | **PID is copied inside the object cache or transmitted to a different cache**, therefore:<br>cache sends a "plus" message to persistent heap,<br>persistent heap will `++count` for that PID. |
| Erase | **PID is erased (overwritten) inside the object cache**, therefore:<br>cache sends a "minus" message to persistent heap, and<br>persistent heap will `--count` for that PID. |
| Write | **An object is written back to the persistent heap**, therefore:<br>no action is required from the cache because the contents of the cache are unchanged, and<br>"plus" events for the persistent heap's internal pointer tracking algorithm are generated. |
| Create | **A new persistent object is created**, therefore:<br>persistent heap allocates a PID with `swizzled=true; count=0;`<br>cache must request the creation of the object, PID is returned to cache. |

Table 13: Pointer Tracking Events

As noted in each event, all events except the **Read** occur inside the object cache, therefore messages must be sent to the persistent heap to notify it of changes in the caches. In the case of **Read**, the persistent heap is directly involved (it has the data being read), so no such notification is required. In the case of **Write**, communication is required to perform the actual write operation (the cache sends new values to the heap) but because the cache is unchanged, no pointer tracking communication is required. The receipt of the write request at the heap will cause pointer modifications inside the heap and therefore persistent pointer tracking events as per the Surf model.

This algorithm maps to the DTD problem in the same manner as Remembered Sets (Section 3.3.4) except that here we track only pointers in caches and optimise the swizzled-pointer case. For the purposes of this mapping, we assume that the object is at one site, other objects are at other sites and that every cache is at a different site; the analysis still holds in the cases where they are congruent.

This pointer tracking algorithm is identical to the Task Balancing algorithm first presented in DMOS except that all tasks due to swizzled pointers at a given site are collapsed into a single task. Therefore there exists a task for each PID in a cache and a task for each object in a cache and every swizzled pointer thereto. Correctness of this pointer tracking algorithm depends on it integrating correctly with the pointer tracking algorithm used within the persistent store; showing that this interaction works requires only that a single DTDA Job may exist and contain multiple kinds of task. One kind of task may witness the birth of a different kind: after all, a DTDA knows not that it is tracking different kinds of task, all that matters is that the assumptions of the DTD system model are observed.

This section therefore proves the correctness of this pointer tracking algorithm by describing the behaviour of pointers and objects in caches in terms of the DTDA system model. By showing that their behaviour conforms to the DTD abstraction, we know that a DTDA may track the existence of these pointers and objects and therefore that the use of Task Balancing in this implementation is safe and complete.

Three types of task are defined: $Tp$, $Ts$ and $Tc$, representing persistent pointers, cached objects and cached pointers respectively. The table below defines all the ways in which these tasks may be created and how the Doomsday cover rule (the requirement that task birth at sites other than the home site is witnessed by an existing task) is satisfied for every way in which a task may be created.

| **DTD Concept** | **Meaning wrt Remembered Sets & Pointer Tracking** |
|---|---|
| Job | Each object has a Job representing its direct reachability from objects within the persistent heap or any object cache; the home site of the Job is the location ($P$) of the object in question:<br>$J_x \models x_P$ |
| Task | Each pointer in the heap to the object of interest is a task of that object's Job, $Tp$ denotes a "persistent pointer" task:<br>$Tp^n_P \in J_x \models y.x \mid y \in persistent\,heap$<br>Each PID (unswizzled pointer) in a cache to the object of interest is a task of the job, $Tc$ denotes a "cached pointer" task:<br>$Tc^n_Q \in J_x \models z.x \mid z \in cache_Q$<br>Each copy of the object in a cache and all swizzled pointers to it is a task of the job, $Ts$ denotes a cached object and all swizzled pointers thereto:<br>$Ts^n_Q \in J_x \models copy(x) \in cache_Q$ |
| Birth $Ts$ | The first pointer to an object is created when a cache requests the creation of the object itself. The object is created at $P$ and, a pointer to it is created at the same site and sent to the cache, i.e. the very first task created is a $Ts$ and it is created at the home site and subsequently migrates to the cache.<br>$birth(Ts)_P \models create$<br>The only other means to create a $Ts$ is to read an object into the cache. In this case, the $Ts$ is created at the home site (where $x$ is read from) and migrates to the cache:<br>$birth(Ts)_P \models \mathbf{Read}(x)_P$ |

| **DTD Concept** | **Meaning wrt Remembered Sets & Pointer Tracking** |
|---|---|
| Birth $Tc$ | A $Tc$ for $J_x$ may be created during a read operation on $z$ where $\exists z.x$, in which case the birth is witnessed by the $Tp$ that must exist due to $z.x$:<br><br>$birth(Tc)_Q \models \mathbf{Read}(z)_Q \mid (\exists z.x \Rightarrow \exists Tp_Q)$<br><br>A $Tc$ for $J_x$ may also be created by pointer copying inside a cache, in which case the new $Tc$ is clearly witnessed by the existing $Tc$ at the same cache site:<br><br>$birth(Tc^n)_Q \models PID\,copy \mid \exists Tc^m_Q$<br><br>Finally, a $Tc$ may be created by deswizzling a reference; the existence of the swizzled reference implies the existence of a $Ts$ at that cache:<br><br>$birth(Tc)_Q \models deswizzle \mid \exists Ts_Q$<br><br>Swizzled pointers may be created only where the object has been read in, i.e. they are part of a $Ts$ and not individually tracked. |
| Birth $Tp$ | $Tp$ represents a pointer inside the persistent heap; they are created only when a cache writes an object back to the heap. This is represented by the $Tp$ being created inside the cache (witnessed by a $Tc$) and then migrating to the appropriate site in the persistent heap:<br><br>$birth(Tp)_Q \models \mathbf{Write}(z)_Q \mid (\exists z.x \Rightarrow \exists Tc)$ |
| Migration | Pointers may travel across the network in messages, which is modelled by the migration of tasks:<br><br>$send(T^n)_P \models send(y.x \rightarrow Q)^k_P$<br>$recv(T^n)_Q \models recv(P \rightarrow z.x)^k_Q$ |
| Death | Each task ends when the relevant pointer is erased:<br><br>$death(T^n)_P \models minus(y.x)^k_P$ |

| DTD Concept | Meaning wrt Remembered Sets & Pointer Tracking |
|---|---|
| Termination | The lack of pointers to an object is equivalent to the lack of Tasks in (termination of) the Job: $\neg\exists T^n \in J \models \neg\exists y.x$ |

To summarise, this design of DPMOS shows how the presence of formality in the Surf model serves as a framework to extend that same formality to systems that do not directly conform to the Surf system model. In other words, the very formality of Surf makes clear the requirements that must be fulfilled if Surf is to be extended and its formality maintained; this is in contrast to an informal or mechanistic definition of garbage collection. DPMOS extends the Surf model for use in the lower level of a persistent heap by extending the Remembered Set formality to include the tracking of pointers not only in the persistent heap but also in object caches. The newly created pointer tracking algorithm is made efficient by collapsing all swizzled pointers into a single task for each object, thereby reducing the number of tasks, the rate at which task birth and death occurs and therefore the load on the pointer tracking algorithm.

### 5.2.5 Partition Collector

Though the Train Algorithm is complete, its timeliness of reclamation for acyclic garbage is non-optimal since it may take $O(n)$ steps to remove live objects from a train containing garbage. On the assumption that applications will generate (perhaps significant quantities of) acyclic garbage, the store is partitioned into **cars** and a partition garbage collector (PGC) implemented.

Partitions each contain a number of objects but do not span region or site boundaries. Objects are physically stored inside partitions, i.e. partitions are defined by a contiguous region of stable storage space. The reason for contiguity is to minimise load-latency from persistent storage by reducing the number of seeks to one on the assumption that the backing store is disc. Contiguity of storage improves access performance on disc because seek time is much greater than that required to read a single block: contiguous reads are much faster than non-contiguous reads.

The mapping presented in Section 5.1 implies that each object has its own remembered set, representing all pointers to that object; partitioning the store reduces this requirement to maintaining remset entries only for inter-partition pointers. The PGC considers all remset entries for objects in that partition as

roots of collection and is therefore incapable of reclaiming inter-partition cycles. Objects that are trivially unreachable will be reclaimed at the PGC invocation immediately after they become so; the destruction of pointers they contain may result in other objects becoming trivially unreachable and therefore reclaimable by the PGC. Repeated invocation of the PGC will therefore reclaim all acyclic garbage and all cycles of garbage that do not span multiple partitions.

### 5.2.6 Progress and Partition Selection

Because objects are stored within partitions and partitions do not span region boundaries, the relabelling of an object in DPMOS requires that it be physically moved between partitions. This operation is referred to in the Train Algorithm literature as **reassociation** and is bound to execution of the PGC, i.e. an object may be reassociated only during execution of the PGC in the partition that the object is to be removed from. A further advantage to this approach is that reassociation follows the relabelling rules of the mapping, i.e. an object is relabelled only to a younger region containing a pointer to that object and the presence of these pointers from younger regions is indicated by the remembered set. Deciding where to relabel an object requires an analysis of its remset entries (if any) and the remsets of other objects in that same partition that can reach the object in question, e.g. if $y.x$ is the only pointer to $x$, $partition(x) = partition(y) \wedge relabel(y \rightarrow L) \triangleright relabel(x \rightarrow L)$; this information is known only during operation of the PGC because $x$ (not being directly reachable from outside the partition) does not have its own remset. Performing relabelling during execution of the PGC means that whole sub-graphs reachable from a particular remset entry (representing a pointer from a younger region) will be relabelled batch-wise.

Efficiently fulfilling the relabelling rules of the mapping requires that an object is relabelled to the youngest region from which it is apparently reachable. The PGC operates as a tracing collector:

- remsets are sorted in order of increasing source-region age,
- tracing proceeds from each remset in order, visiting each object once only:
  - for remset entries due to pointers in younger regions, the objects discovered by the trace are relabelled to the relevant younger region,
  - for remset entries due to pointers in the same or older regions, objects are copied to another partition in the same region,
- all potentially live (not trivially unreachable) objects have now been copied from the partition,

- pointers in remaining objects are considered erased and remset-update (task death) messages are sent accordingly, and

- the partition's space is freed.

By performing relabelling as part of PGC execution, progress of the Train Algorithm now depends on fairness of the **partition selection policy**, i.e. that which decides the order in which partitions are selected to have the PGC executed thereon. At a minimum, the policy must not exhibit starvation, i.e. every partition containing relabelling work is eventually selected for processing. A naive fair policy is to process partitions in order, which results in poor complexity-to-completion of the collector. Considering that Section 5.1.3 shows that $O(object\,count \times train\,count)$ relabelling events are required to reach completeness and that an in-order partition selection policy may discover only a single item of relabelling work per pass through a train, $O(1)$ to $O(train\,count)$ relabellings may be performed per lap ($O(object\,count)$ PGC invocations) of the store. The resulting complexity to completeness with a naive in-order partition selection policy is therefore somewhere between $O(object\,count^2)$ and $O(object\,count^2 \times train\,count)$ PGC invocations.

Previous research into partition selection [31] has focused on heuristics aimed at reclaiming acyclic data structures by selecting partitions that contain objects that are the targets of erased pointers. This approach is not suitable where cyclic data structures are concerned because such an approach does not promote progress of the complete garbage collector, i.e. the Train Algorithm, as demonstrated by Munro *et al* [80].

### 5.2.7 Train-Centric Partition Selection

The purpose of this section is to use the model of work and progress provided by the Surf model to select partitions that maximise work performed by the Train Algorithm. According to the model, relabelling work is defined by the presence of a pointer from a younger region to an older region; processing this work results in progress by relabelling the target into the region containing the source.

Batching objects into partitions and processing partitions in an arbitrary order introduces the possibility of the Train Algorithm making no progress during a particular PGC invocation and the partition selection policy should seek to avoid this situation. A partition selection policy that achieves $O(object\,count \times train\,count)$ complexity that is available from the Train Algorithm is considered **high quality**, while a policy that fails to enable progress is considered **low quality**. If a policy can ensure that each object is relabelled only once, directly

into the train in which it must finally reside, the policy is referred to as **oracular** and results in a complexity to completeness of $O(object\,count)$.

It is possible to implement a high quality train-centric partition selection policy because information describing available work is already gathered by the system in the form of remembered sets. Any remset entry indicating the presence of a pointer from a younger region indicates that there is work available in the target partition; selecting that partition will ensure that non-zero progress is made for every PGC invocation.

DPMOS uses the Train-Centric policy, defined below. Variations on this policy based on constraining the order in which trains are collected are explored and the drawbacks of each examined using the Surf's model of progress. The choice is between train-centric with no major drawbacks, an optimal but unimplementable policy and two other policies with better complexity to completeness in the short term but poor long-term performance.

#### 5.2.7.1 Train Centric Policy

The policy implemented by DPMOS is to select the partition that contains the greatest number of remset entries from younger trains: this younger-remset size is used as an approximation for the number of objects in a partition that are reachable from younger trains. When the PGC is to be invoked at a particular site, the partition chosen is that with the greatest younger-remset size. Making the pessimistic assumption that a single object will be relabelled per PGC invocation, this policy results in $O(object\,count \times train\,count)$ PGC invocations to reclaim all garbage in a store.

The train-centric policy does not define the order in which partitions are processed with respect to their train membership, it inspects only the remset size. Where the inter-partition connectivity is greatest, progress will be made earliest and the ordering of this progress will define how many times a given object will be relabelled. Where progress is made earlier in youngest trains, fewer relabellings are required because objects will be relabelled directly to the train in which they would finally reside. Conversely where progress is made earlier in the oldest trains, each object may be relabelled as many times as there are trains. The exact ordering in which progress is made means that the throughput achieved by this policy is somewhere between $O(object\,count)$ and $O(object\,count \times train\,count)$ PGC invocations required to reach completeness.

Figure 10 illustrates progress of complexity $O(object\,count)$ where relabelling is performed from younger trains first; each row in the diagram represents a train

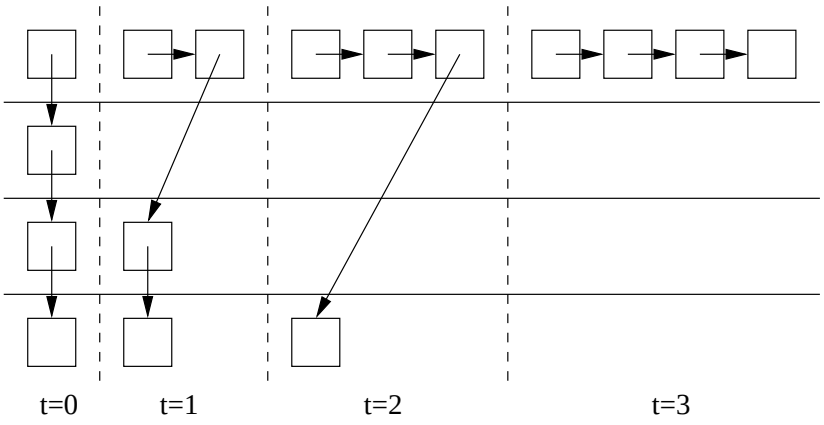


Figure 10: Progress by Younger-First

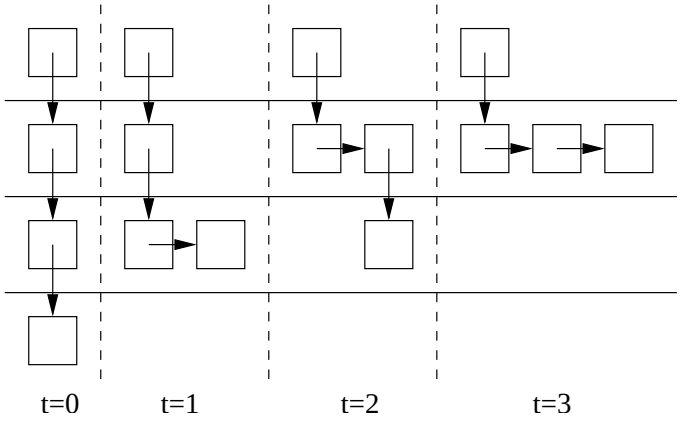


Figure 11: Progress by Older-First

with the younger trains higher in the diagram and each column represents a time-step. Conversely, Figure 11 illustrates progress of complexity $O(object\,count^2)$ where relabelling is performed from older trains first.

It seems obvious that progress in the youngest trains first is preferable due to the reduced complexity but this is a poor approach in the long term, as described below. While it is true that any policy will eventually collapse the live graph into a single train, that is only a drawback for a strict youngest-first policy that reclaims no garbage until the live graph is collapsed. By not constraining operation to youngest-first, the creation of new trains can occur and results in spreading the live graph across multiple trains. The train creation rate must be low enough that starvation of partition selection does not occur in the older regions; clearly there will be some negative impact on the complexity to completeness compared to the case where no train creation occurs. This thesis does not investigate train creation policies in detail.

#### 5.2.7.2 Alternative: Youngest-First Train-Centric

It is possible to define a **short-term oracular** policy, i.e. one which exhibits oracular performance for the garbage that exists in the store, by always executing the PGC in the youngest train for which there exists work. This is referred to as the youngest-first train-centric policy. Because work is always performed in the youngest trains available, objects will never be relabelled to some intermediate train, only to be relabelled again later; see Figure 10. The result is that this policy will first collapse all live objects into the active train and then it will collapse the dead objects into their respective trains. Because no object is relabelled more than once, the $train\,count$ term disappears from the complexity, resulting in $O(object\,count)$ and seemingly oracular performance.

The drawback to the youngest-first train-centric policy is that it will reclaim absolutely no garbage until all live objects have been relabelled into a single active train. It will therefore harm the long-term scalability and performance of the train algorithm because a single active train will become global (all sites are members of it) and contain all live objects. Reclamation of any object that becomes garbage after entering that global train will require the relabelling, again, of every live object in the system to a new active train. Operation of this policy requires that no new trains are created until all garbage is reclaimed, because that would cause all relabelling to occur in the live graph with no progress in the dead regions. By requiring that the whole live graph be relabelled before reclaiming garbage, the timeliness of the collector is degraded to the worst-case as seen from Distributed Marking operating in phases (Section 4.1).

Youngest-first train-centric therefore has optimal complexity to completeness in the short term, i.e. for extant garbage, but poor long-term performance because of its tendency to collapse the live graph into a single train.

#### 5.2.7.3 Alternative: Oldest-First Train-Centric

The oldest-first train-centric approach selects partitions within a train according to their younger-remset size and processing trains from the oldest towards the newest. This policy will have the effect of reclaiming trains in order; see Figure 11 for an example. The oldest-first progress implies that garbage will be reclaimed before the live object graph is collapsed beyond what is necessary to separate it from the dead objects, resulting in a complexity to completeness of $O(dead\,count \times train\,count)$ however it will not be known that completeness is reached until the entire live graph is also processed, in $O(object\,count \times train\,count)$ time.

This policy does not suffer train-creation-induced starvation because newly created trains are the youngest in the system. The drawback is that newly created trains will contain nothing unless there are live objects in the very oldest train, in which case they will be moved to the active train. This policy will also therefore tend to slowly collapse the whole live graph into a single train unless there are a large number of references in the root object to disparate parts of the graph. The difference between this policy and youngest-first is that youngest-first collapses the live graph before reclaiming any garbage while this policy reclaims garbage and then collapses the live graph. The steady state in each case is to have all live objects in one or two global trains.

#### 5.2.7.4 Alternative: Youngest-With-Garbage-First Train-Centric

Youngest-with-garbage-first train-centric is a **truly oracular** policy: it processes work according to the train-centric policy but only for trains containing garbage. This policy seems unimplementable because knowledge of which trains contain garbage is required. If it were implementable, it would first remove live objects from the youngest train containing garbage and then collapse all garbage reachable from that train into that train. The youngest-first approach means that garbage will be collapsed in $O(dead\,count)$ invocations instead of $O(dead\,count \times train\,count)$. Avoiding processing in the live region means that the live graph is not collapsed and the scalability of the train algorithm is not adversely impacted.

This policy is therefore optimal in that it requires $O(object\,count)$ PGC invocations to reclaim all garbage in the system; the drawback is that it seems unimplementable. Further research into heuristics for selecting a youngest-with-garbage train is beyond the scope of this thesis.

### 5.2.8 Mixed Partition Selection

Applying the train-centric partition selector would negate the benefits of the PGC in terms of its ability to reclaim acyclic garbage quickly. Therefore DPMOS uses a mixture of the train-centric policy described above and the acyclic-garbage-finding policy of Cook, Wolf & Zorn [31]. Each evaluation of the partition-selection policy evaluates both policies and selects the output of the policy that predicts it will make the most progress; an alternative is to invoke the policies at some ratio.

It should be noted that the two policies tend to select disjoint sets of partitions. Partitions constituting work for the Train Algorithm:

- contain mostly dead objects in a component that is collapsing; its deadness means that the mutator has no pointers to those objects that it may erase, or

- contain mostly live objects to which pointers have not been erased.

This means that such partitions are unlikely to be selected by the acyclic-centric collector. The disjoint nature of the outputs means that to obtain the benefits of both policies, both policies must be occasionally consulted; neither will eventually return the same output as the other.

It should be noted that the younger-remset size is only an estimate of the relabelling progress available within a partition because it is not known a-priori how large a component is reachable from each remset entry. Likewise, the size of an acyclic garbage component reachable from a single erased pointer is unknown, so the erased-pointer count for a partition is also an estimate of the quantity of acyclic garbage in that partition. Further implementation work not considered in detail here is to have the system build a statistical model of the progress attained due to the invocation of each policy as a function of the estimate produced by that policy; the usefulness of such statistics in deciding which policy to use at a particular PGC invocation depends on their being a good correlation between the predicted and actual progress from each policy.

Knowledge of the ratio of reclaimed garbage to estimated garbage for each policy and the precision of such estimates from each could provide the means to dynamically vary the rate at which each policy is consulted and thereby maximise progress of the system as a whole in reclaiming both acyclic and cyclic garbage.

While runtime analysis of partition selection accuracy is further work, an offline analysis of progress estimation is presented later in this chapter that confirms the accuracy of the train-centric partition selection policy, i.e. that the predictions of available progress correlate well with the progress obtained.

## 5.3 Experimentation

The DPMOS system has been implemented and tested with the ProcessBase language. Testing has focused on:

- confirming safe and complete operation and therefore the validity of the cached-pointer tracking algorithm,
- confirming linear complexity with the new partition selection policy and comparing timeliness with naively fair partition selection policies,
- confirming the correlation between predicted and obtained progress from the train-centric partition selection policy,
- investigating the costs of collection with respect to object instantiation policies, and

- investigating scalability with respect to highly cyclic data structures.

Safety and completeness cannot be proven conclusively by empirical testing; running an arbitrarily large collection of complex tests proves only that the collector is correct with respect to those data structures. Such testing exists only to provide a level of confidence in the system with more extensive testing providing a higher level of confidence that the implementation matches the instantiation described in this chapter; the tests used in this chapter confirm that the collector is correct with respect to highly cyclic mesh data structures.

Each of the testing phases is described in detail below and example results presented for each where applicable. Testing was performed on the South Australian Partnership for Advanced Computing's clusters with OO7 tests run on `perseus` and the FEA tests run on `hydra`:

**`perseus`** 60 dual Pentium-III 500MHz with 256MB of RAM at each node and connected by switched fast ethernet.

**`hydra`** 128 dual Xeon 2.4GHz with 2GB of RAM at each node and connected by Myrinet.

The implementation was designed and constructed with the measurement of collector cost and performance in mind; the design approach is outlined below then individual tests and their results follow. The tests performed are:

- a comparison of partition selection policies with single-site OO7 and FEA meshes,
- an investigation of complexity to completeness with FEA meshes distributed over 4 sites, and
- an investigation of the train-centric policy's progress prediction accuracy with distributed OO7 and FEA meshes.

### 5.3.1 Architecture for Measurement

The implementation of DPMOS uses a layered architecture as shown in Section 5.2.1, making it a specific instance of a generalised layered architecture where higher layers make requests of lower layers; the lower layers provide mechanism that is used and controlled by the higher layers. Each layer provides a service to the layers above it by abstracting over the layers below and providing some additional functionality. This arrangement is illustrated in Figure 12; higher-numbered layers are higher in the system.

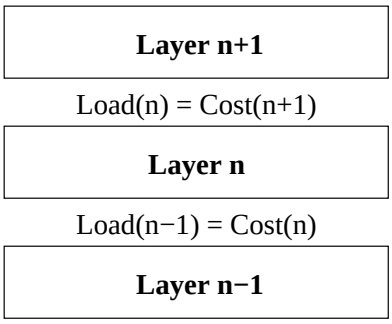


Figure 12: Layered Architecture for Measurement

According to this architecture, the communication between layer $n$ and $n+1$ is the **Load** placed on layer $n$ and also the **Cost** of layer $n+1$. In providing an abstraction, each layer has a cost that is a function of the load placed upon it and the mechanism by which it operates. The **performance** of a layer is defined as that function which transforms the load on a layer into the costs of that layer.

At the lowest levels of the system, the costs are physical, e.g. CPU time, network traffic, disc space, etc. At higher levels where the layers provide more abstraction, the costs become appropriately more abstract, e.g. objects instantiated. To illustrate the architecture by trivial example, consider a sorting algorithm with known complexity: bubble sort is $O(n^2)$ therefore its cost transformation function is:

$$CPUtime(item\_count) = f\,(item\_count)$$
$$f(n) = a \times n^2$$

The layered approach to performance analysis is taken in all parts of the DPMOS implementation: costs are measured at each layer, permitting the determination of performance of each layer by analysing the relationship between load upon and cost of that layer. The results presented in this thesis are coarse-grained, i.e. of the distributed persistent garbage collector as a single layer. For the purpose measuring garbage collection performance, the load represents the activity of the mutators and the object graph that results from that activity. The means by which collector cost is measured varies with the collector component that is being measured.

The relationship between graph topology and PGC invocation count is a succinct description of the partition selection policy only because it does not depend on other mechanisms within the collector such as how metadata is stored or how pages are cached therefore the relationship between graph size and topology and PGC invocation count is a good description of partition selection policy performance. Conversely, if a measure of collector performance *in toto* is required, more physical measurements such as page fault count, quantity of network traffic and CPU or wall-clock time consumed are more appropriate.

When comparing such physical measurements of cost with the load upon the collector (graph topology and mutator activity), the cost/load relationship describes the performance of the collector as a whole. The same measurements may also be used to analyse the performance of collector subsystems by using different concepts of load; for example, the performance of the block-caching subsystem may be measured as a function of the access request to page fault ratio as a function of the access patterns requested of it. Likewise, the performance of the subsystem that maintains metadata may be measured in terms of the block read/write requests made by that subsystem as a function of the pointer update rate.

This chapter presents only results that are relevant to the higher-level issues discussed in the thesis: correctness and a comparison of the complexity of collection with varying partition selection policies. Following sections describe the specific experiments performed in terms of the test loads, measurements made and hypotheses under test in each case.

### 5.3.2 Data Structure: OO7 Medium

The first test load is OO7 [26], a synthetic data structure intended to emulate an object-oriented computer aided design database. It is a tree structure with cyclic components at the leaves of the tree, therefore the ability to reclaim this data structure indicates that a collector is complete with respect to cyclic garbage.

OO7 is published with three defined sizes: `small`, `medium` and `large`. `small` contains a small tree, `medium` contains a larger tree and `large` contains multiple independent trees. The data structure used in this test is the `medium` configuration with a tree size of approximately 2000 partitions of 64kB, plus associated remembered set metadata. The test was performed with all data on a single site, i.e. without distribution.

### 5.3.3 Data Structure: FEA Meshes

Finite Element Analysis (FEA) is a class of scientific computation which divides space into a large number of small elements with finite extent. The spatial division is used as a set of samples over which a set of differential equations are solved numerically, for example:

- stress and strain in mechanical structures,
- electromagnetic fields in a variety of different media,
- fluid flow, and

- weather simulation.

The division of space as a means to solve differential equations is a very generic technique that may be used to simulate nearly any physical aspect of a system. Taking only the mechanical example where forces are simulated in a rigid body, some example end-user (where an end-user is typically a mechanical engineer) applications are:

- simulating deformation of a mechanical component under load,
- crack and tear analysis, and
- harmonic and vibration analysis.

The ability to simulate nearly any physical system that may be described by a set of differential equations makes FEA a very powerful tool for scientific computing. Its computationally intensive nature makes it desirable that it be implementable in a distributed system and the local nature of each computational step make it amenable to implementation in a distributed system. Each simulation step for a mesh element requires only the state of that element and its neighbours, therefore the mesh may be partitioned and each section allocated to a separate site in the distributed system. Communication is required only to exchange information as it crosses the partitioning of the mesh.

Mesh data structures are by their nature highly cyclic and therefore strongly connected. Reclaiming a mesh requires a complete garbage collector.

For most finite element analyses, the mesh topology is unchanging: the simulation merely varies the non-pointer values inside the mesh to indicate the physical state of each element and the connectivity between elements does not change. Therefore, a mesh as observed by a garbage collector is a very large strongly connected component with static topology.

For the purposes of testing the implementation in this chapter, FEA meshes of dimension two are instantiated, composed of a regular grid of triangles. Each vertex is an object and each triangle is an object; each vertex contains a linked list of pointers to the triangles it participates in and each triangle contains three pointers, each to a vertex. The mesh is instantiated in squares of 101x101 vertices, resulting in squares of $2 \times (101 - 1)^2 - 20,000$ triangles. Larger grids of these squares are created with 6 squares per train; the total number of squares varies between tests.

Figure 13 contains an example mesh of triangles with a 9x9 grid of vertices and therefore $2 \times (9 - 1)^2 = 128$ triangles. Figure 14 shows an example grid where an arrangement of 12 of the squares from Figure 13 have been sewn together

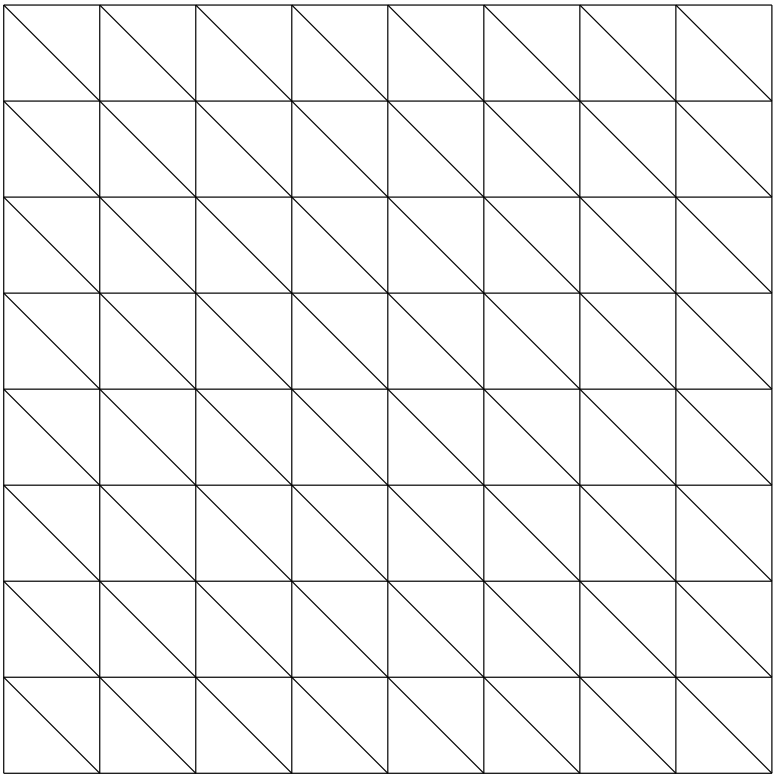

Figure 13: Mesh of Triangles

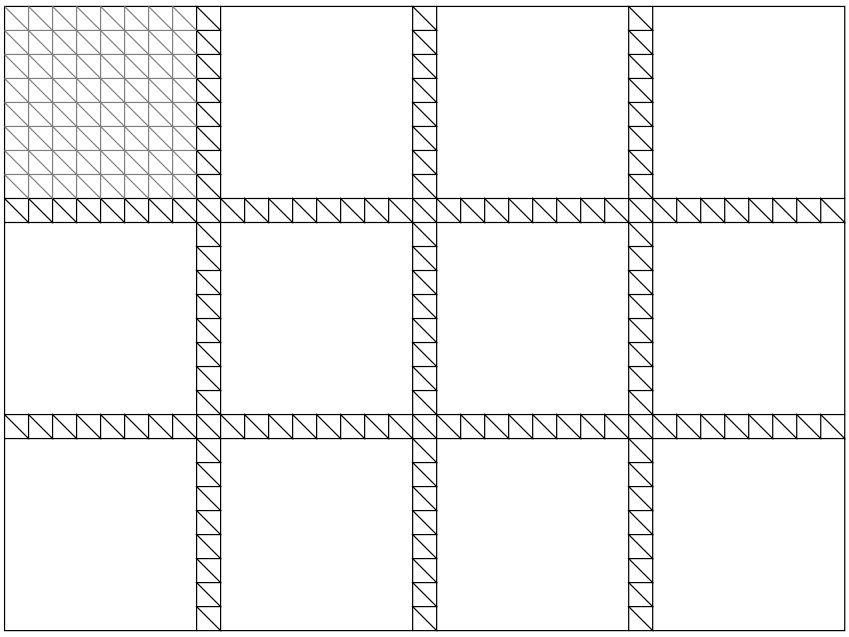

Figure 14: Grid of Meshes

into one larger mesh using additional triangles, resulting in one large strongly connected component containing many cycles.

The experimental testing in this chapter involves instantiating large meshes, sewing them together, allowing them to become garbage and then measuring the cost of reclaiming them. What is being measured is therefore the cost of collapsing a large FEA mesh into a single train.

### 5.3.4 Experiment: Partition Selection with OO7

OO7-medium is used here to analyse partition selection performance and compare previously published partition selection policies with the train-centric policy of Section 5.2.5. The initial state of the store is with three trains: one containing the root object, one containing all other objects and one completely empty, i.e. the active train. To reach completeness, the collector needs only to move the (very few) live objects into the active train and it must not move any dead objects; analysis of this specific arrangement via the Surf model shows the best-case complexity to be $O(live\,count)$ since the dead objects are already collapsed into a single train.

The experiment was executed on a single site, the reason being that the previously published partition selection policies are so poor that they do not reach completeness in a timely fashion with larger or distributed data structures.

Six different partition selection policies were tested; in each case the expected complexity is calculated on the assumption that the number of partitions in which relabelling work is available is tiny compared to the total number of partitions.

- Round Robin: partitions are selected in address order; expect $O(live\,count \times partition\,count)$,
- FIFO: partitions are considered in their order of creation; expect $O(live\,count \times partition\,count)$,
- Reverse FIFO: partitions are considered in the reverse order of their creation; expect $O(live\,count \times partition\,count)$,
- Random: partitions are pseudo-randomly selected; expect $O(live\,count \times partition\,count)$,
- Updated Pointer: the partition that is the target of the most pointer deletions is selected, as per [31]; expect starvation, and
- Train Centric: the new policy of Section 5.2.7; expect $O(live\,count)$

| Partition Selection Policy | PGC Invocations | Page Faults |
|---|---|---|
| Round Robin | 8210 | 23845 |
| FIFO | 8209 | 22411 |
| Reverse FIFO | 16126 | 980653 |
| Random | 14873 | 901436 |
| Updated Pointer, 10:1 | 17847 | 572685 |
| Updated Pointer, 100:1 | 30906 | 1077830 |
| Train Centric | 20 | 12539 |

Table 15: Reclamation Cost to Completeness; Single Site, Single Train OO7

The collector was run to completeness with each of the policies and the cost of reaching completeness measured in terms of the number of PGC invocations and 64kB-page faults incurred.

Results are summarised in Table 15. Note that UpdatedPointer is not a fair policy and not guaranteed to make progress, therefore it is mixed with the FIFO policy at varying ratios to ensure fairness; these ratios are listed in the table of results.

There are two results of significance from this experiment: that the collector is safe and complete with respect to OO7 when a non-starving partition selection policy is applied and that the new partition selection policy is unique amongst the policies tested in its ability to promote progress of the Train Algorithm. Reaching completeness shows that the relabelling process is operating correctly: it never relabels an unreachable object to the active train and it relabels every reachable object.

Given that the FIFO and Reverse FIFO results made four and eight laps of the store respectively, this is an illustration of the $O(n^2)$ nature of the Train Algorithm in the face of naive partition selection. Specifically, there is a small chain of live data which must be traversed in a particular order by the collector for it to be relabelled; any PGC invocations outside of that order are entirely wasted. When traversed in one direction (FIFO), four passes of the structure are required to relabel all live data while in the opposite direction (Reverse FIFO), eight passes are required. This is a demonstration that the collector makes progress as a wave over the graph in a specific direction and that a simple cycle has **handedness**: the pointers travel in a certain direction around the cycle. The ability to make progress in only one or two partitions in the heap is shown by all of the naive fair policies in that they take $O(live\,count \times partition\,count)$ invocations to reach completeness.

Random performed poorly as expected due to the large fraction of garbage in the system and the need to relabel only live objects. However, it still performed better than Reverse FIFO. The approximately 25x increase in page faults per PGC

invocation for Reverse FIFO and Random compared to Round Robin and FIFO is due to remset cache effectiveness, i.e. the ability of the system to keep relevant remsets in memory and not thrash them to/from disc. This shows that the remset caches are more effective with certain access patterns that are determined by the partition selection policy.

The results for UpdatedPointer show that this policy selects partitions that are disjoint with those that promote progress of the completeness algorithm: increasing the ratio at which it is mixed with FIFO reduces the performance of the collector and without such mixing, the collector is not complete. This result confirms the analysis of Section 5.2.8, namely that train-centric and acyclic-garbage-centric policies must be mixed if the benefits of both are to be realised since their results are disjoint. Work spent reclaiming one type of garbage does not typically represent progress in reclamation of the other type.

Finally, the new train-centric policy spends time only where there is progress available, i.e. relabelling live objects from the candidate train; the complexity in that case is the expected $O(live\_count)$.

### 5.3.5 Experiment: Partition Selection with FEA

This section repeats the experiment above with an FEA mesh uniformly distributed across four sites. The mesh contains 24x5 squares, each of 101x101 vertices for a total mutator-visible space containing 13.4M objects in 5300 of 64kB cars. The entire data structure is garbage except for a small number (approximately 5) of objects that are reachable. The garbage structure is spread across 21 trains and is all reachable from a small garbage structure in the 22nd train, which also contains live objects. The progress required to reach completeness is to relabel the live objects from train 22 into the active train and to collapse the 293MB of garbage into train 22.

The collector was attempted to run to completeness but only the train-centric and Reverse-FIFO policies succeeded in the 50 hours allocated on `hydra`. Because the other policies do not ensure relabelling progress for every PGC invocation, they failed to reach completeness within the allocated time and it is not known how long they would take to reach completeness. The fact that they take an unknown time in excess of two days instead of approximately 40 minutes to reach completion with this small data structure indicates that their performance is so poor as to not be worth considering further.

Table 16 summarises the costs of reclamation in terms of the number of PGC invocations and the number of 64kB pages read from disc; DNC denotes that collection did not complete with certain partition selection policies.

| Partition Selection Policy | PGC Invocations | Page Faults |
|---|---|---|
| Round Robin | DNC | DNC |
| FIFO | DNC | DNC |
| Reverse FIFO | 62924 | 46821 |
| Random | DNC | DNC |
| Updated Pointer, 10:1 | DNC | DNC |
| Updated Pointer, 100:1 | DNC | DNC |
| Train Centric | 9382 | 21425 |

Table 16: Reclamation Cost to Completeness; Single Site, 24x5 FEA

The result of this experiment is as expected: the train-centric policy guarantees progress and therefore reclaims large structures of garbage in a timely fashion, while the naive policies do not. The lack of progress in most PGC invocations is confirmed in Section 5.3.7.

### 5.3.6 Experiment: Complexity to Completeness with FEA

The aim of this experiment is to test the predictions of complexity made earlier in this chapter with respect to the train-centric partition selection policy, namely that collapsing a garbage component into a single train for reclamation requires between $O(object\,count)$ and $O(object\,count \times train\,count)$ relabelling events depending on the order in which relabelling occurs. The test load is a range of FEA meshes, each with 24 columns of 100x100 squares of triangles and row counts varying from 10 through 27. Each row of 24 squares contains approximately 2.7M objects consuming 58MB of space.

The result of this experiment is shown in Figure 15; the complexity predictions are confirmed insofar as with a constant number of objects per train, the relationship between total component size and PGC invocation count is approximately linear. Figure 16 shows that the relationship between component size and the number of page faults (sum of 64kB page reads and writes) is not linear, therefore as the load size increases, the cost of each PGC invocation increases. The domain of each result graph is the number of rows in the mesh used as a load on the GC, each row containing 24 squares. These results are from a single invocation of the collector so the variance is unknown.

The non-linear nature of Figure 16 is believed to be because various caches in the system, e.g. remembered sets and pages, have fixed size across these runs and therefore the probability of cache hits falls with increasing load sizes. Figure 17 shows the relationship between the load size in mesh rows and the number of times a remembered set was purged from the cache. The cache is large enough to hold the remsets for 15 rows of data but once the load grows beyond this size, thrashing begins. Figure 18 shows the relationship between the page cache hit

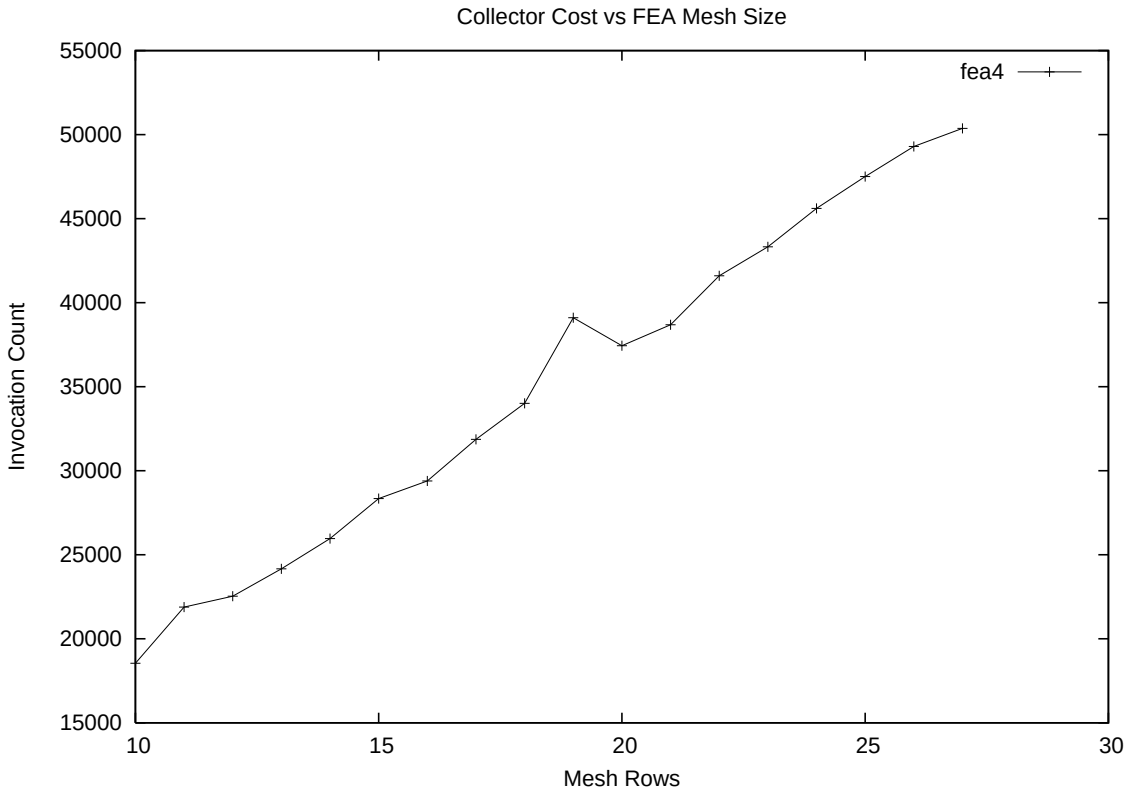


Figure 15: Complexity to Completion, FEA

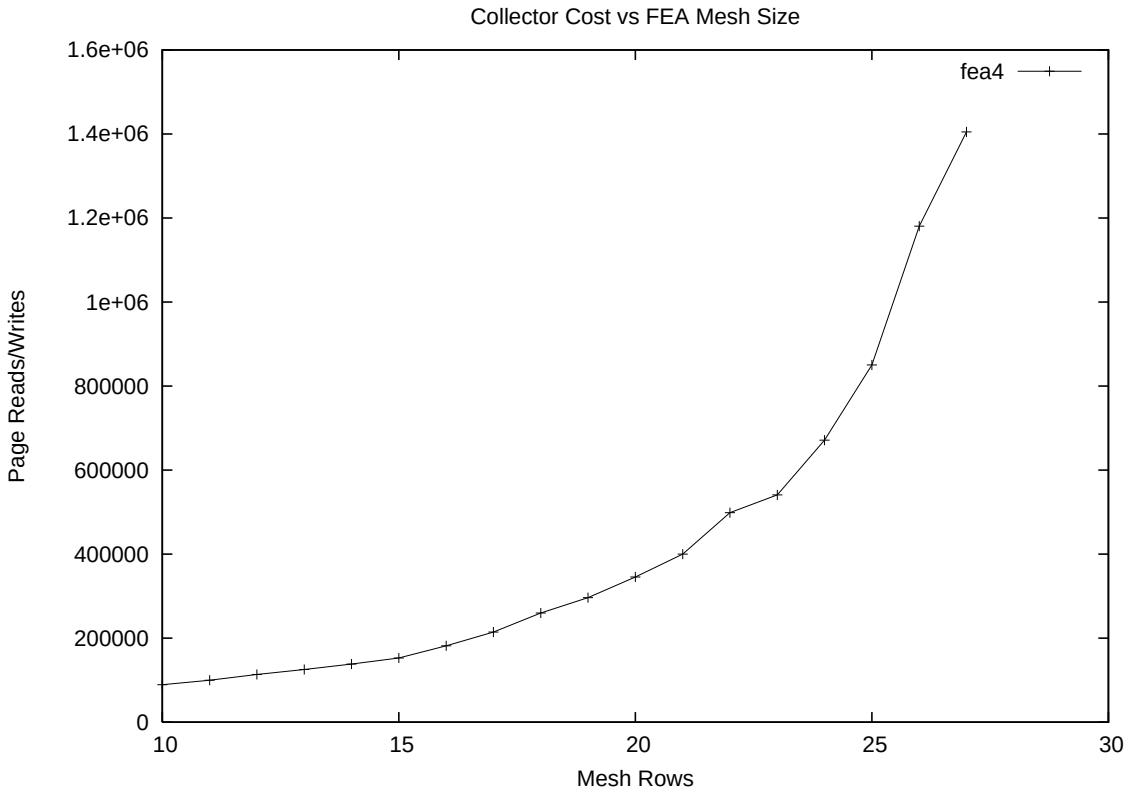


Figure 16: Cost to Completion, FEA

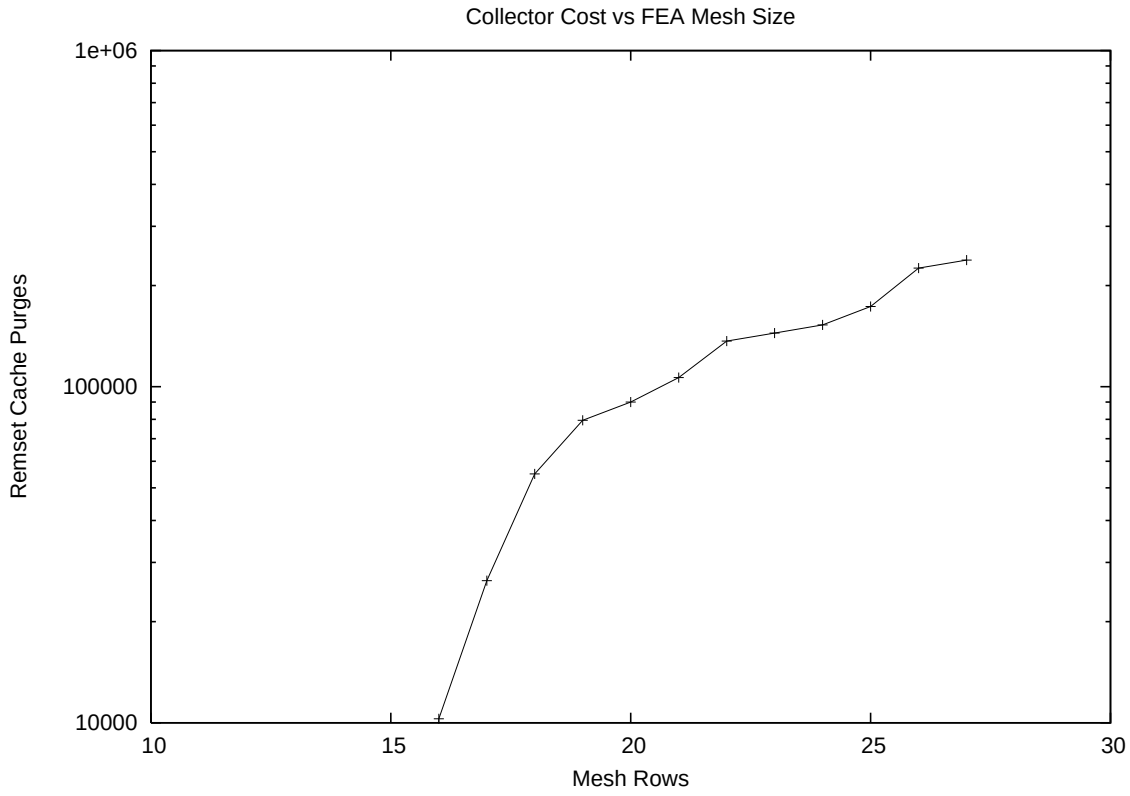


Figure 17: Remembered Set Cache Performance

ratio and load size. With increasing data size and fixed cache size, the hit ratio falls as expected: the more pages in the system, the lower the probability of a given page being in memory. In particular, a downturn in the hit ratio is visible at 16 rows where the remembered set cache begins to thrash.

It should be noted that Figure 15 represents the abstract cost of the collector that was predicted by Surf. The physical cost of the collector (Figures 16, 17 and 18) is a function of this abstract cost and the transformation from abstract cost to physical cost is dependent on the mechanisms implemented within the collector. Referring back to the abstract concept of a layered architecture defined in Figure 12 *with respect only to the results in this section*, the measured implementation may be considered to consist of the following layers:

1. physical subsystems, e.g. disc and network,
2. stable storage subsystem including page cache,
3. remembered set cache,
4. relabelling mechanism (PGC),
5. essence of collector and partition selection policy, and
6. mutator.

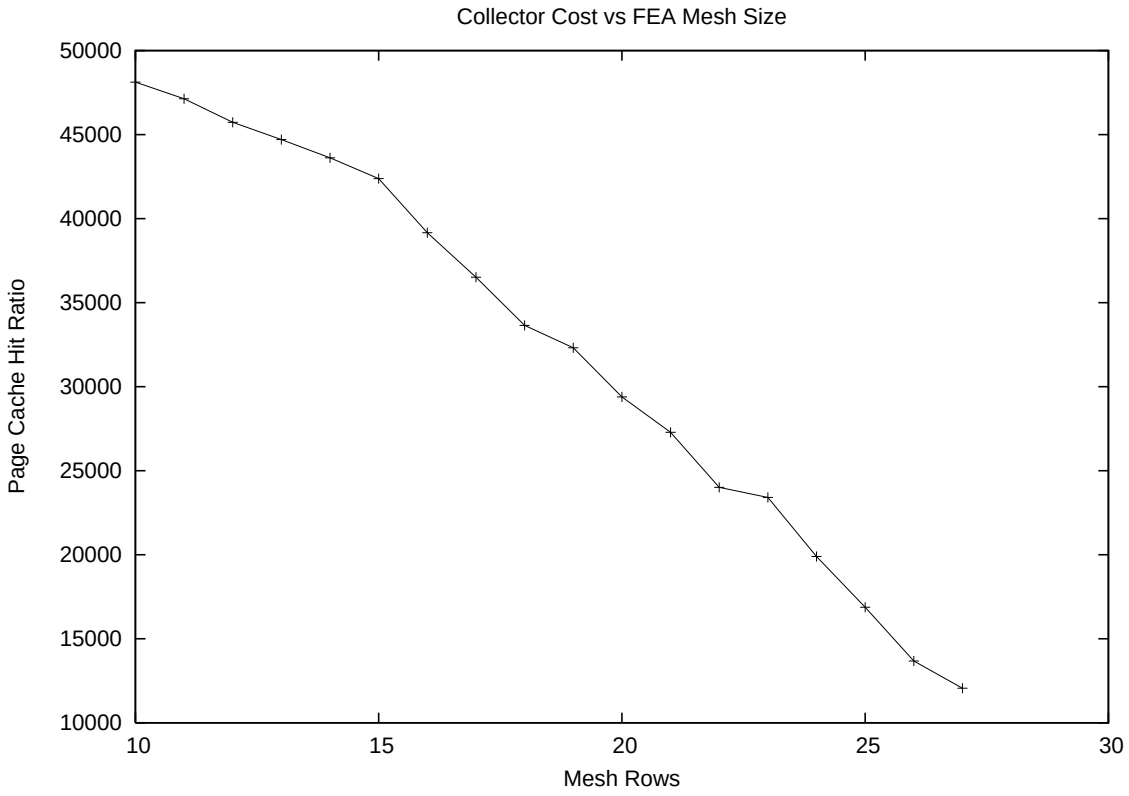


Figure 18: Page Cache Performance

The transformation of load at the uppermost layer (mutator) to costs at the lowest layer (disc IO) occurs as follows:

- The mutator defines the load data structure, which is of a particular size and topology.
- The essence of the collector (the Train Algorithm) as defined by Surf interacts with the partition selection policy to define a transformation between load size/topology and a PGC invocation count.
- The PGC performs relabelling during its execution, transforming its load into a cost in terms of stable read/writes and remembered set updates.
- The remembered set cache is backed by stable storage, so where the required remembered sets do not fit in the cache, stable reads and writes are performed.
- The stable storage layer transforms its load into disc IO as a function of the page cache size and retention policies.

Therefore it should be clear from these experiments that the abstract behaviour of a garbage collector as predicted by Surf is an important factor in the performance of an implementation because it defines the load on the lower layers

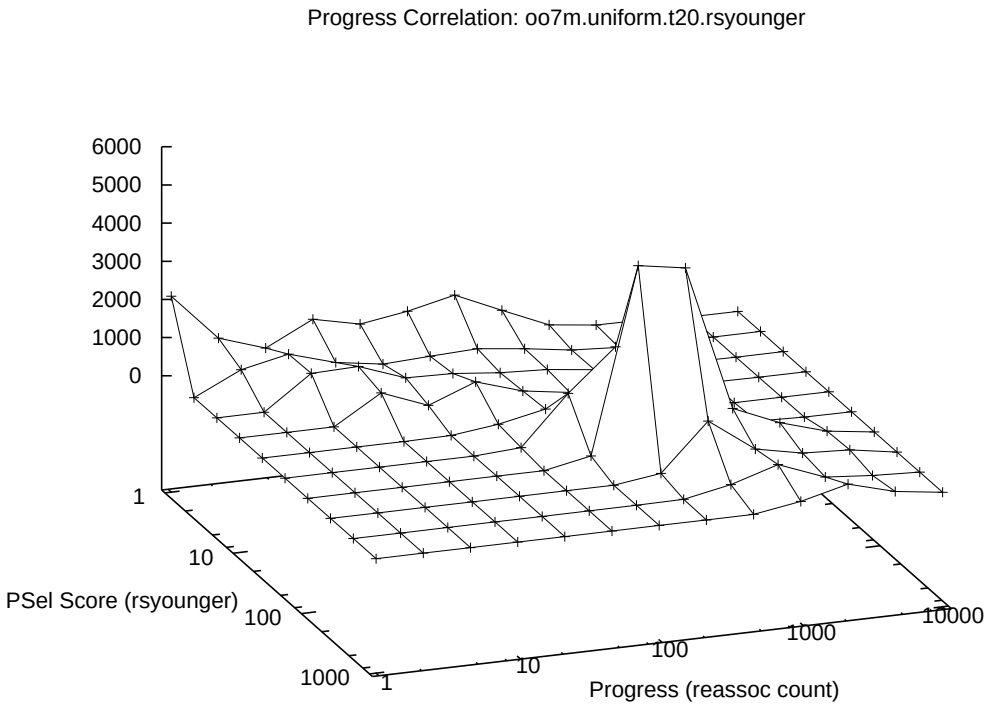


Figure 19: Accuracy of Progress Prediction, OO7

and in particular Surf can be used to analyse the complexity of the collector. However, the lower layers of mechanism have an important impact on real-world performance; for example, relatively simple matters such as cache sizing can make orders of magnitude difference to the observed performance.

### 5.3.7 Experiment: Progress Estimation

A fundamental assumption in the use of the train-centric partition selection policy is that the number of inter-region pointers to a partition is a good estimate of the total number of objects to be relabelled out of that partition were it to be selected for a PGC invocation. The quality of the partition selection policy therefore depends on its accuracy in predicting progress; the fourth and final experiment presented in this thesis is an analysis of the train-centric policy's progress prediction accuracy.

The collector was run on OO7 spread across 20 trains and four sites; for each invocation of the PGC, the system logged both the progress prediction (size of remset from younger trains) and the actual progress achieved (number of objects relabelled). The result is presented as a two-dimensional histogram in Figure 19.

The primary feature of this graph is a straight ridge, indicating that there is usually a good correlation between the predicted and actual progress. The peak

in this ridge shows that the most common case is that approximately 800-1000 objects are relabelled per PGC invocation. There is one interesting feature in the graph, the small ridge across the back edge, i.e. where the prediction is for very little progress yet significant progress was made; this feature indicates the presence of a large connected component within a partition with only a single inter-partition pointer to that component.

Overall, the strong correlation present in this result shows that the train-centric policy is a good predictor of available progress with the OO7 data structure and is therefore a high quality partition selection policy.

The same correlation between predicted and actual progress was investigated with respect to the FEA data structure and the result is shown in Figure 20, which is also a two-dimensional histogram. As per the result with OO7, the primary feature of the graph is a straight diagonal ridge that shows the progress achieved correlates well with the progress predicted.

The large spike in the correlation indicates the common-case performance, i.e. that for most PGC invocations, approximately 3000 objects were relabelled. 3000 objects is approximately the size of a partition, therefore these results that whole partitions are being relabelled, an indication that their contents are typically strongly connected. This behaviour is expected because the load data structure is a mesh, any contiguous subset of which forms a strongly connected component.

There is a small subsidiary ridge extending back from the spike; this ridge indicates the presence of partitions containing 3000-object connected components reachable via a smaller remembered set. This additional ridge is an artefact of the FEA data structure used and occurs because partitions containing corners of the mesh have fewer incoming pointers than partitions in the centre of the mesh.

In comparing the peaks of Figures 19 and 20, the greater average-case progress made in the FEA test indicates a greater degree of intra-partition connectivity with the FEA mesh than OO7.

This test therefore shows that the train-centric partition selection policy produces high quality predictions of the progress available in partitions. The same test is applied to the Reverse-FIFO policy, the only naive policy that reached completeness with the smallest data FEA data structure tested in Section 5.3.5 and the result is shown in Figure 21. Because reverse-FIFO makes no predictions regarding progress, the histogram has only a single dimension, that being the number of objects relabelled in a given invocation, i.e. the progress achieved. Figure 21 shows that in the common case, no progress is made, hence the poorer performance than that observed from the train-centric policy.

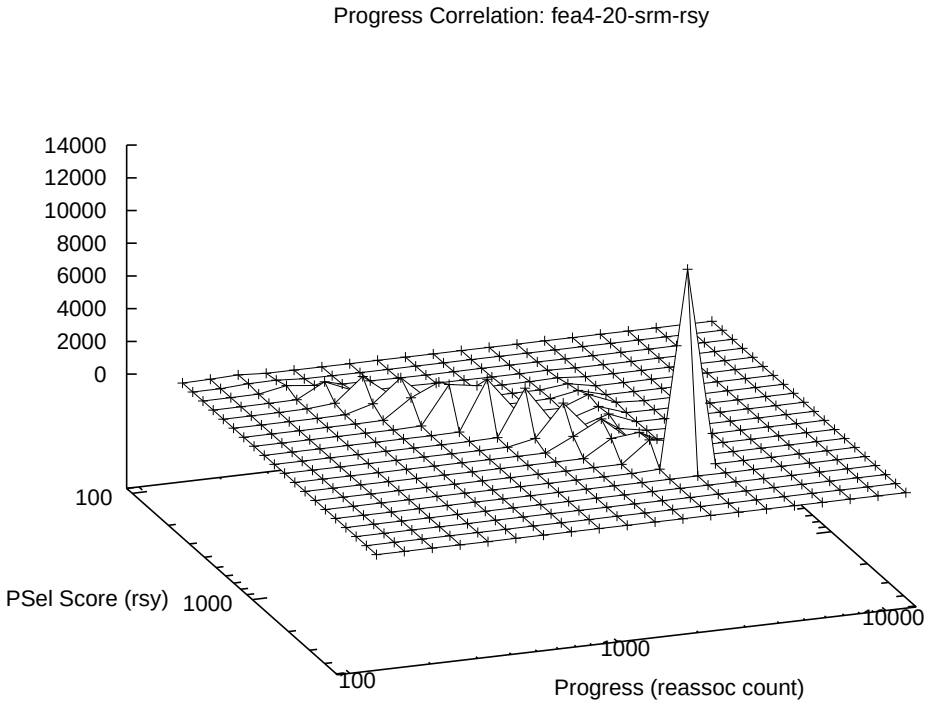


Figure 20: Accuracy of Progress Prediction, FEA

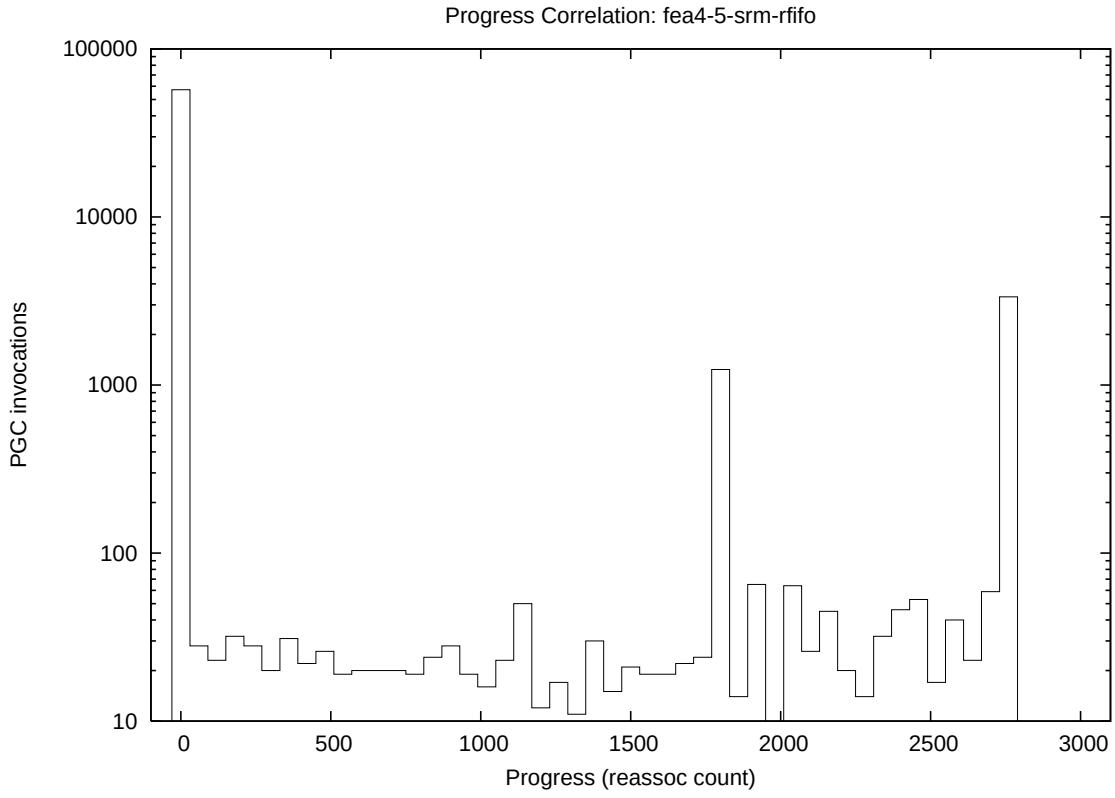


Figure 21: Progress Histogram, Reverse FIFO

## 5.4 Conclusion

This chapter has two purposes, presented in two stages: it shows the utility of the Surf model in designing and analysing a new garbage collector, and it uses experimentation on an implementation to verify the accuracy of the analysis performed in the first stage.

The Surf model is shown to have utility in multiple stages of the design of a new garbage collector, DPMOS. In this chapter:

- the train algorithm is instantiated from the Surf model, resulting in a formal description of a correct garbage collector,
- the model is extended to correctly and efficiently track cached pointers that are held by mutators outside the Surf system model, making the collector suitable for use in a persistent system, and
- the model is used to analyse the means by which DPMOS makes progress and thereby design a partition selection policy that provides linear complexity instead of the quadratic complexity previously seen with naive partition selection policies.

The above demonstrates how the Surf model aids the design and analysis of new garbage collectors and also how the formalism of Surf may be extended for use in systems that do not conform to the Surf system model. Because Surf provides a formal definition of correctness and a simple but formal description of how this correctness is achieved (the use of distributed termination detection), it is possible to prove the correctness of an extension to the model by showing how the extension satisfies the requirements for correct DTD operation.

Because the analysis of progress presented in this chapter is based on the Surf model rather than the specifics of this collector, the analysis is portable to other collectors instantiated from the Surf model.

To verify the validity of the design process and confirm the Surf model's predictions, the newly designed collector is implemented and the predictions tested empirically:

- the implementation appears to be safe and complete, including the cached-pointer tracking extension to the model,
- the prediction of complexity to completeness with a train-centric partition selection policy is confirmed, and
- the train-centric partition selection policy's estimation of progress available is analysed and found to be accurate.

The experimentation therefore validates the analysis performed and predictions made by the Surf model. The net result is a demonstration that the Surf model is an aid to the construction and analysis of new garbage collectors and that analysis derived from the model has been confirmed experimentally via an implementation of the Train Algorithm.

# Chapter 6

# The Tram Algorithm

Having presented the Surf abstract model of garbage collection in Chapter 3 and demonstrated its descriptive and analytical power in Chapter 4, this chapter uses the Surf model to instantiate an entirely new distributed garbage collection algorithm: the Tram Algorithm[1]. This chapter exists to demonstrate the utility of the model in designing entirely new collectors by exploring the design space provided by the model and as a result of this exploration, an entirely new class of garbage collection algorithm is discovered.

The new point in the Surf design space occupied by the Tram Algorithm is relabel-source with many (more than two) regions; this chapter proceeds by presenting a design for the Tram Algorithm by instantiating the Surf model given the new design point and then defining behaviour so that the safety and completeness requirements of the model (see Section 3.5) are satisfied. Having defined how the collector makes progress and identifies garbage, the Surf model is used to make predictions as to the likely behaviour of this new collector. Finally, the Surf model is used to analyse the expected relative performance of the Tram Algorithm with respect to existing collectors such as Distributed Trains (Section 4.2) and Back Tracing (Section 4.4).

In exploring the expected behaviour of Trams, it is predicted that the algorithm is capable of **discovering topological information** within the live graph that is an approximation of strongly connected components [92] (SCC); the analysis shows how this capability is unique to relabel-source, many-region collectors like the Tram Algorithm. Topology in the form of strongly connected components is valuable information for the purposes of garbage collection: if all such components are known as Surf regions then all garbage within the system may be detected without further relabelling work, resulting in excellent timeliness. The drawback to discovering SCCs is that the time-complexity of

[1]The name of this collector is so chosen because it takes features from both Back Tracing (Section 4.4) and the Train Algorithm (Section 4.2), i.e. this collector is constructed using trains that go backwards.

doing so is great and the object graph may concurrently mutate, rendering the topological information out-of-date.

The implementation of the Tram Algorithm and the verification of the predictions regarding the discovery of topology made in this chapter are future work. The purpose of this chapter is not to investigate an implementation but rather to show how the Surf abstract model may be instantiated to form an entirely new garbage collection algorithm and how the Surf's model of progress and termination may be applied to the analysis of garbage collector behaviour and performance in the absence of an implementation.

## 6.1 Design of the Tram Algorithm

The Tram Algorithm is defined by following the instantiation process of Section 3.5. The process requires that the collector be specified as occupying a particular point in the Surf model's design space and that a list of requirements be fulfilled so that the safety and completeness proof from the model may apply. The table below therefore lists the properties of the collector in dimensions of the model's design space and summarises the correctness requirements that apply and how they are fulfilled.

| **Model Concept** | **Mapping to Trams** |
|---|---|
| Direction | Progress is by relabel-source. |
| Regions | Regions are trams and are identified by integer names<br>Regions are ordered, preventing cyclic relabelling. |
| Relabelling job | Relabelling is from older regions to younger regions.<br>Any object in an older region that contains a pointer to a younger region constitutes work for the relabelling job of the younger region. |
| Isolation DTDA | Isolation of regions is detected using Task Balancing [50, 82].<br>The isolation of every candidate region is represented by a job.<br>Every inter-region remset entry constitutes a task of a region isolation job.<br>Termination of a job implies that a region is usefully dead. |
| Remset DTDA | Task Balancing as a pointer tracking algorithm implements remembered sets. |

| Model Concept | Mapping to Trams |
|---|---|
| Safety & Completeness Requirements | The system is subject to the victimised relative problem, therefore synchronisation between relabelling and the region isolation DTDA is required.<br>Relabelling liveness in the face of mutator activity is required and provided by snapshot at the beginning.<br>At the beginning of operation, there are two regions (non-candidate and one candidate) but in the general case there will be more than two regions.<br>A suspicion algorithm that provides the Strong Suspicion Guarantee is to be used.<br>No new regions are created until there is no relabelling work for any existing region, therefore region growth is unbounded and dead regions will become usefully dead. |

The details of this mapping are discussed in further detail below. For each parameter of the algorithm that is defined by the model, predictions as to the collector's behaviour are made by applying the general-case analysis of Section 3.5 to the specific case of this algorithm. Further analysis is then required in terms of support protocols that are not defined by the Surf model, e.g. a protocol to decide when a region may be created and how the suspicion algorithm operates.

### 6.1.1 Direction of Progress

Because progress is made by relabel-source, i.e. back-tracing, the collector begins from a hypothesis (suspicion of a particular object) then evaluates the hypothesis by determining if that object is garbage or not. Given the possibility that the hypothesis may be wrong, there is no guarantee that the collector will make progress at any particular time; progress is entirely dependent on the suspicion algorithm in use. A suspicion algorithm that always produces false hypotheses will result in no garbage being reclaimed.

Where suspicion produces a correct hypothesis, i.e. the suspected object is dead, the collector will reclaim a component of garbage in $O(dead\,count)$ relabelling steps.

### 6.1.2 Definition of Regions

The Tram Algorithm contains many regions, i.e. more than two regions and each region is referred to as a **tram**. Since there are more than two regions, no synchronisation with a region isolation DTDA is required at region creation because in no case will a non-candidate region become a candidate.

A multiplicity of regions also means that there is no reason for any particular region to have very large scope, e.g. all live objects. Smaller regions where their scope is related directly to graph connectivity means that there exists the opportunity to introduce a degree of fault tolerance into the collector, though that is not considered here in detail. In other words, a region may be reclaimed by considering only the contents of that region and its immediately neighbouring graph nodes; the collector does not require interaction with the entire system to reclaim a small isolated region; the collector therefore is expected to exhibit some scalability.

There is a total ordering on regions; each is identified by a unique integer and the ordering of integers defines the region ordering. Regions are created in increasing order, starting at one. Older regions have lower numbers, younger regions have higher numbers.

The root object exists alone in a special infinitely young non-candidate region for which no relabelling is performed.

### 6.1.3 Relabelling Jobs

Every region has a relabelling job; every inter-region pointer constitutes a job of the region that it points to. Therefore, termination of a job for a region that does not contain the root implies that that region is usefully dead.

Relabelling work is defined to exist for every inter-region pointer from an older region to a younger region, the processing of this work implies that the source of the pointer is relabelled into the younger region. The available relabelling work for a region is therefore a non-strict subset of the set of tasks for a region and therefore there will exist states wherein the relabelling job for a region has not terminated yet there is no work available for that job. Once the system enters such a stable state, the region configuration is stable and will remain unchanged until the pointer in question is erased or a younger region subsumes the regions on both sides of the pointer that constitutes a job.

The ability to form stable region configurations derives directly from the way in which Trams instantiates the Surf model and it is necessary for the collector to be able to represent object graph topology as described in a later section.

### 6.1.4 Region Isolation DTDA

There exists a DTDA job for every region; the mapping is such that termination of the job implies isolation of the corresponding region. Each object in the region that is the target of an inter-region pointer is a task of the isolation job and these tasks are themselves jobs of the lower-level DTDA (remembered sets); see Sections 3.3.4 and 3.3.5 for an explanation of how the two levels of DTDA (object trivial unreachability and region isolation) interact.

The synchronisation requirement for victimised relative drives the choice of Task Balancing as the Isolation DTDA; being derived from the Doomsday protocol [63], the synchronisation scope is less than that required for a wave-based DTDA.

### 6.1.5 Remembered Sets

Each object has a DTDA job representing its trivial reachability. Each pointer to that object is a task of the job; when no such pointers remain then the job terminates and the object is trivially unreachable. See Section 3.3.4 for a detailed description of how pointer tracking maps to the DTD model.

### 6.1.6 Safety Requirements

Because the Tram algorithm contains many candidate regions and performs relabelling amongst those regions, it is subject to the victimised relative problem; this occurs because objects may be relabelled out of a candidate region. Safety in the Tram Algorithm therefore requires synchronisation between the relabelling processes and region isolation; the approach taken is that of Norcross [82] in requesting a witness.

Under the Doomsday system model, spontaneous task creation may occur only at the home site and tasks may be created on other sites only if there is an existing task to witness the task creation. Therefore, when a remote site ($R$) wishes to relabel an object that would imply unsafe task creation, it sends a request for witness ($wreq$) to the relevant region's ($\mathfrak{M}$) home site ($H_{\mathfrak{M}}$). A witness task is created at the home site, migrates back to $R$, witnesses the task creation due to relabelling then itself dies. This process, illustrated in Figure 22, satisfies the cover rule stated in the Doomsday system model, i.e. that task creation at a site other than the home site must be witnessed by an existing task. Each vertical line in the graph represents the execution process at a single site, circles are events and dotted lines represent messages.

The situation leading to the events Figure 22 is that there is work for $\mathfrak{L}$:
$\exists y.x \wedge x \in \mathfrak{L} \wedge y \in \mathfrak{M} \Rightarrow work(\mathfrak{L})$

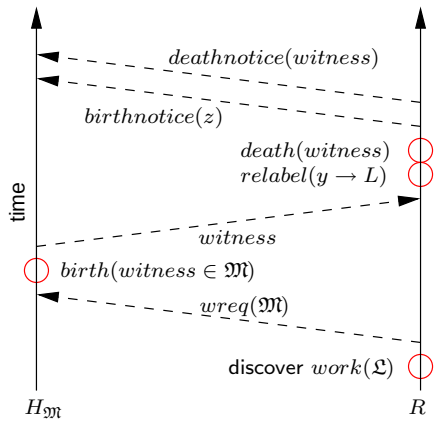


Figure 22: Witness Request Protocol

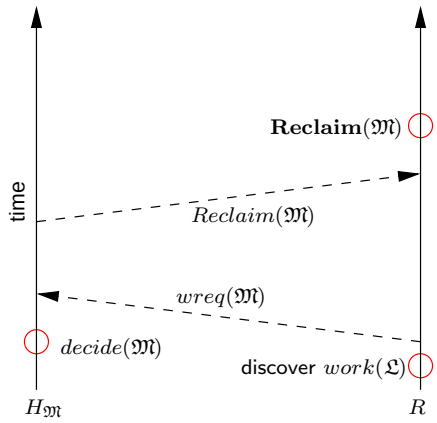


Figure 23: Witness Request Denied

Processing this work implies relabelling $y$ but if $y$ contains intra-region pointers then additional tasks must be created for the isolation job in $\mathfrak{M}$ because additional objects within $\mathfrak{M}$ will become the targets of inter-region pointers:

$relabel(y \rightarrow L)_R \wedge \exists y.z \wedge z \in \mathfrak{M} \Rightarrow birth(T^z \in \mathfrak{M})_R$

If there exist tasks in $\mathfrak{M}$ at $R$ already, then this task creation is safe. However, there is no guarantee that such tasks exist and if none do, then a request for witness is required and the relabelling must be deferred.

The race condition that this synchronisation protocol prevents is the case where a region is decided (at its home site) to be isolated while some other site is concurrently relabelling reachable objects into the region. Should the home site decide that the region is isolated, it will deny the request for witness from the remote site and the intended relabelling action at the remote site will not be permitted. The object that would have participated in an unsafe relabelling is now garbage and will be reclaimed once the remote site is notified that the region is isolated, thereby obviating the need for the relabelling event. Figure 23 illustrates the case where a request for witness is denied.

Because $\mathfrak{M}$ has been reclaimed, the work for $\mathfrak{L}$ that was previously under

consideration no longer exists and the object that would have been relabelled no longer exists. Though completeness requires that some work due to pointers that are erased by the mutator is preserved, work due to pointers erased by the reclamation process is never preserved.

### 6.1.7 Completeness Requirements

Completeness in the face of mutator activity requires that something similar to snapshot-at-the-beginning is used; though a less conservative approach is possible, it is easier to prove correctness with SATB. Therefore, the erasure of an inter-region pointer does not imply that that pointer no longer represents relabelling work, a mechanism must be provided so that the work is processed as if the pointer were not erased. Without SATB or some other means of preserving relabelling work, it would be possible for the mutator to hide relabelling work and prevent relabelling progress. Safety is not compromised because the Surf model proves that mutator activity conforms to the DTD model constraints, i.e. remembered sets are correct.

Completeness also requires that all dead objects are eventually members of usefully dead regions; the Surf model breaks this down into two more specific requirements: that all dead objects are eventually members of dead regions and that dead regions be permitted to grow into usefully dead regions. The first requirement is fufilled by a suspicion algorithm that satisfies the strong suspicion guarantee, i.e. that all objects are eventually suspected. The second requirement (unbounded growth) implies that no new regions should be created until no further growth is possible on any existing region; the improper creation of new regions could stymie the growth of existing dead regions and prevent them becoming usefully dead. Deciding when a new region may be created requires global knowledge and the means by which this is efficiently obtained is described in the following section.

Finally, completeness requires a suspicion algorithm that provides the suspicion guarantee, i.e. that the collector will eventually suspect every dead object or some other dead object reachable therefrom.

### 6.1.8 Suspicion

The Tram Algorithm requires a suspicion algorithm to generate hypotheses, i.e. starting points of new regions. The suspicion algorithm is not part of the mapping but it is critical for correctness and performance. The primary requirement is that the algorithm satisfy the suspicion guarantee (Section 3.4.2.3), i.e. that all

dead objects will eventually be in a candidate region. Trams provides the strong suspicion guarantee, i.e. that every object is eventually suspected.

Performance requirements are more difficult to define because work spent relabelling the live graph is not necessarily wasted: it detects no garbage immediately, but it may result in the discovery of topology.

An oracular suspicion algorithm for the immediate detection of garbage would select the most downstream (in terms of pointer direction) object in a garbage component as soon as it becomes garbage, resulting in a successful trace and the detection of garbage in $O(n)$ time from the point of suspicion. However, the Tram Algorithm may discover useful topology in a trace from a live object so choice of a suspicion algorithm promoting high performance with Trams involves some tradeoffs between the immediate detection of garbage and the detection of topology. The outcome of any particular suspicion algorithm will be a mixture of the three following options:

- performing no relabelling work before a component becomes garbage, then suspecting it sometime thereafter will result in low computational overhead,
- performing significant (high overhead) and useful (discovers topology) relabelling work on the live graph can result in excellent detection timeliness, and
- performing significant but useless (discovers no topology) relabelling work on the live graph is a waste of time.

A simplified view of the tradeoff is that more time spent analysing the live graph should result in an improvement in timeliness: it is a time/space tradeoff where lower average performance (assuming the garbage collector shares CPU time with the mutator) buys a reduced garbage load on the system and therefore the ability to store more live objects. Analysis of the live graph will never result in lower net time overhead than applying the oracular suspicion algorithm for immediate reclamation of garbage since the number of relabelling events (objects traced) will always be greater than if only the dead regions are traced.

Making use of the time-space tradeoff requires that the regions formed approximate strongly connected components (SCCs); the process by which this occurs is examined in detail in Section 6.2.

Assuming that we wish to make use of the time/space tradeoff and implement a suspicion heuristic that will improve detection timeliness, the task is to design a suspicion ordering that matches that stated in Section 6.2.1, i.e. SCC $\mathfrak{A}$ should be suspected after $\mathfrak{B}$ if $\mathfrak{B}$ is reachable from $\mathfrak{A}$. Clearly however, the

suspicion heuristic does not know the graph topology so it must make guesses at it using metadata computed during previous traces and perhaps some additional information. The heuristic presented here uses a distance-from-root heuristic in conjunction with information from the Tram Algorithm as to how often a particular object has been relabelled.

Inspection of an ideal suspicion ordering results in the following conclusions regarding the ordering:

- the suspicion order approximately matches the ordering of objects by their distance (in pointers) from the root,
- having suspected one object in a strongly connected component, no other objects in that component are suspected, and
- a **directory** object (from which multiple SCCs are reachable) changes label multiple times and should be traced after all the objects reachable from it.

These three observations are combined to form a heuristic to select objects for suspicion in an ordering that approximates the ideal ordering derived previously. The suspicion algorithm is expressed formally in Algorithm 1 and additional functionality required during relabelling to support suspicion is expressed in Algorithm 2.

The distance estimation may be computed via an occasional forward trace of the graph; it is likely that in an implementation the distance estimates would be computed incrementally, piggy-backed on a relabel-target partition collector that the Tram Algorithm would be implemented as a compound with for performance reasons in the face of acyclic garbage.

A small example object graph is provided in Figure 24; for one phase of operation (i.e. the period of time over which the strong suspicion guarantee is applied), the suspicion order selected by this heuristic is $g$, $b$, $d$, $e$, $h$, $y$, $z$, $r$ and the labelling resulting from this execution is shown in the same figure. It should be noted that the labelling is optimal for this graph though the resulting labelling is not guaranteed to be optimal, i.e. this is not a new algorithm for determining strongly connected components.

It appears (though it is unproven and untested) that the suspicion heuristic presented here matches the optimal ordering derived previously where the object graph is a tree of SCCs; the more general case of a DAG of SCCs that contains an SCC reachable from multiple SCCs — a diamond in the graph — can cause both branches of the diamond to end up in the same region. The heuristic is therefore not optimal but it may provide a good approximation if the distance heuristic is accurate.

**Algorithm 1** Suspicion Heuristic

```
// object ID
typedef int oid;
// per-object metadata
typedef struct {
    int relabel_count;    // relabellings in this phase
    distance_t distance;  // distance from root
} metadata_t;
// retrieve object metadata given ID
metadata_t& get(oid);

// all objects in the heap
list<oid> objects;
// objects not yet touched in this phase
priority_queue<distance_t, oid> untouched;
// objects touched twice in this phase
priority_queue<distance_t, oid> directories;

// decide which object to suspect next
oid suspect()
{
  // beginning of phase
  if(untouched.empty() && directories.empty()){
    for(id in objects){
      metadata_t& md=get(id);
      md.relabel_count=0;
      untouched.add(- md.distance, id);
    }
  }

  // suspect furthest untouched
  if(!untouched.empty()){
    return untouched.first();
  }

  // none untouched, suspect furthest directory
  if(!directories.empty()){
    oid id=directories.first();
    directories.remove(id);
    return id;
  }

  // graph appears to be empty
  throw suspicion_failure;
}
```

**Algorithm 2** Relabelling Work for Suspicion

```
// action to perform when relabelling an object
// maintains metadata required for suspicion
void on_relabel(oid id)
{
  untouched.remove(id);
  metadata_t& md=get(id);
  if(++md.relabel_count == 2){
    directories.add(- md.distance, id);
  }
}
```

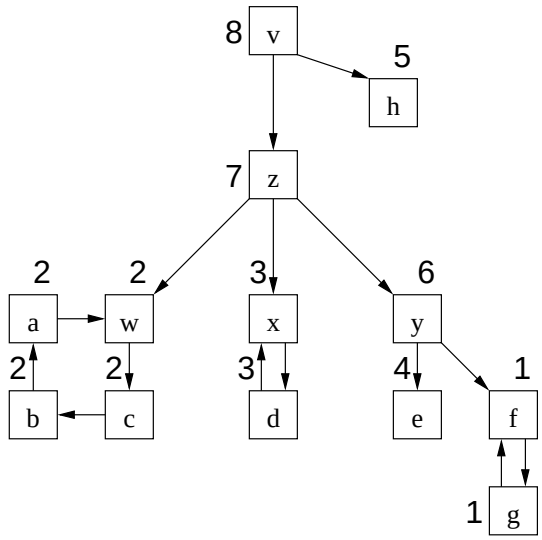


Figure 24: Labelling from Heuristic

The presented suspicion heuristic represents one extreme of the time/space tradeoff by spending significant quantities of time — $O(n^2)$ — to arrive at an approximation of the optimal labelling and therefore a high probability of detecting garbage without further relabelling if the tracing algorithm has had sufficient time to inspect the live graph. It should be noted that mutator activity and distance estimation inaccuracy will reduce the accuracy of the labelling produced by this suspicion heuristic but that such effects are not analysed here. Frequent mutations to the graph will cause the topology discovered to be out of date.

### 6.1.9 Support Protocols

Completeness requires that regions be created only when no further relabelling work is available. The Surf model in Section 3.5 states that this may be detected

by a piggy-back on the relabelling termination DTDA. By considering only inter-region pointers from older regions to be tasks, a new region-creation DTDA job is defined that is a subset of the relabelling termination job. When this smaller job terminates, there exists no relabelling work for a region and a new region may be created.

Because the region-creation DTDA considers only a subset of tasks, it is not safe with respect to mutator activity, i.e. the mutator could create a pointer which represents work. In other words, relabelling work may become available for a region after it has decided that it has no further work. However, this has no effect on correctness because:

- safety is uncompromised because the relabelling-termination jobs are correctly mapped as defined by the Surf model,
- completeness is uncompromised because region growth is required only so that dead regions may become usefully dead, and
- unbounded growth is not required for live regions.

Interaction between this piggy-back DTDA that operates on a subset of tasks and the mutator can result only in the loss of unbounded growth for live regions.

The Tram Algorithm therefore contains three DTDA mappings: one for detecting trivial isolation (remembered sets), one for detecting region isolation via termination of relabelling and one that controls region creation.

Because regions are distributed, Surf states that a mechanism to control region membership is required. Because region growth is by relabelling therefore diffusing, membership may be tracked at a region's home site by inspecting the state of the relabelling-termination job.

### 6.1.10 Extensions to the Tram Algorithm

This abstract description of the Tram Algorithm is for the fundamental form, i.e. an instantiation that is not compounded with any other collector. For performance reasons, it would be prudent for an implementor of Trams to construct a compound collector with trams and some means of rapidly detecting acyclic garbage, e.g. a partition collector. The design of a partitioned Tram Algorithm is not considered here in detail because it has no relevance to the Surf model or the ability of the Tram algorithm to detect topology.

## 6.2 Discovering Topology

This section uses the Surf model to analyse how the Tram Algorithm makes progress and therefore how topology may be discovered and represented by the regions formed by the collector. The aim in discovering topology is to form regions that are congruent with strongly connected components (SCCs), i.e. graph components wherein every object in the SCC is reachable from every other object in the component:

$\mathbf{SCC}(\mathfrak{C}) \equiv (x \in \mathfrak{C} \wedge y \in \mathfrak{C} \Rightarrow \mathbf{Reachable}(x, y))$

The nature of a SCC is that liveness is uniform across the component, i.e. it is entirely live or dead:

$\mathbf{SCC}(\mathfrak{C}) \Rightarrow (x \in \mathfrak{C} \wedge y \in \mathfrak{C} \Rightarrow (\mathbf{Live}(x) = \mathbf{Live}(y)))$

Therefore an SCC becomes garbage atomically and makes a useful smallest unit in the detection of garbage. When the last pointer to an SCC is erased, the whole SCC becomes garbage instantaneously. If there were a Surf region representing an SCC, it would immediately (modulo DTDA latency) detect the isolation of that SCC. A region smaller than an SCC will detect less garbage than is possible while a region larger than an SCC may not detect that the SCC has become garbage because the region may contain other reachable objects. Where regions match SCCs, collection timeliness will be excellent because no relabelling work is required between an SCC becoming garbage and its being detected as such.

Representing SCCs using regions requires a collector where regions are capable of assuming a stable configuration, i.e. the collector will not merge regions merely because there is a connection between them. By the definition of the relabelling jobs in the Tram Algorithm above, the Tram Algorithm is capable of forming stable region configurations because it will not relabel objects from younger regions into older regions. The aim of this section is to analyse the way in which regions grow using the Surf model and control this growth so that the regions so formed are a good approximation of strongly connected components.

The control of region growth is to be implemented via the suspicion heuristic, i.e. suspect objects for the creation of new regions are chosen in a carefully designed order so that the desired region configuration is reached. The nature of progress in the Tram Algorithm is such that where an object in an SCC is suspected, the region will grow to encompass the whole SCC. The suspicion heuristic is therefore designed so that where one SCC $\mathfrak{B}$ is reachable from $\mathfrak{A}$, an object in $\mathfrak{A}$ must be suspected later than any object in $\mathfrak{B}$.

Following subsections examine the decomposition of the graph into a directed acyclic graph (DAG) of strongly connected components and show that the Tram Algorithm is unique in its ability to do this. The next section uses that analysis

to derive a suspicion heuristic based on an object's distance from the root and an estimation of which objects constitute bifurcations in the DAG.

### 6.2.1 Region Formation

The Surf model of progress is used here to analyse how regions form and find a stable configuration of regions that correspond to SCCs.

The way in which the completeness requirements of the Surf model are fulfilled by Trams — unbounded region growth without mutator interference — ensures that each region will grow to be at least as large as any strongly connected component it contains, i.e. a region will never contain only a strict subset of an SCC when relabelling reaches idleness, therefore regions are large enough. What remains is to determine a means to ensure that regions are small enough, i.e. contain only a single SCC and nothing more.

To fulfil the second requirement, consider that any directed graph (specifically, the mutator's object graph) may be decomposed as a directed acyclic graph (DAG) of SCCs. Where cycles exist in the graph, each is a single SCC and may be collapsed into a meta-node; the remaining graph is a DAG of these meta-nodes, or more precisely, a DAG of SCCs. The term "meta-node" is used here to represent a graph component; every pointer to an object in the meta-node is a pointer to the meta-node and likewise every outgoing pointer from objects inside the meta-node are pointers from the meta-node.

Consider further the model of relabel-source progress when applied to a DAG; it starts at some point in the graph and proceeds against the grain of pointers, visiting SCCs from which the start-point is reachable exactly once. The aim here is to select a suspicion order that applies a different label to each SCC meta-node and it is possible to do this merely by constraining the order in which suspicion occurs; this is because only pointers from older regions to younger regions constitute relabelling work. Because the structure under consideration is acyclic, it is possible to choose a labelling where no younger region is reachable from an older region, i.e.:

$\mathbf{Reachable}(\mathfrak{A}, \mathfrak{B}) \Rightarrow age(\mathfrak{A}) < age(\mathfrak{B})$

In this situation, there is no relabelling work available so the configuration is stable; if each SCC in the graph is uniquely labelled, it will remain so.

To arrive at this situation, one can constrain the suspicion algorithm so that it suspects SCC $\mathfrak{A}$ after SCC $\mathfrak{B}$ where $\mathfrak{B}$ is reachable from $\mathfrak{A}$, i.e.

$\mathbf{Reachable}(\mathfrak{B}, x) \wedge z \in \mathfrak{A} \wedge \mathbf{Reachable}(\mathfrak{A}, \mathfrak{B}) \wedge (\mathfrak{A} \cap \mathfrak{B} \equiv \varnothing)$

$\Rightarrow suspect(x) \prec suspect(z)$

The first suspicion hypothesis (of $x$) will form a region containing both $\mathfrak{A}$ and $\mathfrak{B}$ and the later hypothesis will change $\mathfrak{B}$ to be in a different, younger region.

The arrangement will be preserved until the suspicion algorithm later suspects another object reachable from $\mathfrak{B}$ or the mutator changes the graph topology. Suspecting SCCs in this manner will ensure that each receives a unique label.

This therefore shows that constraining the order in which the suspicion algorithm suspects objects will cause the regions formed by the Tram Algorithm from a DAG of SCCs to be congruent with the SCCs in the graph, in other words, analysis via the Surf model of progress proves that the Tram Algorithm's definition of relabelling work permits it to discover SCCs in an unchanging graph if the correct suspicion order is chosen.

### 6.2.2 Uniqueness

It would appear that the ability of the Tram Algorithm to discover graph topology while the graph is live is unique. Discovery of topology for the purposes of garbage collection implies that the collector is aware of the lack of connectivity between different areas of the graph so can reclaim an area when the last pointer to it is erased and not require further relabelling progress to discover what is garbage; if the graph topology is known to the collector, it need perform no additional work when a region becomes unreachable because it already possesses all the information required to make the determination of isolation.

Assuming that topology discovery is to be implemented using the region abstraction of the Surf model, two properties are required that are shared by no other collector:

- regions of live objects must have the opportunity to be stable in the long term, and
- regions must be sufficiently fine-grained to contain only the graph area that becomes garbage in one step.

Using the Surf model, it can be shown that these two properties are shared only by a collector that takes the relabel-source approach and has more than two regions, i.e. the Tram Algorithm or variants thereof.

Consider relabel-target and assume that there is an ordering on regions such that objects may be relabelled only from older regions to younger regions, thereby preventing livelock. The continual creation of new regions and placing the root in the youngest (or a virtual infinitely-young) region means that the root will eventually be in a younger region than all other objects; any pointer from a live older region to a younger region will soon be in an even younger region as it is relabelled towards the root. Relabel-target approaches therefore attempt to collapse all live objects into a single region if the region creation rate is low; if

region-creation is rapid, then objects will be strung out across a number of regions according to their distance from the root instead of any useful topology. Relabel-target is therefore incapable of extracting useful topological information from the graph because there is no stable configuration of more than one region within the live portion of the object graph.

Of the relabel-source approaches, the only published algorithms have a single candidate region only. Should a candidate become a non-candidate by absorbing the root, the connectivity information that was discovered is discarded so that tracing may begin again at a new suspect.

An alternate way to express this is that collectors form regions on the basis of hypotheses. Each new region is formed on the basis of a new hypothesis of connectivity; each hypothesis is tested and its implications for connectivity found by diffusing the region across the graph. Relabel-target collectors form new regions at the root, i.e. each hypothesis the axiom of liveness of the root; testing the hypothesis of each new region amounts to discovering all live objects and any object not so discovered must therefore be garbage. Because the same hypothesis is used for every new region in all published relabel-target collectors, the regions formed will tend to be congruent, i.e. the collector at equilibrium will collapse all live objects into a single region and discover no topology.

In contrast, relabel-source collectors are a multiple-hypothesis approach. Each suspected object represents the hypothesis that that object is garbage; the hypothesis is tested by diffusing a region across the graph to determine if the root may reach the object in question. The Tram Algorithm evaluates many different hypotheses, therefore the equilibrium state of the collector is many different regions. The use of Surf to derive a multiple-hypothesis relabel-target collector may result in another collector with the ability to detect topology in the live graph; such a system is future work and examined briefly in Section 7.2.

Therefore the only mapping into Surf that provides topological information about the live region is currently the Tram Algorithm.

### 6.2.3 Complexity of Optimal Labelling

Assuming a perfect suspicion algorithm and no mutator activity, the Tram Algorithm requires $O(n^2)$ relabelling steps ($O(n)$ regions created, each relabelling $O(n)$ objects) to arrive at a region configuration that corresponds to SCCs, therefore the excellent timeliness result is available only after significant computation time has been spent. In contrast, Tarjan's Algorithm [92] is capable of detecting SCCs in linear time but only on uniprocessor systems as it requires global set manipulations during its operation. If further research were to discover

a means to detect SCCs in a distributed graph in linear time, that would be a major breakthrough for distributed garbage collection.

## 6.3 Comparisons with Other Collectors

The Tram Algorithm shares design features with both the Train Algorithm and Thor's Back Tracing. It is a multiple-hypothesis (many region) collector like Trains but it makes progress via relabel-source, like Thor. This section explains the similarities between the previously published collectors and Trams and then uses the Surf model to make coarse predictions of their relative performance based on how each collector instantiates the model.

The performance aspects considered are those defined as desirable properties of a garbage collector in Section 1.1.2: latency, throughput, overhead and timeliness. Lastly, because Trams' ability to detect topology is dependent on mutator activity, the characteristics of an application suitable for use with Trams are investigated.

### 6.3.1 Antecedents

MOS (the train algorithm) makes use of multiple regions, an approach that improves collector performance:

- the presence of more than two regions means that no synchronisation is required in the creation of a new region;
- regions need not be global, thereby adding asynchrony and robustness to the system since the only sites that need participate in the reclamation of garbage are those that it spans.

Back Tracing is the only collection approach yet published that takes the relabel-source (Section 3.4.2) approach. Relabel-source is inherently expensive because it requires the maintenance of metadata describing where each object is reachable from and for a given suspicion hypothesis there is no guarantee that garbage will be reclaimed; for these reasons, the relabel source approach seems not to be a popular implementation choice.

There is, however, a benefit to relabel-source in that it is $O(dead\,count)$ when a dead object is correctly suspected: with use of an accurate suspicion algorithm, the amount of time spent tracing the graph can be much smaller than that of forward-tracing collectors like distributed marking (Section 4.1) in the face of a large live region.

### 6.3.2 Latency

Latency refers to the pause-times introduced by the collector. Since the collector is concurrent with both itself and the mutator, the model predicts that it should introduce no noticeable pause times.

The only interaction required with the mutator is the creation and destruction of DTDA tasks representing pointers held by the mutator. No synchronisation is required, merely the sending of a message indicating that a task has been created (pointer copy) or destroyed (pointer erasure), therefore mutators are never interrupted by collector activity except indirectly on uniprocessor sites where true concurrency is not available and CPU-time is shared. All of Trains, Back Tracing and Trams exhibit this no-interruption property during normal operation.

There is one contrast in latency with Thor: no synchronisation is necessary when garbage is discovered because the region isolation DTD is correctly mapped and aware of pointers held by the mutator as tasks. Thor ignores such pointers and relies on a disruptive synchronisation step to provide safety when the back-trace discovers garbage; this synchronisation in Thor is an implementation peculiarity related to the update model (see Section 4.4.4) and not inherent to the Back Tracing algorithm.

### 6.3.3 Throughput

In common with all other relabel-source approaches to garbage collection, the throughput of this collector is predicted to be very poor since it depends on a suspicion algorithm to discover dead objects from which usefully dead regions can be formed. Where the suspicion algorithm is inaccurate, the collector may make little to no progress, though it turns out that this work may not be entirely wasted.

When the suspicion algorithm selects a dead object, the throughput is excellent: the Surf model of work and progress states that it takes $O(dead\,count)$ relabelling operations to grow a dead region into a usefully dead region, at which point it may be reclaimed. This complexity is the same as for Thor-style (two-region, single-candidate) Back Tracing where the suspicion algorithm forms a correct hypothesis, i.e. suspects garbage.

The Train Algorithm and other relabel-target approaches can make zero progress only if policies controlling their function are flawed. For example in the Train Algorithm where the train-creation policy creates trains at too high a rate, the Train Algorithm may spend its entire time relabelling live objects to ever-newer trains and never form a usefully dead region; it should be noted however

that it is possible to design policies for the Train Algorithm where this does not occur and for which progress is guaranteed.

Barring further breakthroughs into suspicion algorithms, it is therefore likely that the relabel-source approach of Trams has fundamentally lower throughput than the relabel-target approach of Trains because it is more difficult to accurately suspect garbage than it is to decide which train to evacuate. Where the suspicion algorithm presented in this chapter is used, throughput will be lower still because the collector expends time analysing the topology of the live graph.

Thor reduces the cost of wasted relabelling work by preventing relabelling in the believed-live, according to the distance-suspicion metric, region of the graph. This reduces the relabelling costs incurred by an inaccurate suspicion algorithm but it can result in a lower net throughput in the face of large-circumference garbage cycles because back traces that would otherwise succeed are aborted. In contrast, the Tram Algorithm will not abort region growth for reaching too close to the root since the computation of that region may discover structure in the object graph that will permit the reclamation of garbage with better timeliness. The effects of the early-abort tradeoff are expected to be data-structure dependent and perhaps subtle; experimentation in this area would be desirable.

### 6.3.4 Overhead

The time and space overheads of relabel-source seem unavoidably higher than those of relabel-target due to the additional requirements for the maintenance of metadata, particularly if the Tram Algorithm is compounded with a partition collector (PGC).

Grouping objects into partitions and processing them in batches is a valid implementation technique for reducing the number of remsets that need be maintained. In the case of relabel-target collection compounded with a PGC where the PGC performs relabelling work (a common implementation approach, see Chapter 5 or any other published instance of the distributed train algorithm for examples), the remset for a partition is the only information required to perform relabelling work at a partition.

A relabel-source collector must keep remsets so that the region isolation DTDA may be implemented but the remset of a partition provides no information as to what relabelling work is available at that partition. A relabel-source collector that wishes to retain asynchrony of collection therefore must either:

- track what regions are referred to by inter-partition pointers out of a particular partition, or

- decouple relabelling from PGC operation, which requires the determination of intra-partition connectivity.

Both of these options require the maintenance of more meta-data than relabel-target approaches to garbage collection. The latter option is taken by Thor, which uses an variant of Tarjan's Algorithm [92] to calculate connectivity within a partition during PGC operation and then store that information for later use by the relabel-source collector. When the relabel-source collector visits a partition with up-to-date connectivity information, it uses that information to decide which objects in the partition to relabel.

Permitting relabelling to operate close to the roots, unlike Thor, removes the optimisation of only computing intra-partition connectivity for objects far from the root. Metadata space overhead is therefore higher for Trams than Thor though computation time for determining intra-partition connectivity is expected to be substantially similar due to the linear complexity of Tarjan's Algorithm.

### 6.3.5 Timeliness

As per Thor's Back Tracing, timeliness is heavily dependent on the behaviour of the suspicion algorithm. In the worst case, a component of garbage will be detected $O(dead\,count)$ relabelling steps after a member of that component is suspected and a dead candidate region is created.

The timeliness of single-candidate relabel-source (e.g. Thor) will always exhibit this linear relationship with garbage component size since there is no means to gather information on graph structure before a component becomes garbage: traces that occur in Thor from live objects result in zero progress. In contrast, the Tram Algorithm has the opportunity to gather information about the topology of live objects that may be valuable if those objects become garbage.

For any given suspicion algorithm providing the Surf suspicion guarantee, timeliness in the Tram Algorithm is bounded by the same linear complexity relationship with garbage component size as single-candidate Back Tracing; Trams will therefore exhibit the same timeliness as Back Tracing in the worst case. In the best case, it is predicted here that timeliness of isolation detection in the Tram Algorithm can be as good as $O(1)$ relabelling steps if sufficient time is available for the collector to inspect the object graph before parts of it become garbage.

It should be carefully noted that the stated $O(1)$ complexity refers to the prediction that no relabelling need be performed for garbage to be detected once it becomes isolated; the region isolation DTDA must still detect the isolation before any garbage is reclaimed but this time is independent of the live and

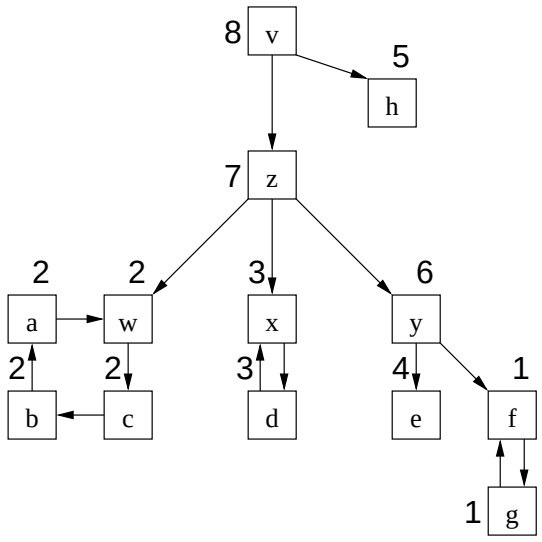


Figure 25: Optimally Labelled Regions

dead component sizes, i.e. it could also be considered $O(1)$. Where garbage is spread across a number of regions, e.g. one component that contains a pointer to a second component, only the first component will be detected in $O(1)$ time; the second component must wait until the first component and all pointers that it contains are destroyed and notification thereof passed to the second component, making it isolated. In the worst case, $n$ garbage objects may be spread across $n$ strongly connected components (e.g. a vine of objects) and these components will be reclaimed in sequential order.

Consider Figure 25 which is a copy of Figure 24; the erasure of $v.z$ triggers the follow sequence of activity:

1. Region 7 is detected as isolated.
2. $z$ is reclaimed, resulting in the erasure of $z.w$, $z.x$ and $z.y$.
3. Regions 2, 3 and 6 are detected as isolated.
4. $a$, $b$, $c$, $w$, $x$, $d$ and $y$ are reclaimed, resulting in the erasure of $y.e$ and $y.f$.
5. Regions 1 and 4 are detected as isolated.
6. $e$, $f$ and $g$ are reclaimed.

The complexity of this reclamation wave is therefore $O(dead\,count)$. Note that there is no explicit wave algorithm implemented to reclaim objects, merely that object reclamation makes progress in a wave across dead regions due to normal operation of the pointer tracking protocol in conjunction with the reclamation of objects implying that the pointers they contain are erased.

By analogy, reference counting collectors exhibit the same behaviour with acyclic garbage: the time to detect trivial isolation of a single object is $O(1)$ but the reclamation process spreads as a wave across the graph and will take $O(dead\,count)$ steps to reclaim a dead component. The Tram Algorithm behaves similarly to a reference-counting collector in the cascading way that it reclaims garbage with the exception that the granularity of the cascading process is strongly connected components rather than single objects.

This analysis of reclamation timeliness/complexity is not specific to the Tram Algorithm, it is derived directly from the Surf model and this behaviour will occur in any collector where the regions formed are congruent with strongly connected components. It is applicable to Trams only because Trams are predicted to form such regions whereas other collectors are not.

### 6.3.6 Suitable Application Behaviour

Because the Tram Algorithm has high overheads compared to relabel-target approaches like the Train Algorithm, it is predicted to be suitable for use only on applications where it has a high probability of successfully detecting the topology of the live graph and therefore provides good timeliness.

Suitable applications are those where data structures are instantiated once and their topology is unchanging in the long term except for the removal of pointers. The non-pointer values may change with no effect on topology.

Examples of such applications are:

- Ray tracers, which instantiate a complex scene graph then traverse it repeatedly to build a bitmapped image, and
- Finite Element Analysis, wherein a mesh of objects is instantiated once and only the values at each node are changed over the course of the computation.

In the Finite Element example, the only topological change typically seen is tearing, i.e. materials separating into discontinuous regions due to stress. Regions never rejoin, therefore there will be no pointer creations that interfere with the Tram Algorithm's determination of topology. Both of these examples are long-running numerically-intensive application classes that give the Tram Algorithm sufficient time to discover the object graph topology and therefore provide excellent timeliness of garbage detection.

Applications that rapidly mutate the structure of their object graph, e.g. compilers, are predicted to be unsuitable for use with the Tram Algorithm.

## 6.4 Conclusions

This chapter instantiates the Surf model to generate a new garbage collector, the Tram Algorithm, that is predicted to have unique properties. This algorithm is new because it occupies a unique design point within the abstract model and it is interesting because Surf predicts that it is uniquely capable of discovering the topology of the live object graph and using that information to improve timeliness of garbage detection at the cost of processing time before objects become garbage.

In discovering topology, Trams attempt to form a labelling that is a good approximation of the optimal labelling wherein every strongly connected component resides in its own region. Where the labelling is optimal, garbage can be detected without further relabelling work after it is created; where this does not occur, the performance of Trams falls back in the worst case to being the same as that of single-candidate relabel-source collection, i.e. Thor-style Back Tracing. Careful design of the suspicion heuristic is required to ensure that the labelling is close to optimal and one such heuristic is described in this chapter.

Significant computation time overhead is involved in arriving at this optimal labelling while the predicted benefit of better timeliness would reduce the garbage load on the system; therefore the Tram Algorithm represents a time-space tradeoff. The Tram Algorithm detects an approximation of distributed strongly connected components in $O(n^2)$ time, compared to the $O(n)$ achieved by Tarjan's Algorithm on a non-distributed graph.

In summary, this chapter shows that not only does the Surf model have descriptive and analytic power with respect to existing collectors, it may be used to instantiate and analyse in detail entirely new garbage collectors with unique properties via exploration of the provided design space.

# Chapter 7

# Conclusion

This chapter looks back on the knowledge gaps that drive the contributions of this thesis, the contributions themselves, the further work that is suggested and finally, the outcomes expected (hoped for) as a result of the publication of this thesis.

## 7.1 Overview

Without solving the halting problem, the garbage collection problem reduces to graph analysis: a garbage collector observes connectivity or lack thereof in a graph and therefore decides which portions are not reachable by the mutator. While garbage collectors — systems that analyse object graphs for the purpose of reclaiming unused data — have previously been designed in an ad-hoc fashion and classified according to their mechanistic details (e.g. copying, tracing, mark/compact, etc), the underlying process by which garbage collectors make progress was obfuscated in mark bits, train management protocols and other implementation details.

Distributed garbage collection in particular is a difficult problem because there exists a tension between the scalability of a garbage collector, which derives from its ability to perform work without communication with other sites, and the completeness of the garbage collector, which requires communication in order that distributed cycles of garbage be reclaimed. In other words, a complete distributed collector cannot operate using only local information, it must use some form of distributed algorithm to discover the extent and reachability of strongly connected components rather than just the reachability of objects. This process of discovering distributed strongly connected components has been described in a number of different ways that are not amenable to formal comparisons without fitting them into some higher level model of their operation.

Given that garbage collection in the most abstract sense is merely a process of applying labels to objects according to connectivity within the graph, it is beneficial to have an abstract model that specifies exactly what one must do to label a graph in such a way that garbage is safely and completely detected. Such a process should capture the essence of a garbage collector, i.e. the means by which it makes progress and the means by which it determines when the progress is complete and that garbage has been detected. An abstract model that captures these two fundamental properties of a garbage collection algorithm permits comparisons between garbage collection algorithms to be drawn at the most fundamental and abstract level, without clouding the analysis with mechanism. To be useful in the comparative analysis and design of distributed garbage collection algorithms, an abstract model should:

- be expressed in a way that takes into account the asynchronous nature of distributed computation and communication,
- define a design space in which garbage collectors exist, with each property (e.g. means of progress) of the collector being a dimension of the design space,
- lead to proofs of correctness for garbage collectors fitting within the model,
- provide a concrete process for analysing existing collectors in terms of the model and instantiating the model to create new collectors,
- be capable of defining a broad range of garbage collection algorithms within the same model of progress, and
- contain sufficient detail that it provides insight into the operation of specific garbage collectors.

This thesis therefore presents in Chapter 3 the Surf abstract model of garbage collection that defines a minimal set of components that may be used to construct a complete garbage collector:

- a definition of the labels that are to be applied to objects,
- a process for changing the labels on objects, and
- a means to detect when the relabelling process has terminated.

By careful construction, the Surf model is defined so that termination of relabelling corresponds to a lack of pointers between certain regions and therefore the ability to infer isolation of some regions. By describing garbage

collection as a relabelling process and its termination, the Surf model captures the essence of garbage collection. Instantiating the model in different ways represents different ways of making progress and different ways of detecting when that progress has discovered garbage.

Having defined an abstract model of garbage collection that claims to not only describe a wide range of existing collectors but also provide insight into their operation, it is necessary to test the model's analytic process and the insight that it provides for each collector. Chapter 4 explores a number of existing collectors by presenting them as instantiations of the model: "mapping" them into the model, comparing the model's predictions with the previously observed and published behaviour of these collectors and finding them to be in agreement. That process demonstrates that it is possible to use Surf in the analysis of a garbage collection algorithm that was independently designed and also that Surf's proofs of safety and completeness may be applied to existing collectors. If a collector is found not to fulfil the requirements laid down by Surf for the proofs to hold, the missing requirement leads a designer towards the modifications necessary to make the collector correct.

If the Surf model is to be useful to garbage collection algorithm designers beyond proofs of correctness and an abstract examination of a collector's progress, it should provide insight that informs the design of components not directly defined by the Surf model. For example, though a large amount of relabelling work may be available at any point in time, an implementation will typically process this work serially at each computation site and some means to decide the order in which to process the work is required. Chapter 5 uses the Surf model of work and progress to determine the presence of work in partitions and thereby ensure progress of an implementation of the Train Algorithm, reducing the collector's complexity to completeness by a factor of $O(object\,count)$ over naive fair policies. These predictions are verified experimentally.

The Surf model defines a design space in which collectors are points; not every point in this design space has yet been occupied by a published collector. Chapter 6 describes the Tram Algorithm, a garbage collector at a previously unoccupied design point. The unique nature of the design point chosen results in previously unseen behaviour from a garbage collector: the predicted ability to discover strongly connected components in a graph and thereby detect garbage with $O(1)$ timeliness in certain circumstances.

By investigating the ways in which Surf may be instantiated and the predictions that it is capable of making, this thesis shows that Surf fulfils all the desirable properties of an abstract model of distributed garbage collection.

## 7.2 Further Work

This research into an abstract description of distributed garbage collection raises a number of questions, primarily related to the model and its limitations and secondarily related to experimentation on implementations. Investigation into Trial Deletion and the Tram Algorithm yield two separate paths of future work that may be exploited to produce a more powerful abstract model of garbage collection.

### 7.2.1 Extensions to the Model

The limitations of Surf are outlined in Section 3.6; those constraints suggest that it would be valuable to extend the model to describe replication and migration of objects. The existence of heaps and collectors that possess those features makes it desirable that the model be capable of describing those features.

A second extension would be to support the concept of multiple phases of model execution for each phase of garbage collector execution, as seen in Section 4.6 for Trial Deletion. Currently, Trial Deletion is described as a special case of Distributed Marking that operates over a restricted suspected region; it would be desirable to have an abstract model of collection that provides a formal methodology for combining such phases and extends the proof of safety and completeness to such combinations in a coherent way. It is speculated that the execution of each phase amounts to the computation of a region with some meaning and that a correct collector will operate by determining the intersection, union or difference of such regions, resulting in a final region that is known to be **Dead**.

### 7.2.2 Tram Algorithm

In designing the Tram Algorithm, it is observed that this collector is to Back Tracing what the Train Algorithm is to Distributed Marking, i.e. a multi-region version with an isolation job per region. It may be possible to instantiate an intermediate relabel-source design in the style of Hughes' Algorithm with each Job representing the reachability of a range of regions instead of a single region. There is also exploration to be done on Trams outside the abstract model, such as the use of external linear-complexity algorithms to prime the system with an optimal labelling as well as an implementation.

Likewise, if a linear-complexity algorithm can be used to determine the strongly connected components in an object store, this is a powerful tool for garbage collection. If further research leads to an algorithm capable of

discovering strongly connected components in a distributed graph in linear time, that would be a major breakthrough because a garbage collector that knows where the strongly connected component boundaries are can detect garbage in constant time.

The Tram Algorithm is the first step towards such algorithms but it has high computational complexity and overheads.

## 7.3 Conclusion

The primary contributions of this thesis are:

- an abstract model that aids in the understanding of how distributed garbage collectors make progress,
- a means of analysing existing garbage collectors,
- a means of designing new garbage collectors,
- a means of proving the safety and completeness of distributed garbage collectors,
- a verification of the abstract model by comparison with published collectors and experimentation with an implementation, and
- an entirely new garbage collection algorithm with unique properties.

This thesis is therefore a journey through the requirement for a new abstract model, the definition of that model and an investigation of the model's predictions. The conclusion of that investigation is that the Surf model can provide specific insight into the operation of a broad range of garbage collection algorithms, therefore it possesses the desired properties of an abstract model. The research gap identified in the first two chapters is filled by this new model.

The process of designing the new abstract model and instantiating collection algorithms from it results in the further insight that strongly connected components are critically important in the detection of garbage, therefore further investigation into the efficient discovery of such components may be a fruitful avenue of further research into garbage collection.

It is hoped that this thesis will change the process by which garbage collectors are designed. Collectors should no longer be assembled in an ad-hoc manner, they should be designed with a formal understanding of how they hypothesise the existence of garbage, how they evaluate and extend those hypotheses via

the connectivity in the graph and how they verify those hypotheses. The result should be that garbage collection design is no longer a black art except at the most mechanistic level but rather a process of following concrete steps and checking off formal requirements for correctness.

By increasing the level of understanding of each collector, it will hopefully become easier to apply the lessons learned from one implementation, express them in terms of the abstract model and then apply those lessons learned in the design of new collectors.

# Bibliography


[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] S.E. Abdullahi and G.A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, Volume 30, Number 3, pages 330–373, September 1998.

[3] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 26–37, February 1996.

[4] L. Amsaleg, M. Franklin and O. Gruber. Efficient incremental garbage collection for client–server object database systems. In *Proceedings of Very Large Databases (VLDB) 1995*, pages 42–53, Zurich, Switzerland, 1995.

[5] G.R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, Volume 3, Number 4, pages 405–430, October 1981.

[6] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[7] A.K. Arora. *A foundation of fault-tolerant computing*. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA, 1992.

[8] M.P Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison. An approach to persistent programming. *Computer Journal*, Volume 26, Number 4, pages 360–365, December 1983.

[9] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, Volume 7, Number 4, pages 539–559, October 1985.

[10] M.P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB (Very Large Data Bases) Journal*, Volume 4, Number 3, pages 319–401, 1995.

[11] L. Augusteijn. Garbage collection in a distributed environment. In de Bakker et al. [32], pages 75–93.

[12] D. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, Budapest, June 2001.

[13] D.F. Bacon, P. Cheng and V.T. Rajan. A unified theory of garbage collection. *SIGPLAN Notices*, Volume 39, Number 10, pages 50–68, 2004.

[14] H.G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, Volume 21, Number 4, pages 280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[15] Y. Bekkers and J. Cohen (editors). *Proceedings of International Workshop on Memory Management*, Volume 637 of *Lecture Notes on Computer Science (LNCS)*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[16] M. Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E.M. Schmidt (editors), *Automata, languages and programming. Ninth colloquium*, pages 14–22, Aarhus, Denmark, July 12–16 1982. Springer-Verlag.

[17] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, Volume 6, Number 3, pages 333–344, July 1984.

[18] A. Birrell, D. Evers, G. Nelson, S. Owicki and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1993.

[19] A. Birrell, G. Nelson, S. Owicki and E. Wobber. Network objects. Technical Report 115, DEC Systems Research Center, Palo Alto, CA, February 1994.

[20] S.M. Blackburn, R.L. Hudson, R. Morrison, J.E.B. Moss, D.S. Munro and J. Zigman. Starting with termination: A methodology for building distributed garbage collection algorithms. In *Proceedings Australasian Computer Science Conference 2001*, Feb 2001.

[21] S.M. Blackburn, R.E. Jones, K.S. McKinley and J.E.B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, Programming Languages Design and Implementation (PLDI), Berlin, June, 2002*, Volume 37(5) of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[22] M. Bowman, Saumya K. Debray and Larry L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, Volume 15, Number 5, pages 795–825, 1993.

[23] R.S. Boyer and J.S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, Volume 31, Number 3, pages 441–458, 1984.

[24] W.F. Brodie-Tyrrell, H. Detmold, K.E. Falkner and D.S. Munro. Garbage Collection for Storage-Oriented Clusters. In *Conferences in Research and Practice in Information Technology*, Volume 26, pages 99–108, Dunedin, New Zealand, 2004.

[25] L. Cardelli. Typeful programming. In E.J. Neuhold and M. Paul (editors), *Formal Description of Programming Concepts*. Springer-Verlag, 1991. Revised 1 January, 1993.

[26] M.J. Carey, D.J. DeWitt and J.F. Naughton. The OO7 benchmark. *SIGMOD Record*, Volume 22, Number 2, pages 12–21, 1993.

[27] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, Volume 13, Number 11, pages 677–678, November 1970.

[28] W. Cockshott, M.P. Atkinson, K. Chisholm, P. Bailey and R. Morrison. Persistent object management system. *Software Practice and Experience*, Volume 14, Number 1, pages 49–71, January 1984.

[29] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, Volume 13, Number 6, June 1970.

[30] G.E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, Volume 3, Number 12, pages 655–657, December 1960.

[31] J.E. Cook, A.L. Wolf and B.G. Zorn. Partition selection policies in object databases garbage collection. In Richard T. Snodgrass and Marianne Winslett (editors), *Proceedings of ACM SIGMOD International Conference on the Management of Data*, Volume 23(2), pages 317–382, Minneapolis, May 1994. ACM Press.

[32] J.W. de Bakker, L. Nijman and P.C. Treleaven (editors). *PARLE'87 Parallel Architectures and Languages Europe*, Volume 258/259 of *Lecture Notes on Computer Science (LNCS)*, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[33] M.H. Derbyshire. Mark scan garbage collection on a distributed architecture. *Lisp and Symbolic Computation*, Volume 3, Number 2, pages 135–170, April 1990.

[34] E.W. Dijkstra. Shmuel Safra's version of termination detection. Technical Report EWD 998, The University of Texas at Austin, 1987.

[35] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[36] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, Volume 11, pages 1–4, August 1980.

[37] A. Einstein. Zur electrodynamik bewegter korper. *Annalen der Physik*, Volume 17, 1905.

[38] K.E. Falkner. *The Provision of Relocation Transparency Through a Formalised Naming System in a Distributed Mobile Object System*. Ph.D. thesis, Department of Computer Science, University of Adelaide, 2000. Available as DHPC Technical Report DHPC-094.

[39] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, Volume 2, Number 1, pages 42–55, January 1980.

[40] F.C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, Volume 31, Number 1, pages 1–26, 1999.

[41] J. Gosling, W. Joy and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[42] D. Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, Volume 20, Number 12, pages 921–930, December 1977.

[43] D. Gries. On believing programs to be correct. *Communications of the ACM*, Volume 20, Number 1, pages 49–50, January 1977.

[44] W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[45] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, Volume 15, Number 4, pages 287–317, December 1983.

[46] P.B. Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, Volume 21, Number 11, pages 934–941, November 1978.

[47] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, Volume 21, Number 8, pages 666–677, August 1978.

[48] P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on LISP and Functional Programming*, pages 168–178, Pittsburgh, PA (USA), August 1982.

[49] R.L. Hudson, R. Morrison, J.E.B. Moss and D.S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelfth Annual Conference*, Volume 32(10) of *ACM SIGPLAN Notices*, pages 162–175, Atlanta, GA, October 1997. ACM Press.

[50] R.L. Hudson, R. Morrison, J.E.B. Moss and D.S. Munro. Where have all the pointers gone. In *Proceedings of 21st Australasian Computer Science Conference*, pages 107–119, Perth, 1998.

[51] R.L Hudson and J.E.B. Moss. Incremental collection of mature objects. In Bekkers and Cohen [15], pages 388–403.

[52] R.J.M. Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud (editor), *Record of the 1985 Conference on Functional Programming and Computer Architecture*, Volume 201 of *Lecture Notes on Computer Science (LNCS)*, pages 256–272, Nancy, France, September 1985. Springer-Verlag.

[53] A. Itzkovitz and A. Schuster. Distributed shared memory: Bridging the granularity gap. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, June 1999.

[54] A. Itzkovitz and A. Schuster. Multiview and millipage: Fine-grain sharing in page-based dsms. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, February 1999.

[55] R.E. Jones. The garbage collection bibliography. http://www.cs.ukc.ac.uk/people/staff/rej/gcbib/gcbib.html.

[56] R.E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[57] N.C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 103–115, Saint-Malo (France), September 1992. Springer-Verlag.

[58] P. Keleher, A. L. Cox and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *International Symposium on Computer Architecture*, pages 13–21, May 1992.

[59] B. Lang, C. Queinnec and J. Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming*, pages 39–50, Albuquerque, New Mexico, January 1992.

[60] C.W. Lermen and D. Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, August 1986. ACM Press.

[61] H. Lieberman and C.E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, Volume 26(6), pages 419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[62] M.C. Little and S.K. Shrivastava. Replicated k-resilient objects in arjuna. In *Workshop on the Management of Replicated Data*, pages 53–58, 1990.

[63] M. Livesey, R. Morrison and D.S. Munro. The Doomsday Distributed Termination Detection Protocol. In *Distributed Computing*, Volume 19, pages 419–431. Springer, 2006.

[64] M.C. Lowry. *A New Approach to the Train Algorithm for Distributed Garbage Collection*. Ph.D. thesis, University of Adelaide, 2004.

[65] M.C. Lowry and D.S. Munro. Safe and Complete Distributed Garbage Collection with The Train Algorithm. In *Proceedings of International Conference on Parallel and Distributed Systems, ICPADS'02*, pages 651–658, Taipei, Taiwan, Dec. 2002.

[66] U. Maheshwari. *Garbage Collection in a Large, Distributed, Object Store*. Ph.D. thesis, MIT Laboratory for Computer Science, September 1997. Technical Report MIT/LCS/TR-727.

[67] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of Principles of Distributed Computing,*

*PODC 1995*, 1995. Later appeared in Distributed Computing, Springer Verlag, 1996.

[68] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by back tracing. In *Proceedings of Principles of Distributed Computing, PODC 1997*, pages 239–248, 1997.

[69] U. Maheshwari and B. Liskov. Partitioned garbage collection of a large object store. In *Proceedings of SIGMOD'97*, pages 313–323, 1997.

[70] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, Volume 43, Number 3, pages 207–221, November 1998.

[71] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, Volume 30, Number 4, pages 195–200, 1989.

[72] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, Volume 3, pages 184–195, 1960.

[73] J. Misra and K.M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 1, pages 37–43, January 1982.

[74] L. Moreau, P. Dickman and R.E. Jones. Birrell's distributed reference listing revisited. *ACM Trans. Program. Lang. Syst.*, Volume 27, Number 6, pages 1344–1395, 2005.

[75] R. Morrison, D. Balasubramaniam, M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro and B.C. Warboys. *ProcessBase Standard Library Reference Manual (Version 1.0.1)*. Universities of St Andrews and Manchester, 1999.

[76] R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C. Kirby, K. Mayes, D. Munro and B.C. Warboys. A compliant persistent architecture. *Software, Practice & Experience*, Volume 30, Number 4, pages 363–386, 2000.

[77] J.E.B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, Volume SE-18, Number 8, pages 657–673, August 1992.

[78] J.E.B. Moss, D.S. Munro and R.L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proceedings of the Seventh International Workshop on Persistent Object Systems*, pages 140–150. Morgan Kaufmann, June 1996.

[79] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. Ph.D. thesis, University of St. Andrews, 1993.

[80] D.S. Munro and A.L. Brown. Evaluating partition selection policies using the PMOS garbage collector. In A. Dearle, G. Kirby and D. Sjoberg (editors), *POS9 Ninth International Workshop on Persistent Object Systems*, pages 104–115, Lillehammer, Norway, September 2000.

[81] D.S. Munro, A.L. Brown, R. Morrison and J.E.B. Moss. Incremental garbage collection of a persistent object store using PMOS. In R. Morrison, M. Jordan and M.P. Atkinson (editors), *Advances in Persistent Object Systems*, pages 78–91. Morgan Kaufman, 1999.

[82] S. Norcross. *Deriving Distributed Garbage Collectors from Distributed Termination Algorithms*. Ph.D. thesis, University of St Andrews, 2003.

[83] G. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little. The design and implementation of Arjuna. *Computing Systems*, Volume 8, Number 3, pages 255–308, 1995.

[84] S.P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, Volume 17, pages 43–46, July 1983.

[85] H.C.C.D. Rodrigues and R.E. Jones. A cyclic distributed garbage collector for Network Objects. In Ozalp Babaoglu and Keith Marzullo (editors), *Tenth International Workshop on Distributed Algorithms WDAG'96*, Volume 1151 of *Lecture Notes on Computer Science (LNCS)*, Bologna, October 1996. Springer-Verlag.

[86] A. Schiper and A. Sandoz. Strong stable properties in distributed systems. *Distributed Computing*, Volume 8, Number 2, pages 93–103, 1994.

[87] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In W. Olthoff (editor), *Proceedings of European Conference on Object Oriented Programming, 1995*, Volume 952 of *Lecture Notes on Computer Science (LNCS)*, pages 235–252, Aarhus, Denmark, August 1995. Springer-Verlag.

[88] S.K. Shrivastava, G.N. Dixon and G.D. Parrington. An overview of Arjuna: A Programming System for Reliable Distributed Computing. Technical Report 298, University of Newcastle-upon-Tyne, Newcastle-upon-Tyne, England, November 1989.

[89] G.L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, Volume 18, Number 9, pages 495–508, September 1975.

[90] T. Sterling, D. Savarese, D.J. Becker, J.E. Dorband, U.A. Ranawake and C.V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[91] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proceedings of Second Symposium on Parallel and Distributed Processing*, pages 719–724. IEEE, December 1990.

[92] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, Volume 1, Number 2, 1972.

[93] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[94] G. Tel and F. Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, Volume 15, Number 1, pages 137–149, January 1993.

[95] G. Tel, R.B. Tan and J. van Leeuwen. The derivation of graph marking algorithms from distributed termination detection protocols. *Science Of Computer Programming*, Volume 10, Number 2, pages 107–137, 1988.

[96] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–365, 1936-1937.

[97] D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, Volume 19, Number 5, pages 157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[98] F. Vaughan and A. Dearle. Supporting large persistent stores using conventional hardware. In *Proc. of the Fifth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 29–50, San Miniato Pisa (Italy), September 1992.

[99] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In de Bakker et al. [32], pages 432–443.

[100] J. White. A high level framework for network-based resource sharing. IETF Network Working Group Request for Comments: 707, January 1976. http://www.ietf.org/rfc/rfc707.txt.

[101] P.R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [15].

[102] J.N. Zigman. *A General Framework for the Description and Construction of Hierarchical Garbage Collection Algorithms*. Ph.D. thesis, Australian National University, June 2004.