Birrell's Distributed Reference Listing Revisited

Luc Moreau
School of Electronics and Computer Science, University of Southampton

and
Peter Dickman
Department of Computing Science, University of Glasgow

and
Richard Jones
Computing Laboratory, University of Kent

The Java RMI collector is arguably the most widely used distributed garbage collector. Its distributed reference listing algorithm was introduced by Birrell et al. in the context of Network Objects, where the description was informal and heavily biased toward implementation. In this paper, we formalise this algorithm in an implementation-independent manner, which allows us to clarify weaknesses of the initial presentation. In particular, we discover cases critical to the correctness of the algorithm that are not accounted for by Birrell. We use our formalisation to derive an invariant-based proof of correctness of the algorithm that avoids notoriously difficult temporal reasoning. Furthermore, we offer a novel graphical representation of the state transition diagram, which we use to provide intuitive explanations of the algorithm and to investigate its tolerance to faults in a systematic manner. Finally, we examine how the algorithm may be optimised, either by placing constraints on message channels or by tightening the coupling between application program and distributed garbage collector.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Distributed garbage collection; distributed reference counting/listing; proof of correctness

Authors' address: Luc Moreau, `L.Moreau@ecs.soton.ac.uk`, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.
Peter Dickman, `pd@dcs.gla.ac.uk`, Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK.
Richard Jones, `R.E.Jones@ukc.ac.uk`, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.

1. INTRODUCTION

Modern programming systems have popularised the idea of distributed object-oriented languages that abstract away from the physical reality of distributed memory by providing the same invocation interface for local and remote objects. These abstractions may be part of the language (for example, Emerald [Jul et al. 1988]) or provided through libraries (such as Modula-3's Network Objects [Birrell et al. 1994a; 1994b; 1995] or Java's RMI package [Sun Microsystems 1996]). A crucial component of this abstraction layer is a distributed garbage collector that extends uniprocessor garbage collection [Jones 1996] in order to reclaim unused objects in a distributed setting.

In a significant number of distributed languages, among them Java RMI, distributed garbage collection relies on distributed reference counting or listing algorithms. Intuitively, distributed reference counting *counts* the number of remote references to an object, whereas distributed reference listing keeps a *list* of remote processes holding a reference to the object. Such algorithms provide timely, low-cost, incremental recovery of garbage objects by determining when the last reference to an object is discarded or overwritten. Besides their technical appeal, such algorithms are also popular from an implementation viewpoint because they can be integrated relatively easily with existing uniprocessor garbage collectors. The drawback is that distributed reference counting and listing algorithms, in their basic forms, are unable to collect distributed cyclic data structures. However, this can be addressed through hybridisation with a more expensive complete[1] collector or partially overcome by inclusion of a limited cycle collector [Rodrigues and Jones 1998; 1996; Lang et al. 1992].

Because of their wide application, distributed reference counting and listing algorithms are an important subject of investigation. In this paper, we provide new insights into the principles, formalisation, implementation and extension of a critically important algorithm that is in widespread use in distributed systems. This distributed reference listing algorithm was initially presented by Birrell, Evers, Nelson, Owicki and Wobber, and for short we will refer to it as *Birrell's algorithm*. Through its use in Java, it is arguably the most disseminated and widely used algorithm in the distributed reference counting and listing family.

Birrell's reference listing algorithm is built into a Remote Procedure Call (RPC) layer constructed over a lossy message-passing layer. First presented in DEC SRC Technical Report 116 [Birrell et al. 1993], an adjunct to the Network Objects Technical Report 115 [Birrell et al. 1994a], Birrell gives an informal description and an outline proof of safety and liveness properties of their algorithm. Unfortunately, the informality of their presentation leaves a number of questions open and the underspecification of key points of their algorithm provides only limited guidance to the implementor.

Distributed garbage collection algorithms are difficult to specify or implement correctly. In this paper, we identify the limitations of the original presentation and provide a formal description of the algorithm. We represent the distributed system

[1] A *complete* collector is one that guarantees to reclaim all garbage eventually. Typically, tracing collectors are complete whereas reference counting/listing collectors are not.

as an abstract state machine, composed of processes that communicate through asynchronous message passing, whose behaviour is described by state transitions. This representation has five advantages. *(i)* We can describe the behaviour of the system with a novel, and highly intuitive, graphical representation. *(ii)* Our description is independent of implementation technique (such as RPC or stack-based run-time systems). *(iii)* In order to address concurrent execution, the *critical sections* of the algorithm are made explicit. *(iv)* It permits us to provide new, simpler proofs of key properties (*safety* and *liveness*) than those based on (notoriously difficult) temporal reasoning. *(v)* Finally, we are able to clarify details of Birrell's algorithm to ensure that future implementors avoid subtle pitfalls.

Outline of the paper

We motivate our approach in Section 2 by highlighting problematical subtleties in the original presentation and by showing how the original informal proofs depend on hard-to-formalise aspects of the implementation. To allow more robust proofs and simplify the comparison with other algorithms, the bulk of this paper describes (Section 3) and proves properties (Section 4) of a formalisation of Birrell's algorithm. We suggest variants and optimisations of the algorithm in Section 5. In Section 6, we show how our formalisation may be extended to accommodate communication and process failures. We compare related work in Section 7 and identify directions for further work in Section 8 before concluding in Section 9.

2. OVERVIEW OF BIRRELL'S ALGORITHM

Distributed reference counting and listing algorithms have been developed over the last twenty years [Jones 1996; Plainfossé and Shapiro 1995; Abdullahi and Ringwood 1998] and are all variations and extensions, to a greater or lesser extent, of the original reference counting algorithm proposed by Collins [1960] for single-threaded LISP-style languages on uniprocessors.

In this section, we introduce the terminology adopted to describe such algorithms, present the naive definition of reference counting in a distributed setting and revise Birrell's answer to this problem.

2.1 Terminology

Object-based distributed systems compute using a dynamically manipulated graph of objects and references. From the point of view of the garbage collector, the application program simply mutates the object graph. Thus, the program is referred to as the *mutator* in memory management literature [Dijkstra et al. 1978]. Whenever a new instance of an object is created, a *reference* to it is provided to the creating process. Such references may be *discarded* by overwriting the storage containing them with another reference or the distinguished value NULL. References may also be *copied*, with the copy being placed in another storage slot or passed in a *message* to another object. References may be *received* in messages. A process can send messages only to objects for which it holds references. Objects continue to exist until deemed to be *garbage* by a garbage collection (GC) algorithm, at which point they may be *reclaimed* and any references they hold are discarded.

Objects reside in *processes*, which partition the computational and storage resources. Messages pass between objects and when the source and destination are in different processes they are passed between the processes in point-to-point channels, which may have certain key properties, such as being FIFO or lossy. A message sent by a process, or scheduled to be sent, and not received by its recipient is said to be *in transit*. In a given process, a reference is said to be *local* if it refers to an object allocated in the same process; alternatively, a reference is said to be *remote* (or *global*) if it refers to an object allocated in another process. We usually refer to the *owner* of a reference as the process that initially allocated the object to which the reference refers.

In a garbage-collected distributed system, objects are allocated in each process in a local heap managed by a local garbage collector. The purpose of the garbage collector is to reclaim heap resources occupied by objects that can no longer affect the computation, i.e. that are *dead* or *garbage*. The primary safety guarantee is that the collector should not reclaim *live* objects. Most garbage collector implementations define the liveness of an object in terms of its *reachability*[2] from a set of roots. *Roots* are distinguished memory locations holding references to heap objects; in general, roots include processor registers, slots in program stack frames and global variables. An object in the heap is *locally reachable* if and only if a reference to the object is held in a root or in another locally reachable heap object [Jones 1996].

In a distributed environment, references to objects allocated in one process may be used by computations running in another process. Using the same 'liveness by reachability' estimate, an object may be live because it is reachable from a live object in a remote process via a remote reference, even though it may not be locally reachable. Conversely, the purpose of a *distributed* garbage collector is to reclaim space used by objects that are no longer globally reachable. Formally, an object o is *globally reachable* if one of the following conditions holds:

(1) o is locally reachable in a process taking part in the computation,
(2) there is a locally reachable object that contains a remote reference to o,
(3) there is a globally reachable object that contains a reference (local or remote) to o,
(4) there is a message in transit between between processes that contains a reference to o.

In order to ascertain whether an object is globally reachable in a computationally efficient manner, distributed reference counting or listing algorithms tend to associate extra state with objects, so that global reachability can be determined solely on the basis of local information. Distributed reference counting or listing algorithms are thus responsible for maintaining such state as computation proceeds. The timeliness and incrementality of these algorithms is achieved by performing key actions when references are copied or overwritten, and their low cost is achieved by piggy-backing collector messages onto mutator messages or by completing certain actions in the background (when the processor would otherwise be idle) in order

[2]We note, however, that reachability is a conservative estimate of liveness, since objects may continue to be reachable long after their last use [Röjemo and Runciman 1996; Hirzel et al. 2002; Shaham et al. 2002].

to avoid delaying the mutator. A distributed garbage collector must also correctly handle references contained in messages that are in transit between processes.

Families and variants of algorithms arise from differing choices as to: how much information is retained and where it is held; which participant in message exchanges is responsible for the GC-related activity; the degree of, and approach to, fault tolerance; the precise semantics of remote communications; and the eagerness or laziness of execution of GC actions.

2.2 Naive Distributed Reference Counting

Reference counting was initially conceived in the context of uniprocessor applications [Collins 1960]. In a reference counting system, each object is associated with a reference counter that is incremented every time a new reference to the object is created, and decremented every time such a reference is discarded. A reference counter equal to zero indicates that there is no reference to the object and therefore that the space occupied by the object may be reclaimed safely. A corollary is that the reference counts of objects that are members of an isolated cyclic data structure never fall to zero, and hence such cycles cannot be reclaimed purely by reference counting.

This reference counting technique cannot be extended naively to a distributed context. Suppose that, as before, a reference counter is associated with each object, but two collector messages, *increment* and *decrement,* are introduced to mark remote processes' requests to increment and decrement the object's reference counter when references to the object are copied or discarded. Unfortunately, this results in an incorrect algorithm.

The essence of the problem is summarised in Figure 1, in which a reference r to an object located in process p_1 is passed from process p_2 to process p_3, immediately followed by process p_3 discarding its reference r. A naive extension of reference counting would send an *increment* message to request an increment of the counter for p_1 when r is sent to p_3, and it would send a *decrement* message to request the counter for p_1 be decremented when p_3 discards its reference r. Unfortunately, a race condition between *increment* and *decrement* messages may cause the counter for p_1 to be decremented before it is incremented, possibly making it zero temporarily. A zero counter would cause the object to be collected in process p_1, even though p_2 still holds a live reference r.

Naive reference *listing* would maintain the set of processes having access to the reference, instead of maintaining a counter counting the number of references. Here, an *increment* message has the effect of adding the process that sent the message to the set, whereas a *decrement* message has the effect of removing the process that sent the message. Naive distributed reference listing suffers from the same problem as naive distributed reference counting — the set of processes may temporarily become empty if the *decrement* message is processed before the *increment* message.

2.3 Birrell's Algorithm

Many algorithms have been designed to combat the problem inherent in naive distributed reference counting, including Weighted Reference Counting [Bevan 1987; Watson and Watson 1987], Indirect Reference Counting [Piquer 1991], and those of Lermen and Maurer [1986], Birrell et al. [1993], Moreau and Duprat [2001] and

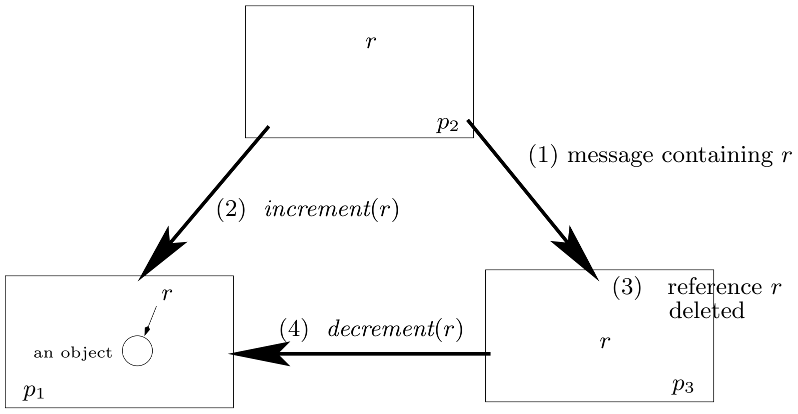


Fig. 1. Naive Extension of Reference Counting to a Distributed Context

Dickman [2000]. We discuss some of these in the related work section, but we focus here on Birrell's algorithm, which is used in the distributed garbage collector for Java RMI.

As the rest of the paper focuses on our own presentation of Birrell's algorithm, we quote here relevant excerpts[3] of the original technical report [Birrell et al. 1993] and discuss some of the issues raised by this presentation. First, the algorithm is presented as follows in the absence of failures.

Terminology
[Section 1] A *network object* is an object that can be shared by processes in a distributed system. The process that allocated the network object is called its *owner*, and the instance of the object at the owner is called the *concrete* object. Other processes, known as *clients* may have references to the object.
A client cannot directly read or write the data fields of a network object to which it holds a reference, it can only invoke its methods. A reference in a client program actually points to a *surrogate* object, whose methods perform remote procedure calls to the owner, where the corresponding method of the concrete object is invoked. There is at most one surrogate for an object in a process, and all references in the process point to that surrogate.
A network object is marshaled by transmitting its *wireRep*, which consists of a unique identifier for the owner process, plus the index of the object at the owner.
[Section 2] Each process maintains an *object table*, which maps a wireRep $w(o)$ to the local instance of the corresponding network object O, if there is one. For the owner of an object, the table contains a pointer to the concrete object. A concrete object must be in the table whenever another process has a surrogate for it.
A *dirty set* is maintained for each object by its owner. The dirty set contains

[3]We quote Birrell at length in order to avoid giving *our* interpretation of his description.

identifiers for all processes that have surrogates for the object. When the dirty set becomes empty, the object can be removed from its owner's object table.

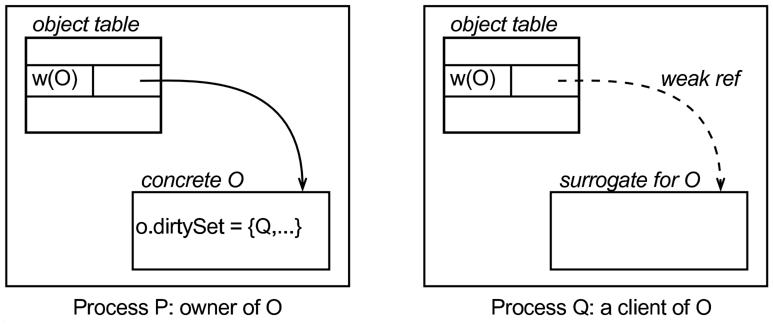


Fig. 2. Object tables at owner and client processes [Birrell et al. 1993]

Serialisation (marshalling)

[Section 2.1] Suppose P marshals [also called *serialising*] a network object O to process Q, as an argument or result of remote method invocation. P may be the owner of O, or it may be a client that has a surrogate for O. In either case, P sends Q the wireRep $w(O)$. When $w(O)$ arrives, Q looks it up in its object table to see if there is a corresponding local object.
If Q finds either a surrogate or concrete object, that object is used as the required argument or result. Note that if a client transmits a remote object back to its owner, this use of the object table causes the owner to access the concrete object; no surrogate is created.
If Q does not find an object in the table, there are two possibilities to consider. First, $w(O)$ may be in the table but mapped to a NIL reference. In this case surrogate creation is under way, and the thread doing the unmarshaling suspends itself until the surrogate is created or the attempt fails. Alternatively, $w(O)$ may not be in the table, or it may be there with a null weak ref indicating that a surrogate existed but had been collected. In this case, the recipient must create a new surrogate. It first enters $w(O)$ in the table with a mapping to NIL, releases the lock on the table, and then makes a *dirty call* [emphasis added] to the owner of the object. Assuming no communication failure, the owner receives the call and adds Q to O's dirty set. When the dirty call returns, Q creates a surrogate for O and enters it in the object table.

Race conditions

[Section 2.1] There is one more wrinkle to be considered in transmitting a network object. This is the potential race condition between the dirty call from client Q and a *clean call* [emphasis added] from a client whose surrogate has been deleted. If the clean call arrived first, and if it left $O.dirtySet$ empty, then O might be removed from its owner's object table

and its space reclaimed by the local collector. When the dirty call arrives, the object is no longer available.
To prevent this scenario, we make sure that $O.dirtySet$ remains non-empty while O is being transmitted. When the sending process P is O's owner, this is accomplished by putting P into $O.dirtySet$ until an acknowledgement from Q indicates that the reference has been received. Since Q sends the acknowledgement after completing the dirty call, this guarantees that O's dirty set remains non-empty, and its memory is not collected. When P is not the owner of O, it must have a surrogate for O. This surrogate is kept reachable until Q's acknowledgement is received. Since a reference to the surrogate is on the stack during transmission, we simply ensure that the transmitting procedure not return until acknowledgement from Q is received. So long as this surrogate is reachable, the basic collector invariant guarantees that P is in $O.dirtySet$ and O's space will not be collected.

[Section 2.2] Collection of surrogates is the responsibility of the client's local garbage collector. When the client's collector determines that a surrogate is unreachable, the object's owner must be informed so that the client can be removed from its dirty set. We have already mentioned how weak refs allow the distributed a garbage collector to be informed of surrogate collection so it can take this action. To recap, when the client's collector determines that the surrogate is not reachable (except from the weak ref in the client's object table), it prepares to reclaim the surrogate's memory. However, it first schedules a clean up routing that was registered when the weak ref was created and replaces the weak ref with a special null value.
When the cleanup routine begins execution, it checks the object table to see if the entry for this object's wireRep still has the special null weak ref. If not, a new surrogate for the objection will have been created (or will be in the process of being created) and no clean up action is required. But if the null weak ref is still present, clean up action is necessary. The object table entry is removed, and the wireRep is put on a queue of objects to be processed later by a cleaning demon. This demon is responsible for sending clean calls to the owner.

2.4 Weaknesses

While we recognise that we have only quoted particular excerpts from the original presentation of Birrell's algorithm, we believe that they are sufficient both to summarise the key ideas of the algorithm and to reveal the limitations of that presentation.

(1) The discussion of the algorithm is tightly bound to specific communication assumptions, namely the remote method invocation paradigm.
(2) The algorithm makes strong assumptions about the specific run-time implementation. Our point is not that Birrell's implementation is deficient, but that their description is biased towards a particular implementation or use of the algorithm. These implementation details are also hard to formalise.
    For instance, it is assumed that method invocation will push references to

its argument objects onto the stack, thereby making them locally reachable for the duration of the call. Birrell's presentation also requires that there be precisely one surrogate in process p for a remote concrete object. Thus, their algorithm can be thought of as 'object listing' rather than reference listing; the implementation of concrete object O in Figure 2 makes this very plain. However, this is an implementation decision and not a requirement of reference listing *per se*: more generally, one might associate the list of remote processes with an entry in the object table (of references) rather than the concrete object itself. Furthermore, Birrell's presentation makes use of specific GC-related techniques such as weak references.

(3) From a technical viewpoint, important aspects of the algorithms are underspecified. Consequently, it is left to the implementor to judge how to complete the algorithm:

   (a) Critical sections are generally underspecified.

   (b) A race condition is identified in Section 2.1 but a solution is discussed only for the case of a method invocation, during which a reference on the stack remains accessible. No solution is presented for a reference returned as part of an invocation result.

   (c) It is unclear how the algorithm should behave when a reference is received but the cleanup action has been already initiated.

   (d) It is unclear how the algorithm would handle parallel sending of references to the same destination.

(4) The informal proof by Birrell depends on hard to formalise aspects of the implementation. Indeed, the correctness proof of a system that uses the run-time stack to keep references live during the invocation of a remote procedure requires the formalisation of the run-time system and method calling conventions. Likewise, the use of GC weak references (as in Figure 2) requires a formal account of local garbage collection, including reachability and finalisation. A further problem with this approach is that it brings elements of liveness into the proof of safety. Birrell's proof is therefore unconvincing, as none of these elements have been formalised. In fact, we shall see in the rest of the paper that they need not be formalised in order to prove the correctness of the algorithm provided the algorithm is specified at a suitable level of abstraction. Finally, the informal proof of Birrell's algorithm uses temporal reasoning which is notoriously difficult to handle.

In summary, the initial presentation of Birrell's algorithm has shortcomings that have potentially serious consequences. Implementors must make decisions on how to extend the algorithm for the cases that are not completely specified: this may result in incorrect implementations or implementations that cannot inter-operate properly. It is very difficult to port the algorithm to a different context, such as a message passing system or a distributed termination mechanism [Tel and Mattern 1993]. For algorithm designers, it is quasi-impossible to compare such an algorithm with others as its principles are hidden behind its implementation. We address these issues below in a revisited presentation of the algorithm.

3. DESCRIPTION OF THE ALGORITHM

In this section, we present a precise and implementation-independent formalisation of Birrell's algorithm, accompanied by an informal description of the algorithm. The formalisation is intuitive since it uses a notation that bears some similarity with executable pseudo-code, but it is also rigorous and suitable for both manual and mechanical proofs. By making our formalisation independent of implementation technique, we do not restrict the implementer's design decisions (e.g. by requiring them to use specific mechanisms such as weak references). We also present a novel graphical representation of the state transition diagram for the algorithm which we believe substantially improves understanding of the algorithm.

3.1 Design Philosophy

We represent a distributed system — in this case, the distributed reference listing algorithm — by an *abstract machine*, identifying a finite set of *processes* able to communicate by *asynchronous message passing*. The behaviour of the distributed system is described by the transitions that the abstract machine is allowed to perform and that modify the internal state of processes and/or communication channels.

In the past, we have successfully used this formalism to describe a new distributed reference counting algorithm [Moreau and Duprat 2001] and to specify the behaviour of a fault-tolerant directory service for mobile agents [Moreau 2001a; 2002]. Our practical experience, reported in the first of these papers, has shown that the formalism is suitable for mechanical proof derivations, which we have carried out for these algorithms using the Coq theorem prover [Barras et al. 1997].

The notation allows us to express any form of transitions, but is not prescriptive about their nature, complexity or granularity. In this paper, we adopt a systematic coding style, which we summarise as follows.

—A transition involves only one process at a time.

—The input of a message changes only a process's internal state. Sometimes we need to express that the receipt of a message is meant to trigger the sending of another message (for example, when an acknowledgement has to be sent). In such a case, we use two transitions: the first one receives the input message and stores some useful information in a *to do* table; the second transition is in charge of extracting the information from the *to do* table and generating a suitable output message.

—We generate acknowledgement messages explicitly (rather than relying on particular implementations such as RPC).

Not only does such a coding style make rules simple, but it also brings realism to the formalisation, as it offers a route to implementation:

—Inputs and outputs are desynchronised.

—The size of critical sections is minimised for each transition of the abstract machine, which favours proper concurrent execution in the implementation.

—As far as implementation is concerned, outputs can be generated asynchronously by a background daemon that reacts to changes in the *to do* tables.

3.2 States and Transitions: a Graphical Representation

Birrell's algorithm defines a life cycle for objects, which we refine and specify in this section. An object's life cycle typically evolves as messages containing references to this object are processed by the distributed reference listing algorithm. More precisely, it is the state of the *reference* held by the client that evolves (reference received, dirty call acknowledged,. . .). Thus, we first identify the messages that the algorithm handles, and then continue with a study of a reference's life cycle.

Messages. We do not make any assumptions of the environment in which the reference listing algorithm can be embedded. For example, it could be embedded in a distributed object language (such as Java RMI) or in a distributed termination detection system. We indicated in the previous section that, whereas Birrell's algorithm assumed a communication layer based on remote procedure calls, we model the algorithm as an asynchronous message passing system. Consequently, we will express calls as two messages: an outbound message followed by a return message that carries the call completion acknowledgement (including, possibly, a result). Birrell's algorithm supports three different calls, which we in turn express as the six messages summarised in Figure 3 and described below.

reference copy	copy
copy acknowledgement	copy_ack
dirty call	dirty
dirty call acknowledgement	dirty_ack
clean call	clean
clean call acknowledgement	clean_ack

Fig. 3. Algorithm messages and their formal notation

From the reference listing algorithm's viewpoint, references can be copied between processes, which we express by a *reference copy message*. In practice, one (or more) copy message(s) could be part of a remote method invocation (whether as arguments or as a return value) or could be sent alone, depending on the environment in which the algorithm is embedded. For the algorithm, it is important that a reference sender be notified of the receipt of a reference — this is the role of a *copy acknowledgement* message. A *dirty call* message allows a process to register a new instance of a reference with the reference's owner; such a call is followed by a *dirty call acknowledgement* that informs the sender of the dirty call of the successful completion of that call. Symmetrically, a *clean call* message allows a process to notify the reference's owner that the reference has been discarded, and such a message is followed by a *clean call acknowledgement* that marks the end of the cleanup operation.

It is important to realise that we present the algorithm in an abstract manner, that is not muddied by implementation details. It is *not* a requirement that the algorithm be implemented as an asynchronous distributed system, although such a form of modelling is very appropriate for identifying the flow of information in the system. At a later stage, an implementor may make particular implementation decisions, such as to batch up messages or to piggy-back GC messages onto mutator messages in order to reduce communication costs. We also note that Birrell's algorithm makes a further implementation decision by imposing at most one surrogate per object. Our formalisation is about references and not about objects; we therefore do not impose a similar constraint, but implementors may elect to adopt it in their specific implementations.

Life cycle. Before an object's reference is received (in a copy message) by a process p for the first time, the reference is unknown and considered to be nonexistent by p. Such an initial state is marked $\perp$ and denotes the reference's pre-existence[4]. When a reference is received by p, the reference moves to a temporary state in which it is known by p but is not yet usable, since the reference has not been registered with its owner. Following Birrell's algorithm, we refer to this state as nil. As soon as the object's owner is aware of the existence of the remote reference, and p is aware that the owner is aware (through a dirty call followed by an acknowledgement), the reference becomes usable, which we indicate by the OK state. As the reference becomes locally unreachable on p, the reference is scheduled for cleanup. When the cleanup is triggered, a clean call is initiated and the reference moves to the state ccit (standing for 'clean call in transit'), in which the reference is no longer in use but has not been completely recycled by the process. As soon as a clean call acknowledgement is received, the space used by the reference is recycled. This marks the end of the reference's existence in this process p: its state therefore reverts to $\perp$.

The states and transitions of the state machine can be illustrated by a state transition diagram. Our graphical representation[5] of the state transition diagram for (our formalisation of) Birrell's algorithm lays out its states as vertices of a cube, with cube edges marking permitted transitions (see Figure 4). Our representation has several advantages. It provides the implementor with an intuitive, yet precise, description of the algorithm that is independent of implementation technique. It readily identifies transitions (and their associated states and messages) in a systematic fashion. For example, consider the cube's x-axis to be horizontal, y-axis to be vertical and z-axis to be into the page. Now, all copy messages lie on edges parallel to the x-axis, dirty and clean calls to a reference's owner lie on edges parallel to the z-axis and acknowledgement messages from a reference's owner lie parallel to the y-axis, and so on. The layout makes it easy to determine which state changes might be possible, simply by considering the edges at each vertex. One outcome has been that this representation immediately revealed the need for the introduction of a new state, not present in Birrell's description, but crucial for correctness.

The sequence of transitions $\perp \rightarrow \mathsf{nil} \rightarrow \mathsf{OK} \rightarrow \mathsf{ccit} \rightarrow \perp$ is purely determined by the process p sending messages to the reference's owner and receiving acknowledgements from the owner. In addition, other instances of the same reference may be received by p at any moment during the reference's life cycle. These cause a movement along the cube's edges in three cases: from $\perp$ to nil if the reference was

[4]Webster's dictionary defines pre-existence as 'existence in a former state, or previous to something else' (Merriam-Webster's Collegiate Dictionary, Tenth Edition); this captures our concept of the reference being in an initial state, before its existence is revealed to the receiving process.
[5]Colour versions of our diagrams are available at `http://www.ecs.soton.ac.uk/~lavm/birrell`.

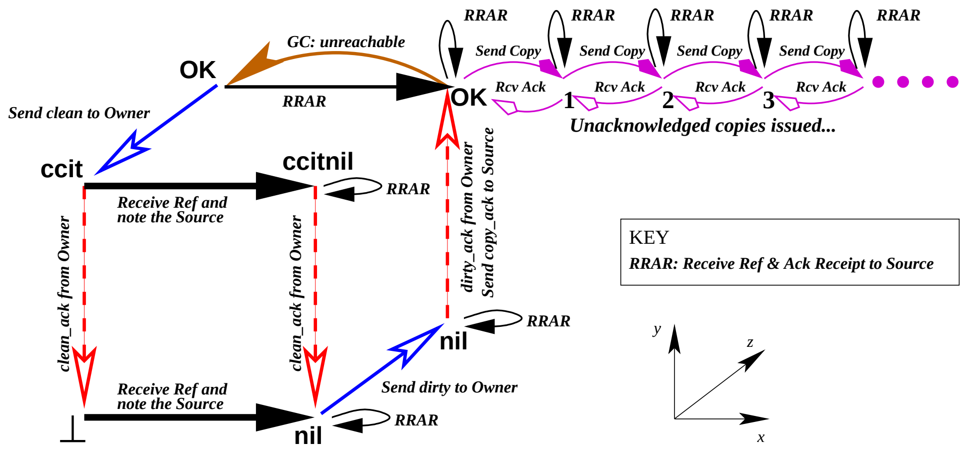


Fig. 4. Birrell's algorithm transitions

unknown by the process p; from an unreachable OK reference to a reference that is reachable again; and from a ccit state to a new state, in which a clean call is still in transit, but the reference is ready to be used again, so its deallocation should be cancelled, which we mark by ccitnil. The latter state was not explicit in Birrell's algorithm but is immediately apparent from our layout of the state transition diagram. This new state turns out to be crucial in order to ensure the algorithm's correctness.

Once registered with its owner, a reference becomes usable (the OK state) and may be sent to remote processes. The distributed reference listing algorithm must make sure that the reference remains locally reachable until the acknowledgement of the copy message is received. We ensure this by the sequence of states OK $\rightarrow$ **1** $\rightarrow$ **2** $\rightarrow \ldots$ in the diagram. Every time a reference is sent remotely, the state changes from i to $i+1$; every time the copy of the reference is acknowledged, the state changes from i to $i-1$.

Birrell's algorithm is distinguished by the number of states in a reference's life cycle, their position on the cube and the permitted transitions allowed between them. For instance, there is no transition from ccitnil to OK because the algorithm prevents the sending of a dirty message until the clean call acknowledgement message has been received. The spatial representation of states in our diagram also lends itself to a very intuitive explanation of the state of a reference: has a process p declared that it has a reference? is the owner aware of this? and, is the reference usable? Figures 5, 6 and 7 illustrate these questions.

In Figure 5, states are partitioned by a vertical slicing of the cube. States on the 'front' plane indicate that the process p has not declared possession of a reference while, in the 'back' states, the process has declared possession. In Figure 6, states are partitioned by a horizontal slicing of the cube. In 'lower' states, a process p has received a reference, but has not received a confirmation from the owner that the

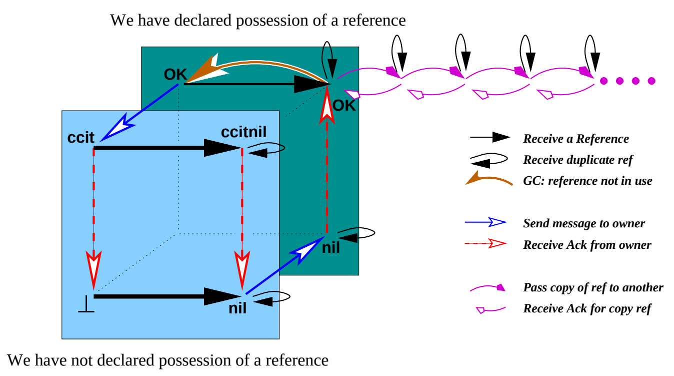


Fig. 5. Depth Slicing: has a process announced its possession of a reference?

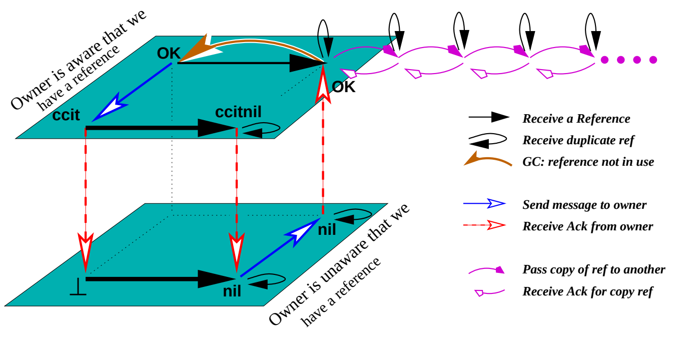


Fig. 6. Horizontal Slicing : does the owner know that a process has a reference?

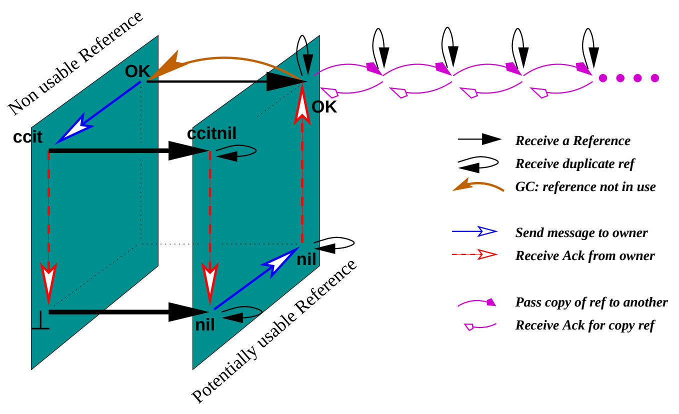


Fig. 7. Vertical Slicing: is a reference usable?

reference has been properly registered. In 'upper' states, process p considers that the owner is aware that p has a reference. In Figure 7, the states on the 'left' plane describe situations in which the reference cannot be used by the process (i.e. pre-existence or post-cleanup). The states on the 'right' describe situations in which a reference has begun its active existence.

Figure 4 presented a fairly complete summary of Birrell's algorithm but overlooked some of its details, such as when should messages be issued, what information needs to be shared between states, and so on. In the following section, we answer all these questions with a complete specification of the algorithm.

3.3 Algorithm Formalisation

In this section, we introduce our specification of Birrell's algorithm based on the notion of an abstract machine modelling an asynchronous distributed system, and the life cycle and transitions described in Section 3.2. First, we define our notation, inspired by previous formalisations, namely another distributed reference counting algorithm [Moreau and Duprat 2001] and a distributed directory service for mobile agents [Moreau 2001a].

3.3.1 *Notation.* The algorithm is formalised by an abstract machine, whose state space is shown in Figure 8. In this abstract machine, we model only messages exchanged by the distributed reference listing algorithm, and we do not model any form of computation in which it would be used.

A finite number of processes are involved in the algorithm, and we consider a finite number of remote references. The set of messages is defined by an inductive type, whose constructors are named according to the messages presented in Section 3.2, namely copy, copy_ack, dirty, dirty_ack, clean and clean_ack. Communication channels are assumed to be reliable and not to duplicate messages; we do not make any assumption about message order, and therefore we represent channels as bags of messages between pairs of processes. We shall relax these assumptions when discussing fault-tolerance (Section 6).

A series of tables are defined as functions whose first argument is a process. While the formal notation sees such functions as 'global' in the abstract machine, we expect their implementation to be distributed across each process. A configuration of the abstract machine consists of a tuple composed of all tables and message channels.

We assume that each reference r refers to an object that has initially been allocated and created by a given process, which we refer to as its *owner*. In our formalisation, we define a function

$$owner \,:\, \mathcal{R} \rightarrow Process$$

that maps each reference onto the process that owns the object to which the reference refers.

We use *pseudo-statements* such as *post*, *receive* or table updates in order to provide an imperative look to the algorithm. Informally, $post(p_1, p_2, m)$ inserts a message m into the channel from process p_1 to process p_2, and $receive(p_1, p_2, m)$ collects the message. Formally, these pseudo-statements act as configuration transformers and are defined as follows.

$$
\begin{array}{rcll}
\mathcal{P} & = & \{p_0, p_1, \ldots, p_{n_s}\} & \text{(Set of Processes)} \\
\mathcal{R} & = & \{r_0, r_1, \ldots, r_{n_o}\} & \text{(Set of Object References)} \\
\mathcal{I} & = & \{id_0, id_1, \ldots\} & \text{(Set of Identifiers)} \\
\mathcal{M} & = & \mathsf{copy} : \mathcal{R} \times \mathcal{I} \rightarrow \mathcal{M} & \text{(Set of Messages)} \\
 & & \mid\ \mathsf{copy\_ack} : \mathcal{R} \times \mathcal{I} \rightarrow \mathcal{M} & \\
 & & \mid\ \mathsf{dirty} : \mathcal{R} \rightarrow \mathcal{M} & \\
 & & \mid\ \mathsf{dirty\_ack} : \mathcal{R} \rightarrow \mathcal{M} & \\
 & & \mid\ \mathsf{clean} : \mathcal{R} \rightarrow \mathcal{M} & \\
 & & \mid\ \mathsf{clean\_ack} : \mathcal{R} \rightarrow \mathcal{M} & \\
\mathcal{D}^t & = & \mathcal{P} \times \mathcal{P} \times \mathcal{I} & \text{(Set of Transient Dirty Entries)} \\
\mathcal{D}^p & = & \mathcal{P} & \text{(Set of Permanent Entries)} \\
\mathcal{B} & = & \mathcal{I} \times \mathcal{P} \times \mathcal{R} & \text{(Set of Blocked Entries)} \\
\mathcal{RS} & = & \{\bot, \mathsf{nil}, \mathsf{OK}, \mathsf{ccit}, \mathsf{ccitnil}\} & \text{(Set of Reference States)} \\
 & & & \\
\mathcal{K} & = & \mathcal{P} \times \mathcal{P} \rightarrow Bag(\mathcal{M}) & \text{(Set of Channels)} \\
\mathcal{DT}^t & = & \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{D}^t & \text{(Set of Transient Dirty Tables)} \\
\mathcal{DT}^p & = & \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{D}^p & \text{(Set of Permanent Dirty Tables)} \\
\mathcal{RT} & = & \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{RS} & \text{(Set of Receive Tables)} \\
\mathcal{BT} & = & \mathcal{P} \times \mathcal{R} \rightarrow \mathbb{P}(\mathcal{B}) & \text{(Set of Blocked Tables)} \\
\mathcal{CPAT} & = & \mathcal{P} \rightarrow \mathbb{P}(\mathcal{B}) & \text{(Set of Copy Ack Tables)} \\
\mathcal{DAT} & = & \mathcal{P} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{R}) & \text{(Set of Dirty Ack Tables)} \\
\mathcal{CLAT} & = & \mathcal{P} \rightarrow \mathbb{P}(\mathcal{P} \times \mathcal{R}) & \text{(Set of Clean Ack Tables)} \\
\mathcal{DCT} & = & \mathcal{P} \rightarrow \mathcal{R} & \text{(Set of Dirty Call Tables)} \\
\mathcal{CCT} & = & \mathcal{P} \rightarrow \mathcal{R} & \text{(Set of Clean Call Tables)} \\
\mathcal{C} & = & \mathcal{DT}^t \times \mathcal{DT}^p \times \mathcal{RT} \times \mathcal{BT} \times \mathcal{CPAT} & \text{(Set of Configurations)} \\
 & & \times\, \mathcal{DAT} \times \mathcal{CLAT} \times \mathcal{DCT} \times \mathcal{CCT} \times \mathcal{K} &
\end{array}
$$

Characteristic variables:

$$
\begin{array}{l}
p \in \mathcal{P},\quad r \in \mathcal{R},\quad m \in \mathcal{M},\quad k \in \mathcal{K},\quad c \in \mathcal{C}, \\
tdirty\_T \in \mathcal{DT}^t,\quad pdirty\_T \in \mathcal{DT}^p,\quad rec\_T \in \mathcal{RT},\quad blocked\_T \in \mathcal{BT}, \\
copy\_ack\_todo\_T \in \mathcal{CPAT},\quad dirty\_ack\_todo\_T \in \mathcal{DAT},\quad clean\_ack\_todo\_T \in \mathcal{CLAT}, \\
dirty\_call\_todo\_T \in \mathcal{DCT},\quad clean\_call\_todo\_T \in \mathcal{CCT}
\end{array}
$$

Initial State:

$$
\begin{array}{rcl}
c_i & = & \langle tdirty\_T_i, pdirty\_T_i, rec\_T_i, blocked\_T_i, copy\_ack\_todo\_T_i, dirty\_ack\_todo\_T_i, \\
 & & \quad clean\_ack\_todo\_T_i, dirty\_call\_todo\_T_i, clean\_call\_todo\_T_i, k_i\rangle
\end{array}
$$

$$
\begin{array}{rcl}
k_i & = & p_1, p_2 \rightarrow \emptyset \\
tdirty\_T_i, pdirty\_T_i, rec\_T_i & = & p, r \rightarrow \bot \\
blocked\_T_i & = & p, r \rightarrow \emptyset \\
copy\_ack\_todo\_T_i & = & p \rightarrow \emptyset \\
dirty\_ack\_todo\_T_i, clean\_ack\_todo\_T_i & = & p \rightarrow \emptyset \\
dirty\_call\_todo\_T_i, clean\_call\_todo\_T_i & = & p \rightarrow \bot
\end{array}
$$

Fig. 8. State Space

—If $table\_T$ is a component of a configuration $\langle \ldots, table\_T, \ldots \rangle$, then the expression $table\_T(a_0, \ldots, a_n) := V$ denotes the configuration $\langle \ldots, table\_T', \ldots \rangle$, where $table\_T'(x_0, \ldots, x_n) = table\_T(x_0, \ldots, x_n)$ if $(x_0, \ldots, x_n) \neq (a_0, \ldots, a_n)$, and $table\_T'(a_0, \ldots, a_n) = V$.

—If k is the set of message channels of a configuration $\langle \ldots, k \rangle$, then the expression $post(p_1, p_2, m)$ denotes the configuration $\langle \ldots, k' \rangle$, with $k'(p_1, p_2) = k(p_1, p_2) \oplus \{m\}$, and $k'(p_i, p_j) = k(p_i, p_j)$, $\forall (p_i, p_j) \neq (p_1, p_2)$[6].

—If k is the set of message channels of a configuration $\langle \ldots, k \rangle$, then the expression $receive(p_1, p_2, m)$ denotes the configuration $\langle \ldots, k' \rangle$, with $k'(p_1, p_2) = k(p_1, p_2) \ominus \{m\}$, and $k'(p_i, p_j) = k(p_i, p_j)$, $\forall (p_i, p_j) \neq (p_1, p_2)$.

The abstract machine is characterised by an initial state and a set of transitions. The initial state appears in Figure 8, and intuitively can be summarised as composed of empty tables and empty message channels. The arrow notation used here defines the initial state of the tables as functions taking one or two arguments and returning a result. Thus $tdirty\_T_i$ takes a $\mathcal{P}$ and an $\mathcal{R}$ argument and returns $\perp$[7]. In the initial configuration, the receive table rec_T_i maps each process, reference pair p, r to $\perp$. Thus, in a manner similar to physical addresses in a real machine, references are never 'created' although they are initially associated with $\perp$.

Rules are the formal way to express permissible transitions of the abstract machine. They are specified in the next section, using the following syntax:

```
  $rule\_name(v_1, v_2, \ldots)$ :
   $condition_1(v_1, v_2, \ldots) \wedge condition_2(v_1, v_2, \ldots) \wedge \ldots$
$\rightarrow$ {
      $pseudo\_statement_1$;
      ...
      $pseudo\_statement_n$;
   }
```

A rule is identified by its name and is parameterised by a number of variables. Conditions can appear to the left-hand side of the arrow: these are guards that must be satisfied in order for the rule to be fireable. The right-hand side denotes the configuration that is reached after transition: its value is the result of applying the configuration transformer obtained by composing all the pseudo-statements to the configuration that satisfied the guard. From a concurrency viewpoint, we assume that the execution of a rule, i.e. verification of its guards and application of its pseudo-statements, is performed atomically, meaning that rule execution can neither be interrupted nor interleaved with the execution of other rules. The design philosophy of Section 3.1 is intended to minimise the number of operations to be performed when rules are fired. Consequently, these rules provide a realistic basis for an implementation design.

[6]We use the operators $\oplus$ and $\ominus$ to denote union and difference on bags.
[7]Or, in λ-calculus notation, $tdirty\_T_i = \lambda pr.\perp$.

In the presentation of the algorithm, we distinguish the guards of a rule from assertion statements. Guards are conditions that must be satisfied for a rule to be fireable. Assertion statements appear as comments in a rule body and indicate that a given property is proved to hold at that point in the execution. When we want to provide the reader with better understanding of the condition in which a rule is fired, we use assertion statements as opposed to guards, otherwise we would have to prove that the extra guards do not induce deadlocks. Furthermore, assertion statements can describe a global state of the distributed system, which does not necessarily have to be computable by the process executing the rule.

3.3.2 *Configuration.* Let us now examine the content of a configuration of the abstract machine. It consists of a set of communication channels and a set of tables, for which we provide some intuitive explanation; their usage will be specified by the transition rules.

We use *dirty-* and *receive-tables* to represent a process's knowledge about the references that it has sent and received. Whenever a process p_1 sends a reference to a remote process p_2 or receives a dirty call from it, p_1 makes an entry in a dirty table. The dirty tables are the explicit representation of reference listing that the algorithm achieves. Symmetrically, the receive-table contains all the references that have been received by and are alive in a process. If a reference r does not exist in a process p, then $rec\_T(p,r) = \perp$, meaning that the reference r is in its pre-existence state in process p. The permissible values in a receive-table are the states of the life cycle ($\perp$, nil, OK, ccit, $\mathsf{ccitnil}$). The life cycle states **2**, **3**, ... of Figure 4 are not represented explicitly as potential values in the receive tables; instead, they are encoded in the form of reference listing in the transient dirty table. Note that this representation neither relies on nor prescribes details of implementation and hence differs from Birrell's original account (which assumed, for example, references would be held on process stacks).

When a reference is received for the first time, the receiver must initiate a dirty call in order to register the reference with its owner before the reference becomes usable by the receiver. During that time, deserialisation activities (and associated remote invocations) must be suspended. Such information is modelled by the *blocked-table.*

All 'outbound' messages are followed up by an acknowledgement. With our design rules, we decouple the receiving of a message from the sending of its acknowledgement so as to ensure short *and explicit* atomic sections during rule execution. The corollary is that it is necessary to introduce states internal to each process that represent those messages remaining to be acknowledged: such states are encoded by the tables $copy\_ack\_todo\_T$, $dirty\_ack\_todo\_T$ and $clean\_ack\_todo\_T$. Similarly, clean and dirty calls are typically made in response to another event in the system. Their scheduling is indicated by entries in the $clean\_call\_todo\_T$ and $dirty\_call\_todo\_T$ tables.

3.3.3 *Transition Rules.* We now study the algorithm's transition rules, which can be found in Figures 9 to 12. For each figure, we first give an overall description of the rules, which we follow with specific details, cross-referencing lines in the figures. Notice that some pseudo-statements are annotated with a *termination* *measure*, denoted by $\pm x$ for some integer x; these measures are used in the proof of the liveness of the algorithm (see Section 4.2).

Rule *make_copy* in Figure 9 models p_1 sending a reference to a process p_2 (supposed to be different from p_1). Such a transition can only occur if the reference is locally reachable by p_1 and if its state in the receive table is OK. The effect is to add a new entry into the transient dirty table, and to send a copy message in the channel between p_1 and p_2.

The receipt of a copy message is modelled by rule *receive_copy*, which is one of the most complex rules of the algorithm as it has potential implications for numerous internal tables of a process. This rule can take place only if there is a copy-message in transit between two processes p_1 and p_2; its intuitive behaviour is better explained by referring to Figure 7 in which the reference's state is shifted to the 'right' of the figure. Once the message has been received by p_2 — expressed by the statement $receive(p_1, p_2, \mathsf{copy}(r, id))$ — the rule suspends the deserialisation of the contents of the message for states that need to complete a dirty call. In these cases, it also schedules such dirty calls, cancels pending clean calls and schedules copy acknowledgements.

(Note 1). Birrell's algorithm assumes RPC-style communications that are composed of a method invocation and the associated return of a result. As we rely on asynchronous message passing, we create a new identifier *id* for each copy of a reference, which allows us to identify the matching acknowledgement message, copy_ack. Such identifiers are also able to distinguish multiple messages sent in parallel between two processes. We note that the identifier *id* must be new to the whole distributed system; URI schemes could be used here, incorporating the name of the process that executes the transition.

(Note 2). Birrell's algorithm prescribes that a reference sent remotely by a remote method invocation is kept locally reachable on a stack frame. We formalise this in an implementation technique independent manner: we assume the existence of a table, defined to be a root of the local collector, which we call the *transient dirty table* and denote *tdirty_T*. Any object referred to by an entry in this table remains locally reachable by the garbage collector. Adopting such a dirty table is beneficial for a number of reasons:

(1) We need not make a distinction between remote method invocation and return of a result. Birrell's algorithm is silent about method returns, which typically coincide with stack deallocation, although references still have to be kept locally reachable. In our algorithm definition, we *always* store a reference in the transient dirty table before it is communicated remotely, whether it is an argument or the result of a remote method call.

(2) In Birrell's algorithm, the dirty set is only present in the memory space of the reference owner, and is meant to contain the list of processes that sent dirty calls to the owner. For the owner, we introduce permanent dirty tables that have a similar role (as discussed in Note 6). Transient dirty tables for all processes contain information about copy messages sent, which we record as a triple: the sender, the receiver and the message identifier. We refer to such a triple as a *transient dirty entry*, as opposed to *permanent dirty entries*, to be introduced later.

```
  $make\_copy(p_1, p_2, r)$ :
   $p_1 \neq p_2 \ \wedge \ rec\_T(p_1, r) = \mathsf{OK}$
$\rightarrow$ {
      $id := new\ Identifier;$                                                    (Note 1)
      $tdirty\_T(p_1, r) := tdirty\_T(p_1, r) \cup \{\langle p_1, p_2, id\rangle\};$                  (Note 2)
      $post(p_1, p_2, \mathsf{copy}(r, id));$
   }

  $receive\_copy(p_1, p_2, r, id)$ :
   $\mathsf{copy}(r, id) \in k(p_1, p_2)$
$\rightarrow$ {
      $receive(p_1, p_2, \mathsf{copy}(r, id));$                                  //–14
      $if\ rec\_T(p_2, r) = \mathsf{nil} \ \vee \ rec\_T(p_2, r) = \mathsf{ccitnil},\ then$
         $blocked\_T(p_2, r) := blocked\_T(p_2, r) \cup \{\langle id, p_1, r\rangle\};$        //+2  (Note 3)
      $elif\ rec\_T(p_2, r) = \perp \ \vee \ rec\_T(p_2, r) = \mathsf{ccit},\ then$
      {
         $if\ rec\_T(p_2, r) = \perp,\ then\ rec\_T(p_2, r) = \mathsf{nil};$                //+1  (Note 3)
         $elif\ rec\_T(p_2, r) = \mathsf{ccit},\ then\ rec\_T(p_2, r) = \mathsf{ccitnil};$          //+1
         $dirty\_call\_todo\_T(p_2) := dirty\_call\_todo\_T(p_2) \cup \{r\};$              //+9
         $blocked\_T(p_2, r) := blocked\_T(p_2, r) \cup \{\langle id, p_1, r\rangle\};$         //+2
      }
      $elif\ rec\_T(p_2, r) = \mathsf{OK},\ then$
      {
         $clean\_call\_todo\_T(p_2) := clean\_call\_todo\_T(p_2) \setminus \{r\};$                (Note 4)
         $copy\_ack\_todo\_T(p_2) := copy\_ack\_todo\_T(p_2) \cup \{\langle id, p_1, r\rangle\};$    //+2
      }
   }
```

Fig. 9. Copy Processing (a)

(Note 3). An incoming new reference r is not immediately usable by programs, since first it has to be properly registered with the reference's owner through a dirty call. During this time, the deserialisation thread has to be suspended, which we indicate by an entry in the *blocked table*, denoted $blocked\_T$, identifying the reference, its sender and the message id. Similarly, the receipt of a reference that is already known to the process but is not currently usable also suspends the deserialisation code.

(Note 4). When we formalise the interaction with the local garbage collector, we shall see that any reference observed to be locally unreachable is scheduled for a cleanup operation; this is represented by an entry in the $clean\_call\_todo\_T$ table, as discussed in Note 9. The normal course of action would be to send a clean message to its owner to clear the owner's permanent dirty entry for that process. Before we send such a message, another copy message with this reference could have been received by this process. In order to avoid successively sending a clean and a dirty message, we introduce an optimisation that consists of cancelling both messages and *resurrecting* the reference. It is safe to do so because the owner still believes that the reference is live, and no clean message has been scheduled yet. More importantly, it is efficient to do so, because it avoids blocking the deserialiser for the duration of the clean and dirty messages and their respective acknowledgements. Hence, we remove the entry in the $clean\_call\_todo\_T$.

Rule *do_copy_ack* in Figure 10 is triggered by the presence of an entry in a $copy\_ack\_todo\_T$ table, which indicates that a copy acknowledgement has been scheduled. The entry is removed from the table and a copy_ack message is posted. Rule *receive_copy_ack* deals with the receipt of a copy_ack message.

```
  $do\_copy\_ack(p_1, p_2, r, id)$ :
   $\langle id, p_2, r\rangle \in copy\_ack\_todo\_T(p_1)$
$\to$ {
      $copy\_ack\_todo\_T(p_1) := copy\_ack\_todo\_T(p_1) \setminus \{\langle id, p_2, r\rangle\}$;      //–2
      $post(p_1, p_2, \mathsf{copy\_ack}(r, id))$;                                          //+1
   }

  $receive\_copy\_ack(p_1, p_2, r, id)$ :
   $\mathsf{copy\_ack}(r, id) \in k(p_1, p_2)$
$\to$ {
      $receive(p_1, p_2, \mathsf{copy\_ack}(r, id))$;                                       //–1
      $tdirty\_T(p_2, r) := tdirty\_T(p_2, r) \setminus \{\langle p_2, p_1, id\rangle\}$;
   }
```

Fig. 10. Copy Processing (b)

Rule *do_dirty_call* in Figure 11 initiates the sending of a dirty call that had been previously scheduled (as indicated by an entry in the $dirty\_call\_todo\_T$ table). The receipt of a dirty call message for a reference r by rule *receive_dirty_call* entails the adding of r to the owner's permanent dirty table; this means that the sender of the dirty call possesses reference r. Rule *do_dirty_ack* sends a dirty call acknowledgement, handled by rule *receive_dirty_ack* (changing the status from a 'lower' state to an 'upper' state in Figure 6).

(Note 5). We have not made any assumption of the order in which messages are delivered. In particular, sending a dirty message when a clean message is already in transit could result in the dirty message being processed first. The early receipt of a dirty message would have no effect (as a permanent dirty entry is already present), but the processing of the delayed clean message would clear the dirty entry, which would result in an incorrect situation, as the reference may become locally unreachable by the owner. Consequently, we postpone the sending of dirty messages when the reference state is ccitnil until we have received an acknowledgement that the clean message has been processed.

(Note 6). When a dirty message is received by a reference owner, a new entry is added to its permanent dirty table to indicate that the owner believes that the message sender owns a live reference. The entry consists of the process identifier only, and is called a *permanent dirty entry*; it should be distinguished from the entry (sender, receiver, message identifier) in the transient dirty table described in Note 2. The permanent dirty table must also be defined as a root of the local collector. We see again here that this algorithm does indeed *list* references rather than *count* them. Even though we use a set notation, in practice the permanent dirty table does not need to be a set since the algorithm never receives an incoming dirty message from a process p if there is already an entry for p in the permanent dirty table.

(Note 7). When a dirty_ack message is received, entries in the blocked table (which contain the source of a copy-message, the reference and an identifier) can be added to the set of copy_ack messages to be sent. It is important that the sending of copy_ack is postponed until the dirty_ack has been received, otherwise a race condition reminiscent of naive reference counting may occur: the reference could become garbage on the sender before its receiver had time to register it properly with the owner.

(Note 8). Once the reference has been registered with its owner, all deserialisation threads that were suspended may be resumed.

Figure 12 is concerned with the handling of clean calls and their acknowledgements. Rule *finalize* describes the action to perform on a reference that is no longer reachable locally; the algorithm simply schedules a clean call for the reference. The clean call and its receipt are respectively handled by rules *do_clean_call* and *receive_clean*, the second of which removes the reference from the owner's permanent dirty table: this point marks, for the owner, the end of the reference's life. This is followed by a clean acknowledgement *do_clean_ack* and its processing *receive_clean_ack*. The latter rule moves an 'upper' state in Figure 6 to a 'lower' state.

```
  $do\_dirty\_call(p, r)$ :
   $r \in dirty\_call\_todo\_T(p) \ \wedge \ rec\_T(p, r) \neq$ ccitnil                    (Note 5)
$\rightarrow$ {
      $dirty\_call\_todo\_T(p) := dirty\_call\_todo\_T(p) \setminus \{r\}$;              //–9
      $post(p, owner(r), \mathsf{dirty}(r))$;                                   //+8
   }

  $receive\_dirty(p_1, p_2, r)$ :
   $p_2 = owner(r) \ \wedge \ \mathsf{dirty}(r) \in k(p_1, p_2)$
$\rightarrow$ {
      $receive(p_1, p_2, \mathsf{dirty}(r))$;                                  //–8
      $pdirty\_T(p_2, r) := pdirty\_T(p_2, r) \cup \{p_1\}$;                          (Note 6)
      $dirty\_ack\_todo\_T(p_2) := dirty\_ack\_todo\_T(p_2) \cup \{\langle p_1, r\rangle\}$;      //+7
   }

  $do\_dirty\_ack(p_1, p_2, r)$ :
   $\langle p_2, r\rangle \in dirty\_ack\_todo\_T(p_1)$
$\rightarrow$ {
      $dirty\_ack\_todo\_T(p_1) := dirty\_ack\_todo\_T(p_1) \setminus \{\langle p_2, r\rangle\}$;      //–7
      $post(p_1, p_2, \mathsf{dirty\_ack}(r))$;                                //+6
   }

  $receive\_dirty\_ack(p_1, p_2, r)$ :
   $\mathsf{dirty\_ack}(r) \in k(p_1, p_2)$
$\rightarrow$ {
      $receive(p_1, p_2, \mathsf{dirty\_ack}(r))$;                              //–6
      $copy\_ack\_todo\_T(p_2) := copy\_ack\_todo\_T(p_2) \cup blocked\_T(p_2, r)$; //+X (Note 7)
      // Deserialisation code to be resumed
      // for each entry in $blocked\_T(p_2, r)$                               (Note 8)
      $blocked\_T(p_2, r) := \emptyset$;                                     //–X
      $rec\_T(p_2, r) :=$ OK;                                            //+5
   }
```

Fig. 11. Dirty Processing

```
  $finalize(p, r)$ :
  $\neg locallyLive(p, r) \ \wedge \ rec\_T(p, r) = \mathsf{OK} \ \wedge \ p \neq owner(r)$
  $\wedge \ r \notin clean\_call\_todo\_T(p)$
$\rightarrow$ {
    $clean\_call\_todo\_T(p) := clean\_call\_todo\_T(p) \cup \{r\}$;                    (Note 9)
  }

  $do\_clean\_call(p, r)$ :
  $r \in clean\_call\_todo\_T(p)$
$\rightarrow$ {
    $clean\_call\_todo\_T(p) := clean\_call\_todo\_T(p) \setminus \{r\}$;
    $rec\_T(p, r) := \mathsf{ccit}$; //assert: was $rec\_T(p_1, r) = \mathsf{OK}$          //–4  (Note 10)
    $post(p, owner(r), \mathsf{clean}(r))$;                                   //+3
  }

  $receive\_clean(p_1, p_2, r)$ :
  $p_2 = owner(r) \ \wedge \ \mathsf{clean}(r) \in k(p_1, p_2)$
$\rightarrow$ {
    $receive(p_1, p_2, \mathsf{clean}(r))$;                                    //–3
    $pdirty\_T(p_2, r) := pdirty\_T(p_2, r) \setminus \{p_1\}$;
    $clean\_ack\_todo\_T(p_2) := clean\_ack\_todo\_T(p_2) \cup \{\langle p_1, r\rangle\}$;      //+2
  }

  $do\_clean\_ack(p_1, p_2, r)$ :
  $\langle p_2, r\rangle \in clean\_ack\_todo\_T(p_1)$
$\rightarrow$ {
    $clean\_ack\_todo\_T(p_1) := clean\_ack\_todo\_T(p_1) \setminus \{\langle p_2, r\rangle\}$;      //–2
    $post(p_1, p_2, \mathsf{clean\_ack}(r))$;                                //+1
  }

  $receive\_clean\_ack(p_1, p_2, r)$ :
  $\mathsf{clean\_ack}(r) \in k(p_1, p_2)$
$\rightarrow$ {
    $receive(p_1, p_2, \mathsf{clean\_ack}(r))$;                              //–1
    if $rec\_T(p_2, r) = \mathsf{ccitnil}$, then                                 (Note 11)
     $rec\_T(p_2, r) := \mathsf{nil}$;                                        //–1
    else //assert: $rec\_T(p_2, r) = \mathsf{ccit}$
     $rec\_T(p_2, r) := \bot$;                                          //–1
  }
```

Fig. 12. Clean Processing

(Note 9). Rule *finalize* defines the interaction with the system in which the reference listing algorithm is embedded (for example, a local garbage collector). We maintain asynchrony between the environment and the distributed reference listing algorithm. Once a reference is detected to be locally unreachable, a flag is set in the form of an entry in the $clean\_call\_todo\_T$ table. The state of the reference, as represented by the receive table rec_T, is unchanged. This allows the state to revert before the cleaning sequence is started, as described in Note 4. In order to prevent the repeated firing of rule *finalize*, we have added a guard requiring the reference to be absent from $clean\_call\_todo\_T(p)$.

(Note 10). After a clean call has been scheduled, a clean message is sent to the owner, and the reference state is changed to ccit, to indicate that a 'clean call is in transit'.

(Note 11). A clean call can be received when the reference is in one of two states: ccit or ccitnil. In the former case, the receiver has no useful reference (see Figure 7) and so the state of the reference reverts to state $\perp$. In the latter case, a new, usable copy of the reference has been received subsequently and so the state changes to nil: transition *do_dirty_call* is now fireable and a new life cycle is initiated.

This concludes our presentation of Birrell's algorithm. Our two descriptions, graphical and formal, are complementary. The graphical presentation exposes the complete life cycle of references in an intuitive manner, whereas the formal semantics presents, in an unambiguous fashion, the precise states that must be maintained by processes taking part in the algorithm. Interestingly, the formal notation is suitable both for establishing the correctness of the algorithm and as a basis for an implementation. We now discuss the correctness of the algorithm.

4. PROPERTIES AND PROOFS

The correctness of an algorithm typically has two different facets: *safety* guarantees that nothing bad can happen and *liveness* guarantees that something good will eventually happen. In the specific context of this distributed reference listing algorithm, such properties can be expressed as follows. Safety guarantees that if a remote reference to a resource exists, then one of its owner's dirty tables will contain an entry (permanent or transient) for the reference, which prevents the resource from being recycled by the owner. Liveness guarantees that if all references to a resource are deleted, all its dirty entries (permanent and transient) will eventually be removed by its owner, thus allowing the resource to be claimed at some later point by a local garbage collector. The purpose of this section is to establish both properties.

4.1 Proof of Safety

The proofs that we derive in this paper are based on invariants that we prove to hold for all configurations of the abstract machine. Given an arbitrary valid configuration, the proofs typically proceed by induction on the length of the transitions that lead to the configuration, and by a case analysis on the kind of transitions. For the sake of conciseness, we present only those transitions that have an effect on subterms of the property we are trying to establish; the others have no effect on the property. The base case of the induction also requires us to derive the property in the initial configuration.

This kind of proof has some advantages. It is systematic and can easily be encoded in a mechanical theorem prover (as illustrated by other proofs [Moreau and Duprat 2001; Moreau 2001a; 2002] successfully encoded in Coq by the first author). It is less prone to error than the kind of temporal reasoning used in Birrell's proof. The difficulty of a proof lies in establishing its fine details (such as some of the mutually exclusive cases we consider in Lemma 3), and their formulation becomes complicated in a formalism for temporal-based reasoning.

Outline. Two key invariants are established. First, in INVARIANT 1, we define a condition in terms of messages in transit and process states that specifies when a transient dirty entry appears in the dirty table. Second, in INVARIANT 2, we derive an equation that links the existence of a permanent dirty entry to messages in transit and process states. Third, using these lemmas, we establish that suitable dirty entries are maintained by the algorithm for all types of references found in the system, which ensures the safety property. □

First, let us prove some simple properties. In Lemma 1, we show that whenever a reference is in state ccitnil, a dirty call has been scheduled for the reference.

LEMMA 1. *For any process p and reference r, if $rec\_T(p,r) = \mathsf{ccitnil}$, then $r \in dirty\_call\_todo\_T(p)$.* □

PROOF. This implication holds in the initial configuration since $rec\_T(p,r) = \bot$. The receive table is set to ccitnil only by transition *receive_copy*, which also adds an entry to the table *dirty_call_todo_T*. Entries are only removed from *dirty_call_todo_T* by transition *do_dirty_call*, which fires only if the receive table is not in state ccitnil. □

Then, in Lemma 2, we show that a reference must be in the OK state for an entry in *clean_call_todo_T* to exist.

LEMMA 2. *For any configuration, for any process p and for any reference r, if $r \in clean\_call\_todo\_T(p)$, then $rec\_T(p,r) = \mathsf{OK}$.* □

PROOF. We proceed by induction on the length of transitions that lead to the configuration, and by case analysis on the kind of transitions. The property is initially true since $clean\_call\_todo\_T(p)$ is empty in the initial configuration. Only three rules modify the table $clean\_call\_todo\_T(p)$.

$receive\_copy(p_1, p_2, r, id)$:
: This transition removes r from $clean\_call\_todo\_T(p)$ when $rec\_T(p,r) = \mathsf{OK}$, which validates the consequent whatever the value of the antecedent.

$finalize(p,r)$:
: This can be fired if $rec\_T(p,r) = \mathsf{OK}$; it adds r to the table $clean\_call\_todo\_T(p)$. Therefore, Lemma 2 is satisfied.

$do\_clean\_call(p,r)$:

This removes r from the table $clean\_call\_todo\_T(p)$. After transition, the antecedent is false and therefore the Lemma is trivially satisfied.

□

INVARIANT 1 is the first major invariant we derive for the algorithm. We show that a transient entry in a dirty table can be found if and only if we can find related copy or copy_ack messages in transit, or related entries in *blocked_T* or *copy_ack_todo_T* tables; these conditions are shown to be mutually exclusive.

LEMMA 3 INVARIANT 1. *For any processes p_1 and p_2, for any reference r, for any identifier id, and for any configuration, the following equality holds:*

$$\langle p_1, p_2, id\rangle \in tdirty\_T(p_1, r) = \bigvee \begin{cases} \mathsf{copy}(r, id) \in k(p_1, p_2) \\ \langle id, p_1, r\rangle \in blocked\_T(p_2, r) \\ \mathsf{copy\_ack}(r, id) \in k(p_2, p_1) \\ \langle id, p_1, r\rangle \in copy\_ack\_todo\_T(p_2). \end{cases}$$

Additionally, the four terms:

$$\begin{array}{l} \mathsf{copy}(r, id) \in k(p_1, p_2) \\ \langle id, p_1, r\rangle \in blocked\_T(p_2, r) \\ \mathsf{copy\_ack}(r, id) \in k(p_2, p_1) \\ \langle id, p_1, r\rangle \in copy\_ack\_todo\_T(p_2). \end{array}$$

are mutually exclusive. □

PROOF. The equality holds in the initial configuration since there is no message in transit and all tables are empty. We now consider only those transitions that may have an effect on terms in the equality.

$make\_copy(p_1, p_2, r)$:
If id is the new identifier referred to by transition $make\_copy(p_1, p_2, r)$, then the equality holds after the transition, since there is a copy message in the channel between p_1 and p_2, and the triple $\langle p_1, p_2, id\rangle$ was added to $tdirty\_T(p_1, r)$. At this stage, the identifier id is unknown in process p_2, and $\mathsf{copy}(r, id) \in k(p_1, p_2)$ is the only term that holds among the four.

$receive\_copy(p_1, p_2, r, id)$:
The equality and the mutual exclusivity of terms are preserved by the transition, since the copy message in the channel between p_1 and p_2 (as specified in the rule premise) is replaced by the triple $\langle id, p_1, r\rangle$ either in $blocked\_T(p_2, r)$ if $rec\_T(p_2, r) \neq \mathsf{OK}$, or in $copy\_ack\_todo\_T(p_2)$ if $rec\_T(p_2, r) = \mathsf{OK}$. The equality is preserved because $tdirty\_T(p_1, r)$ is unchanged, and the terms remain mutually exclusive.

$do\_copy\_ack(p_2, p_1, r, id)$:
The equality is preserved by the transition, since the triple $\langle id, p_1, r\rangle$ in the $copy\_ack\_todo\_T$ table is replaced by a $\mathsf{copy\_ack}(r, id)$ message in the channel from p_2 to p_1.

$receive\_copy\_ack(p_2, p_1, r, id)$:
The equality is preserved by the transition, since the triple $\langle p_1, p_2, id\rangle$ is removed from $tdirty\_T(p_1, r)$ when the copy_ack is removed from the channel from p_2 to p_1. We note here that we rely on the fact that the four terms are mutually exclusive: if there is a copy_ack message in transit before transition, then there is no entry in the $blocked\_T$ or $copy\_ack\_todo\_T$ tables, nor is there a copy message in transit.

$receive\_dirty\_ack(p, p_2, r)$:
The equality is preserved by the transition, since any triple $\langle id, p_1, r\rangle$ is removed from $blocked\_T(p_2, r)$ and added to $copy\_ack\_todo\_T(p_2)$. The terms remain mutually exclusive.

□

The states ccit and ccitnil are related to the presence of clean messages in transit, as specified in the following Lemma.

LEMMA 4. *For any processes p_1 and p_2, for any reference r, with $p_2 = owner(r)$, and for any configuration, the following implication holds:*

$$\begin{array}{l} \textit{If } \mathsf{clean}(r) \in k(p_1, p_2) \\ \vee\ \langle p_1, r\rangle \in clean\_ack\_todo\_T(p_2) \\ \vee\ \mathsf{clean\_ack}(r) \in k(p_2, p_1), \\ \textit{then } rec\_T(p_1, r) = \mathsf{ccit} \vee\ rec\_T(p_1, r) = \mathsf{ccitnil}. \end{array}$$

Additionally, the three states:

$$\begin{array}{l} \mathsf{clean}(r) \in k(p_1, p_2) \\ \langle p_1, r\rangle \in clean\_ack\_todo\_T(p_2) \\ \mathsf{clean\_ack}(r) \in k(p_2, p_1) \end{array}$$

are mutually exclusive. □

PROOF. The implication holds in the initial configuration since there is no message in transit and all tables are empty: the antecedent is false, which trivially validates the implication. We now consider only those transitions that may have an effect on terms in the implication.

$receive\_copy(p, p_1, r, id)$:
The implication is preserved by the transition, since it may change the state of $rec\_T(p_1, r)$ from ccit to ccitnil.

$do\_clean\_call(p_1, r)$:
The implication holds after the transition, because it adds a clean message between p_1 and p_2 (with p_2 equal to $owner(r)$), and sets $rec\_T(p_1, r)$ to ccit. Before transition, $rec\_T(p_1, r) = \mathsf{OK}$, by Lemma 2. Therefore, the terms $\mathsf{clean}(r) \in k(p_1, p_2)$, $\langle p_1, r\rangle \in clean\_ack\_todo\_T(p_2)$ and $\mathsf{clean\_ack}(r) \in k(p_2, p_1)$ must be false, otherwise $rec\_T(p_1, r)$ would be equal to ccit or ccitnil. Thus, they remain exclusive.

$receive\_clean\_call(p_1, p_2, r)$:
The implication is preserved by the transition, because it removes a clean message between p_1 and p_2, and sets the $clean\_ack\_todo\_T(p_2)$ to $\langle p_1, r\rangle$.

$do\_clean\_ack(p_2, p_1, r)$:

The implication is preserved by the transition, because it adds a $\mathsf{clean\_ack}$ message between p_2 and p_1, and removes the entry $\langle p_1, r\rangle$ from $clean\_ack\_todo\_T(p_2)$.

$receive\_clean\_ack(p_2, p_1, r)$:

The implication is preserved by the transition, because it removes a $\mathsf{clean\_ack}$ message between p_2 and p_1, and sets $rec\_T(p_1, r)$ to nil or $\bot$; we rely here on the mutual exclusivity property of the conditions to show that there is no clean message in transit, nor $\mathsf{clean\_ack}$ message to be posted.

□

States nil and $\mathsf{ccitnil}$ indicate that some reference has been received and will not be usable until it has been registered with its owner by a dirty call, as specified by the following Lemma.

Lemma 5. *For any processes p_1 and p_2, for any reference r with $p_2 = owner(r)$, and for any configuration, the following implications hold:*

$$\begin{array}{rl} \textit{If} & r \in dirty\_call\_todo\_T(p_1), \\ \textit{then} & rec\_T(p_1, r) = \mathsf{nil} \vee rec\_T(p_1, r) = \mathsf{ccitnil}. \end{array} \qquad (5.a)$$

$$\begin{array}{rl} \textit{If} & \mathsf{dirty}(r) \in k(p_1, p_2) \\ \vee & \langle p_1, r\rangle \in dirty\_ack\_todo\_T(p_2) \\ \vee & \mathsf{dirty\_ack}(r) \in k(p_2, p_1), \\ \textit{then} & rec\_T(p_1, r) = \mathsf{nil}. \end{array} \qquad (5.b)$$

Additionally, the following four states:

$$\begin{array}{l} r \in dirty\_call\_todo\_T(p_1) \\ \mathsf{dirty}(r) \in k(p_1, p_2) \\ \langle p_1, r\rangle \in dirty\_ack\_todo\_T(p_2) \\ \mathsf{dirty\_ack}(r) \in k(p_2, p_1) \end{array} \qquad (5.c)$$

are mutually exclusive.

□

Proof. The statements hold in the initial configuration since all tables are empty and no message is transit.

$receive\_copy(p, p_1, r, id)$ for some p:

The implications are preserved by the transition, because when it adds an entry in $dirty\_call\_todo\_T(p_1)$, it also changes $rec\_T(p_1, r)$ to nil or $\mathsf{ccitnil}$. Additionally, if any of the terms of (5.c) held before transition, then $rec\_T(p_1, r)$ is equal to nil or $\mathsf{ccitnil}$ by (5.a) and (5.b); they do not change value during the transition. Therefore, they remain mutually exclusive.

$do\_dirty\_call(p_1, r)$:

The statements are preserved by the transition, because when it removes an entry from $dirty\_call\_todo\_T(p_1)$, it adds a dirty message between p_1 and p_2. This may falsify the antecedent of (5.a), which therefore trivially validates the implication; it also preserves the mutual exclusivity of (5.c). As (5.a) is true before transition, and do_dirty_call can only fire if $rec\_T(p, r) \neq \mathsf{ccitnil}$, (5.b) is also satisfied.

$receive\_dirty\_ack(p_2, p_1, r)$:
The statements (5.a) and (5.b) are valid after the transition. Indeed, if there is a message $\mathsf{dirty\_ack}(r) \in k(p_2, p_1)$ before transition then, by the mutual exclusivity property, r cannot be in $dirty\_call\_todo\_T(p_1)$ and there is no $\mathsf{dirty\_ack}$ in $k(p_2, p_1)$. This implies that the antecedents of (5.a) and (5.b) are false, and therefore they are trivially satisfied after transition; moreover, (5.c) is also preserved.

$do\_clean\_call(p_1, r)$:
The statements are preserved by the transition. Indeed, if the rule is fired, then $rec\_T(p_1, r) = \mathsf{OK}$ by Lemma 2. If (5.a) holds before transition, then $r \notin dirty\_call\_todo\_T(p)$, which also remains true after transition do_clean_call. Similarly, the antecedent of (5.b) is false before transition and remains false afterwards. None of states of (5.c) is changed by this transition.

$receive\_clean\_ack(p_2, p_1, r)$:
The statements are preserved by the transition. Indeed, it changes $rec\_T(p_1, r)$ from $\mathsf{ccitnil}$ to nil, which satisfies implication (5.a) and preserves (5.b). It also changes $rec\_T(p_1, r)$ from OK to $\perp$, which means that the antecedents of (5.a) and (5.b) are false before transition and remain so after transition. None of states of (5.c) is changed by this transition.

□

Invariant 2 is the second major invariant that we will use to establish the algorithm's correctness. Whilst Invariant 1 focuses on transient entries in dirty tables, Invariant 2 is concerned with permanent entries.

Informally, Lemma 6 states that if an owner has a permanent dirty entry for a reference, or a dirty call is scheduled or alternatively in transit, then either the state of the reference is one of OK, nil or $\mathsf{ccitnil}$, or a clean call has been sent (and vice-versa).

Lemma 6 Invariant 2. *For any processes p_1 and p_2, for any reference r with $p_2 = owner(r)$, and for any configuration, the following equality holds:*

$$\begin{gathered} p_1 \in pdirty\_T(p_2, r) \;\vee\; \mathsf{dirty}(r) \in k(p_1, p_2) \;\vee\; r \in dirty\_call\_todo\_T(p_1) \\ = \\ \mathsf{clean}(r) \in k(p_1, p_2) \vee rec\_T(p_1, r) = \mathsf{OK} \vee rec\_T(p_1, r) = \mathsf{nil} \vee rec\_T(p_1, r) = \mathsf{ccitnil} \end{gathered}$$

□

Proof. First, we note that $\mathsf{dirty}(r) \in k(p_1, p_2)$ and $r \in dirty\_call\_todo\_T(p_1)$ are mutually exclusive by Lemma 5.c. The equality holds in the initial configuration since there is no message in transit and all tables are empty. We now consider the transitions that may have an effect on terms in the equality.

$receive\_copy(p_1, p_2, r, id)$:
The equality is preserved by the transition.

—If $rec\_T(p_1, r) = \mathsf{ccit}$ or $rec\_T(p_1, r) = \bot$ before transition, then after transition $\mathsf{dirty}(r) \notin k(p_1, p_2)$ and $r \notin dirty\_call\_todo\_T(p_1)$ by Lemma 5.b. Before transition, both

$$\mathsf{dirty}(r) \in k(p_1, p_2) \vee r \in dirty\_call\_todo\_T(p_1) \tag{1}$$

and

$$rec\_T(p_1, r) = \mathsf{OK} \vee rec\_T(p_1, r) = \mathsf{nil} \vee rec\_T(p_1, r) = \mathsf{ccitnil} \tag{2}$$

are false; because r is added to $dirty\_call\_todo\_T(p_1)$ and $rec\_T(p_1, r)$ is set to nil or $\mathsf{ccitnil}$, both become true after transition. The equality is therefore satisfied.

—If $rec\_T(p_1, r) = \mathsf{nil}$, $rec\_T(p_1, r) = \mathsf{ccitnil}$ or $rec\_T(p_1, r) = \mathsf{OK}$, then the equality is preserved because no term is changed.

do_dirty_call(p_1, r):

The equality is preserved by the transition, since r is removed from the table $dirty\_call\_todo\_T(p_1)$ and a dirty message is added between p_1 and $owner(r)$.

receive_dirty(p_1, p_2, r):

The equality is preserved by the transition, since p_1 is added to $pdirty\_T(p_2, r)$, and a dirty message is removed from the link between p_1 and p_2.

receive_dirty_ack(p_1, p_2, r):

The equality is preserved by the transition. Indeed, before transition, there is a $\mathsf{dirty\_ack}$ between p_2 and p_1, therefore $rec\_T(p_1, r) = \mathsf{nil}$ by Lemma 5.b. After the transition, $rec\_T(p_1, r)$ is changed to OK, which preserves the equality.

do_clean_call(p_1, r):

The equality is preserved by the transition. Indeed, $rec\_T(p_1, r) = \mathsf{OK}$ before transition, by Lemma 2. After transition, $rec\_T(p_1, r)$ is changed to ccit, but a clean message is added to the channel between p_1 and its $owner(r)$.

receive_clean_call(p_1, p_2, r):

The equality is preserved by the transition. After transition, p_1 is removed from $pdirty\_T(p_2, r)$, and the clean message is removed from the channel between p_1 and p_2. So $p_1 \in pdirty\_T(p_2, r)$ and $\mathsf{clean}(r) \in k(p_1, p_2)$ become false after transition. Let us examine the remaining two terms:

$$\mathsf{dirty}(r) \in k(p_1, p_2) \vee r \in dirty\_call\_todo\_T(p_1) \tag{3}$$

and

$$rec\_T(p_1, r) = \mathsf{OK} \vee rec\_T(p_1, r) = \mathsf{nil} \vee rec\_T(p_1, r) = \mathsf{ccitnil}. \tag{4}$$

If there was a clean message between p_1 and p_2, then by Lemma 4:

$$rec\_T(p_1, r) = \mathsf{ccit} \vee \ rec\_T(p_1, r) = \mathsf{ccitnil} \tag{5}$$

Let us analyse the terms in (3):

—If $r \in dirty\_call\_todo\_T(p_1)$, then $rec\_T(p_1, r) = \mathsf{nil} \vee \ rec\_T(p_1, r) = \mathsf{ccitnil}$ by Lemma 5.a. Therefore, combined with (5), both (3) and (4) are true.
If $r \notin dirty\_call\_todo\_T(p_1)$, then $rec\_T(p_1, r) \neq \mathsf{ccitnil}$, by Lemma 1; consequently, $rec\_T(p_1, r) = \mathsf{ccit}$ by (5).

—If $\mathsf{dirty}(r) \in k(p_1, p_2)$, then $rec\_T(p_1, r) = \mathsf{nil}$ by Lemma 5.b, which contradicts the fact that we just derived that it was equal to ccit.
If $\mathsf{dirty}(r) \notin k(p_1, p_2)$, then both (3) and (4) are false.

$receive\_clean\_ack(p_1, p_2, r)$:
$rec\_T(p_2, r)$ is changed from $\mathsf{ccitnil}$ to nil, or from ccit to $\perp$, leaving the equation unchanged.

□

The following lemma links a transient entry in a dirty table to an OK state in the receive table of that process.

Lemma 7. *For any processes p_1 and p_2, for any reference r, for any identifier id and for any configuration, the following implication holds:*

$$\begin{aligned} &\textit{If } \langle p_1, p_2, id\rangle \in tdirty\_T(p_1, r),\\ &\textit{then } rec\_T(p_1, r) = \mathsf{OK}. \end{aligned}$$

□

Proof. In the initial configuration, transient dirty tables are empty and the implication trivially holds. We consider the four rules that add (resp. remove) entries to (resp. from) transient dirty tables and that modify the content of receive tables to, or from, OK.

$make\_copy(p_1, p_2, r)$:
make_copy adds a transient entry $\langle p_1, p_2, id\rangle$, and its guard ensures that the receive-table is in the OK state.

$receive\_copy\_ack(p_2, p_1, r, id)$:
Rule *receive_copy_ack* removes the entry from the transient dirty table, and therefore trivially satisfies Lemma 7.

$receive\_dirty\_ack(p_1, p_2, r)$:
If Lemma 7 held before transition *receive_dirty_ack*, it also holds after transition since the dirty tables are unchanged and the receive table is set to OK.

$do\_clean\_call(p_1, r)$:
As the dirty tables are a root for the local GC, rule *finalize* cannot be fired, and hence $rec\_T(p_1, r)$ will not be changed by *do_clean_call*.

□

The following lemma establishes a relationship between the presence of a *not yet* usable reference and the existence of a process containing an entry in its blocked-table.

Lemma 8. *For any process p_1, for any reference r, and for any configuration, the following implication holds:*

$$\begin{aligned} &\textit{If } rec\_T(p_1, r) = \mathsf{nil} \ \vee\ rec\_T(p_1, r) = \mathsf{ccitnil}\\ &\textit{and if } \mathsf{dirty}(r) \in k(p_1, owner(r)) \ \vee\ r \in dirty\_call\_todo\_T(p_1),\\ &\textit{then there exists } p_2 \textit{ and } id \textit{ such that } \langle id, p_2, r\rangle \in blocked\_T(p_1, r). \end{aligned}$$

□

Proof. We observe that $\mathsf{dirty}(r) \in k(p_1, owner(r))$ and $r \in dirty\_call\_todo\_T(p_1)$ are mutually exclusive (Lemma 5.c). The implication holds in the initial configuration since there is no message in transit and all tables are empty. We now consider the transitions that may have an effect on terms in the implication:

$receive\_copy(p_1, p_2, r, id)$:
: A new entry $\langle id, p_2, r\rangle$ is added to $blocked\_T(p_1, r)$.

$do\_dirty\_call(p_1, r)$:
: Transition $do\_dirty\_call(p_1, r)$ replaces an entry in the table $dirty\_call\_todo\_T(p_1)$ with a dirty call, $\mathsf{dirty}(r) \in k(p_1, owner(r))$. Therefore, the implication is preserved.

$receive\_dirty(p_1, p_2, r)$:
: Transition $receive\_dirty(p_1, p_2, r)$ removes a dirty message between p_1 and p_2; therefore, the second antecedent becomes false, which trivially validates the implication after transition.

$receive\_dirty\_ack(p_2, p_1, r)$:
: Transition $receive\_dirty\_ack(p_1, p_2, r)$ clears table $blocked\_T(p_1, r)$, but also sets $rec\_T(p_1, r)$ to OK, which trivially validates the implication after transition.

□

Three kinds of references can be found in the distributed system: usable references, references in transit and references not yet usable. We successively establish the safety property for each of them.

Lemma 9 Safety 1: Usable Reference. *For any processes p_1 and p_2, for any reference r with $p_2 = owner(r)$ and $p_1 \neq p_2$, and for any configuration, the following implication holds:*

$$\begin{array}{l} \textit{If } rec\_T(p_1, r) = \mathsf{OK}, \\ \textit{then } p_1 \in pdirty\_T(p_2, r). \end{array}$$

□

Proof. We proceed with the following reasoning. This implication holds trivially in the initial configuration since $rec\_T(p_1, r) = \perp$. If $rec\_T(p_1, r) = \mathsf{OK}$, then by Lemma 4, $\mathsf{clean}(r) \notin k(p_1, p_2)$. Similarly, by Lemma 5.b, $\mathsf{dirty}(r) \notin k(p_1, p_2)$ and, by Lemma 5.a, $r \notin dirty\_call\_todo\_T(p_1)$. Consequently, from Lemma 6, we derive that $p_1 \in pdirty\_T(p_2, r)$. □

Lemma 10 Safety 2: Reference in Transit. *For any processes p_1, p_2, for any reference r, for any identifier id and for any configuration, the following implication holds:*

$$\begin{array}{l} \textit{If } \mathsf{copy}(r, id) \in k(p_1, p_2), \\ \textit{then } p_1 \in pdirty\_T(owner(r), r), \ \textit{if } p_1 \neq owner(r) \\ \quad \textit{or } \langle owner(r), p_2, id\rangle \in tdirty\_T(owner(r), r), \ \textit{if } p_1 = owner(r) \end{array}$$

□

Proof. We proceed with the following reasoning. This implication holds trivially in the initial configuration since k is empty. If $\mathsf{copy}(r, id) \in k(p_1, p_2)$, then $\langle p_1, p_2, id\rangle \in pdirty\_T(p_1, r)$ by Lemma 3, which proves the lemma when $p_1 = owner(r)$. Otherwise, $p_1 \neq owner(r)$ and $\langle p_1, p_2, id\rangle \in tdirty\_T(p_1, r)$ implies that $rec\_T(p_1, r) = \mathsf{OK}$ by Lemma 7. Therefore, $p_1 \in pdirty\_T(owner(r), r)$ by Lemma 9. □

Lemma 11 Safety 3: Unusable Reference. *For any process p_1, for any reference r and for any configuration, the following implication holds:*

$$\begin{aligned}&\textit{If } rec\_T(p_1, r) = \mathsf{nil} \ \vee\ rec\_T(p_1, r) = \mathsf{ccitnil},\\ &\textit{then there exists } p \textit{ such that } p \in pdirty\_T(owner(r), r)\\ &\quad\textit{or there exist } p, id \textit{ such that } \langle owner(r), p, id\rangle \in tdirty\_T(owner(r), r).\end{aligned}$$

□

Proof. The implication holds in the initial configuration since $rec\_T(p_1, r) = \perp$. Otherwise, if $rec\_T(p_1, r) = \mathsf{nil} \ \vee\ rec\_T(p_1, r) = \mathsf{ccitnil}$, then:

$$p_1 \in pdirty\_T(p_2, r) \ \vee\ (\mathsf{dirty}(r) \in k(p_1, p_2) \ \vee\ r \in dirty\_call\_todo\_T(p_1))$$

holds by Lemma 6. One of the two terms is true:

—$p_1 \in pdirty\_T(p_2, r)$: Only *receive_dirty* adds a permanent entry to $pdirty\_T(p_2, r)$, and in this case $p_2 = owner(r)$. This proves Lemma 11 with $p = p_1$.

—$\mathsf{dirty}(r) \in k(p_1, p_2) \ \vee\ r \in dirty\_call\_todo\_T(p_1)$: Only *do_dirty_call* posts dirty messages and these are always to the owner. Thus, $p_2 = owner(r)$. By Lemma 8, there exists p and id such that $\langle id, p, r\rangle \in blocked\_T(p_1, r)$. By Lemma 3, $\langle p, p_1, id\rangle \in tdirty\_T(p, r)$ is true. If $p = owner(r)$, Lemma 11 is satisfied. Otherwise, if $\langle p, p_1, id\rangle \in tdirty\_T(p, r)$, then $rec\_T(p, r) = \mathsf{OK}$ by Lemma 7. Therefore, $p \in pdirty\_T(owner(r), r)$ by Lemma 9, which proves Lemma 11.

□

Finally, we can establish the safety of Birrell's algorithm. A distributed garbage collection algorithm is safe if the collector cannot reclaim live objects, i.e. those to which there exists a remote reference, including any reference contained in copy messages in transit. For Birrell's algorithm, the requirement is to ensure that the entry for a remote reference in the owner's dirty tables cannot be empty if another process holds a potentially usable copy of that reference or if there is a message in transit that contains a copy of that reference.

Definition 12 Birrell's Safety Requirement. *The safety requirement for Birrell's algorithm is that for all references r, and for all processes p_1 and p_2 and all identifiers id,*

$$\begin{aligned}&\textit{If } rec\_T(p_1, r) = \mathsf{OK} \ \vee\ rec\_T(p_1, r) = \mathsf{nil} \ \vee\ rec\_T(p_1, r) = \mathsf{ccitnil}\\ &\quad \vee\ \mathsf{copy}(r, id) \in k(p_1, p_2),\\ &\textit{then there exists } p \textit{ such that } p \in pdirty\_T(owner(r), r)\\ &\quad\textit{or there exist } p, id \textit{ such that } \langle owner(r), p, id\rangle \in tdirty\_T(owner(r), r).\end{aligned}$$

□

Now, we prove the safety property.

THEOREM 13 SAFETY. *Birrell's algorithm is safe.* □

PROOF. A process may hold a potentially usable remote reference r, in which case the state of that reference is one of OK, nil or $\mathsf{ccitnil}$, or a message containing the reference may be in transit to the process. In the first case, we proceed by case analysis of the state of $rec\_T(p_1, r)$, for a process p_1 holding a reference r.

—$rec\_T(p_1, r) = \mathsf{OK}$: $p_1 \in pdirty\_T(p_2, r)$ by Lemma 9 (Usable Reference).

—$rec\_T(p, r) = \mathsf{nil}$ or $rec\_T(p, r) = \mathsf{ccitnil}$: In these cases, there exists p such that $p \in pdirty\_T(owner(r), r)$, or there exist p, id such that $\langle owner(r), p, id\rangle \in tdirty\_T(owner(r), r)$ by Lemma 11 (Unusable Reference).

The only remaining cases to consider are when $rec\_T(p_1, r)$ is $\perp$ or ccit and there is a message in transit: $\mathsf{copy}(r, id) \in k(p_1, p_2)$. Then $p_1 \in pdirty\_T(owner(r), r)$ or $\langle owner(r), p_2, id\rangle \in tdirty\_T(owner(r), r)$ by Lemma 10 (Reference in Transit).

Thus, all live references and all references contained in copy messages in transit must also be held in their owner's dirty tables (as transient or permanent entries). An object cannot be reclaimed by the local garbage collector as long as there is a reference to it held in the owner's dirty tables, since these are considered to be roots for local collections. Thus no object for which there is a live remote reference, or a reference in transit, can be reclaimed by the local collector. The algorithm is safe. □

4.2 Liveness proof

Liveness guarantees that if all references to an object are deleted, its owner's dirty tables will eventually become empty. In order to establish liveness, we adopt the proof technique used by Moreau and Duprat [2001]. Much of this proof consists of repetitive case analysis; in the interests of brevity, we give example cases rather than the complete analysis.

Outline. The liveness is derived by showing that the algorithm can always process incoming messages and in very precise circumstances generate a finite amount of transitions. Therefore, assuming fairness of the communication layer, we can show that when no more transitions can be performed for a given reference, its owner's dirty tables are necessarily empty.□

We first show that whenever there is a message in a channel, a transition may be fired to consume this message.

LEMMA 14. *Let c be a configuration $\langle \ldots, k\rangle$ such that $k(p_1, p_2) = \{m\} \oplus q$, for some m, p_1, p_2 and q. Then, there exist a transition t and a configuration $c' = \langle \ldots, k'\rangle$ such that $c \mapsto^t c'$, with $k'(p_1, p_2) = q$.* □

PROOF. Proof proceeds by case analysis on the type of the message m known to be in a channel. For instance, if there is a message $\mathsf{dirty\_ack}(r)$ in a communication channel $k(p_1, p_2)$, then transition $receive\_dirty\_ack(p_1, p_2, r)$ can be fired to consume this message. □

Lemma 14 ensures that the algorithm itself does not prevent the processing of messages.

Our next step is to prove that the distributed reference listing activity generates a finite number of transitions. To this end, we need to separate transitions that are triggered by the application from those that are specific to the reference listing algorithm. The transition *make_copy* is initiated only by the application, which is external to this algorithm; likewise, transition *finalize* is triggered by a local garbage collector after the application has released all references to an object. Thus, we are required to show that the length of a sequence of transitions that does not include transitions *make_copy* or *finalize* is necessarily finite.

For this purpose, we introduce a new measure, called a *termination measure*, that gives an indication of how far the abstract machine is from completing its transitions related to distributed reference listing. The termination measure is defined in terms of a configuration of the machine as follows.

Definition 15 Termination Measure. *The termination measure of a configuration c,* $\langle tdirty\_T, pdirty\_T, rec\_T, blocked\_T, copy\_ack\_todo\_T, dirty\_ack\_todo\_T, clean\_ack\_todo\_T, dirty\_call\_todo\_T, clean\_call\_todo\_T, k\rangle$*, is defined as:*

$$\begin{aligned} & termination\_measure(c) \\ & \quad = tab\_measure \\ & \quad + \sum_{p_i\in\mathcal{P}}\sum_{p_j\in\mathcal{P}}\sum_{m\in k(p_i,p_j)} msg\_measure(m) \\ & \quad + \sum_{p\in\mathcal{P}}\sum_{r\in\mathcal{R}} rt\_measure(rec\_T(p,r)) \end{aligned}$$

with

$$\begin{aligned} tab\_measure \;=\; & 9\ |dirty\_call\_todo\_T| + 7\ |dirty\_ack\_todo\_T| \\ & + \ 2\ |copy\_ack\_todo\_T| + 2\ |clean\_ack\_todo\_T| \\ & + \ 2\ |blocked\_T| \end{aligned}$$

and

$$\begin{aligned} msg\_measure(\mathsf{copy}) &= 14 & rt\_measure(\mathsf{OK}) &= 5 \\ msg\_measure(\mathsf{dirty}) &= 8 & rt\_measure(\mathsf{ccitnil}) &= 2 \\ msg\_measure(\mathsf{dirty\_ack}) &= 6 & rt\_measure(\mathsf{ccit}) &= 1 \\ msg\_measure(\mathsf{clean}) &= 3 & rt\_measure(\mathsf{nil}) &= 1 \\ msg\_measure(\mathsf{copy\_ack}) &= 1 & rt\_measure(\bot) &= 0 \\ msg\_measure(\mathsf{clean\_ack}) &= 1 \end{aligned}$$

□

Intuitively, the processing of a message can update a table, which in turn may trigger the creation of new messages. The termination measure of a configuration accounts for messages, the sizes of tables and the states of references. The values for the component measures are chosen such that the termination measure of a configuration is always greater than that of any successor configuration; pseudo-statements in Figures 9 to 12 are annotated with the change in termination measure they cause. This is formalised by the following lemma.

LEMMA 16. *For any configurations c and c' and for any transition t, such that $c \mapsto^t c'$, where $t \neq make\_copy(p_1, p_2, r)$ and $t \neq finalize(p, r)$, the following inequality holds:*

$$0 \leq termination\_measure(c') < termination\_measure(c).$$

□

PROOF. First, we note that the termination measure of a configuration is always positive or null. Second, the proof proceeds by an analysis of the different possible cases for transition t. We consider here the transition *receive_dirty_ack*. We compute the termination measure of the configuration after transition:

(1) a dirty_ack is consumed, hence the measure is decreased by 6;
(2) the increase of the measure due to *copy_ack_todo_T* is exactly compensated by the decrease of the measure due to *blocked_T*;
(3) setting $rec\_T(p_2, r)$ to OK increases the measure by 5.

As a result, after transition, the measure is decreased by 1, which proves the lemma. Similar reasoning shows that the other transitions also cause the termination measure to decrease strictly. □

Knowing that the termination measure is non-negative, and having proved that it decreases for every transition other than *make_copy* and $finalize$[8], we can derive the following termination Lemma.

LEMMA 17 TERMINATION. *For any configuration, all transition paths that do not involve make_copy or $finalize$ transitions terminate.* □

PROOF. Let us define a *successor* relation on the set of configurations; c_2 is a successor of c_1 if c_2 is obtained from c_1 by a transition that differs from *make_copy* or $finalize$. Using the termination measure (Definition 15) and the fact that it decreases (Lemma 16), we can establish that the successor relation is well-founded. Therefore, we can derive that, for any configuration c_0, there exists a successor configuration c_k that does not have a successor; c_k is a fixed point of the successor relation, which concludes the proof. □

All the necessary lemmas are in place to prove liveness, specified as follows.

DEFINITION 18 BIRRELL'S LIVENESS REQUIREMENT. *The liveness requirement for Birrell's algorithm is that if all references r to an object are deleted, its owner's dirty tables will eventually become empty.*

$$\begin{array}{rl} \textit{If} & \forall p \neq owner(r),\ \textit{if}\ rec\_T(p,r) = \mathsf{OK},\ \textit{then}\ r \in clean\_ack\_todo\_T(p) \\ \textit{and if} & \textit{no further}\ make\_copy(p_1,p_2,r)\ \textit{transition is fired} \\ \textit{and if} & \textit{no}\ \mathsf{copy}(r)\ \textit{is in transit}, \\ \textit{then} & pdirty\_T(owner(r),r)\ \textit{and}\ tdirty\_T(owner(r),r)\ \textit{become empty} \\ & \quad \textit{after a finite number of transitions.} \end{array}$$

□

[8]Note that we do not require a measure for *clean_call_todo_T* since the only transition that adds entries to it is *finalize* which is not enabled.

Before we can show that Birrell's algorithm satisfies the liveness requirement, we require two straightforward lemmas. First, we show that there can only be an entry in $blocked\_T(p_2, r)$ if there is a dirty or a $\mathsf{dirty\_ack}$ message in transit, or an entry in $dirty\_call\_todo\_T$ or $dirty\_ack\_todo\_T$ (indicating that one of these messages has been scheduled for transmission).

Lemma 19. *For any processes p_1 and p_2, any reference r, any identifier id and for any configuration, the following equality holds:*

$$\begin{aligned}&\langle id, p_1, r\rangle \in blocked\_T(p_2, r)\\ &= \bigvee\begin{cases} r \in dirty\_call\_todo\_T(p_2)\\ \mathsf{dirty}(r, id) \in k(p_2, p_1)\\ \langle p_2, r\rangle \in dirty\_ack\_todo\_T(p_1)\\ \mathsf{dirty\_ack}(r) \in k(p_1, p_2)\end{cases}\end{aligned}$$

□

Proof. The proof follows by case analysis of the transitions in the same fashion as that of Lemma 5 and is omitted here. We also note that these four cases are mutually exclusive. □

Second, we show that if the state of a reference is nil, then the reference is blocked.

Lemma 20. *For any process p_1, any reference r and for any configuration,*

If $rec\_T(p_1, r) = \mathsf{nil}$,
then $\langle id, p_2, r\rangle \in blocked\_T(p_1, r)$ for some id, p_2.

□

Proof. The proof follows trivially by case analysis of transitions *receive_copy*, *receive_dirty_ack* and *receive_clean_ack*. □

Theorem 21 Liveness. *Birrell's algorithm satisfies the liveness requirement.*
□

Proof. We assume that no further *finalize* or *make_copy* transitions related to a given reference r can be fired. By further assuming fairness [Manna and Pnuelli 1991] of message delivery, all transitions pertaining to r will eventually be executed. Consequently, by Lemma 17, there exists a configuration of the abstract machine that cannot fire any further transition pertaining to r. This configuration cannot contain an entry for r in any *to do* table, for otherwise new r-related transitions would be fireable; likewise, this configuration cannot contain any r-related message. Therefore, the algorithm terminates in a finite number of steps for transitions related reference r.

We now show that $pdirty\_T(Owner(r), r)$ and $tdirty\_T(owner(r), r)$ are empty on termination. $\langle p_1, p_2, id\rangle \in tdirty\_T(p_1, r)$ and $\langle id, p_1, r\rangle \in blocked\_T(p_2, r)$, are equivalent by Lemma 3. But since the *to do* tables are empty and there are no messages in transit, $blocked\_T(p_2, r)$ is empty by Lemma 19, and hence there is no transient entry in $tdirty\_T(p_1, r)$. By Lemma 6, $p_2 \in pdirty\_T(p_1, r)$ if and only if the state of r in process p_2 is one of OK, nil or $\mathsf{ccitnil}$. However, $rec\_T(p_2, r)$ can neither be OK (by the premise of Definition 18), nor nil (by Lemma 20) nor $\mathsf{ccitnil}$ (by

Lemma 1 since the $dirty\_call\_todo\_T$-tables are empty). Therefore, $pdirty\_T(p_1, r)$ cannot contain a permanent entry. Since neither dirty table contains an entry relating to r, the liveness of the algorithm is proved. □

5. VARIANTS AND DISCUSSION

Changing the network hypothesis to FIFO channels can dramatically simplify the algorithm, while still preserving the approach of dirty and clean calls. Additionally, optimisations are possible when the sender or the receiver of a reference is also its owner. We study such variants of the algorithm in this section, and discuss their implications in terms of performance.

5.1 FIFO Channels

Changing the communication hypothesis to reliable FIFO channels not only simplifies the algorithm, but also improves its performance. In this section, we summarise this algorithm variant.

By adopting FIFO channels, message order is preserved. Consequently, clean messages are guaranteed not to overtake dirty messages already in transit between the same pair of processes. Therefore, there is no need to wait until a dirty_ack message is received before making a reference usable and potentially recyclable: as soon as a reference is received, it becomes usable and a dirty message should then be scheduled. This means that deserialisation does not have to be blocking, and that garbage collection activity does not force synchronisation points with the mutator activity.

A dirty_ack is still needed, however, to determine when a copy_ack message must be sent; otherwise, a premature copy_ack could entail a race condition typical of naive distributed reference counting. On the other hand, clean_ack messages have now become obsolete: they were only needed to mark the transition of receive tables from ccitnil to nil. As a result, only two states are required for receive tables: OK and $\perp$, marking a usable and an unusable reference, respectively. Finally, whereas we used to have two separate tables for clean and dirty calls scheduled for transmission, now we must use a single queue of calls to do, as their order must be preserved. We remark that, in this optimisation, the FIFO property on channels is required for dirty and clean messages between any process and the reference's owner. Mutator messages do not have to be ordered.

5.2 Owner Optimisations

Our presentation, like Birrell's, focuses on triangular protocols, where a reference sender, the reference receiver and the owner are three different processes. Some optimisation can take place when the sender or the receiver is also the owner.

5.2.1 *Sender is also Owner.* When the sender of a reference is also its owner, there is no point in creating a transient dirty entry only to be replaced by a permanent dirty entry, through two successive dirty and copy_ack messages. Instead, during a *make_copy* transition, the owner could directly create an entry in its permanent dirty table. When a process receives a reference from its owner, it no longer needs to make a dirty call and to send a copy_ack message.

This change potentially introduces race conditions. The owner could send a reference to a process p, and therefore create a permanent dirty entry for p. While the copy message is in transit, a third party process could send the same reference to p, which would then make a dirty call and, if the reference became no longer reachable on p, could immediately make a clean call. The clean call could remove the permanent dirty entry for p before the copy message reached p. To avoid such a race condition, messages need to be ordered, ensuring that the dirty_ack message does not overtake the copy message in transit from the owner.

5.2.2 *Receiver is also Owner.* In this case, there is no need for the sender to create a transient dirty entry for the reference, because we know that the reference is locally reachable by the owner, since the owner contains a permanent dirty entry for the sender. However, race conditions are again possible if message ordering is not introduced. If the sender sends a reference to its owner, does not create a transient dirty entry, recycles the reference immediately and sends a clean message immediately, we need to ensure that the clean message cannot overtake the copy message.

Thus, for both of these 'owner optimisations', and unlike the FIFO channel optimisation introduced above, we require ordering of application messages and some distributed reference listing messages.

5.3 Discussion

The application in which the algorithm is used would typically dictate the network conditions that are permissible. In an RPC-style environment, where remote method calls cannot be ordered, our first algorithm specification appears to be the only solution. It has the major drawback that the mutator's activity must be suspended during deserialisation by the reference listing algorithm.

By maintaining the order of clean and dirty messages, the algorithm variant of Section 5.1 potentially improves the performance of the overall application, because it does not force synchronisation points between the reference listing activity and the application.

The variant of Section 5.2, however, introduces quite stringent constraints on communication channels, sometimes forcing the interleaving of reference listing messages with application messages. It can, however, substantially reduce the traffic to and from a reference owner.

6. FAULT TOLERANCE

So far our discussion of Birrell's algorithm has been restricted to fault-free processes and communications. This is not a realistic model of a distributed world. Birrell's algorithm tolerates communication and process failures through a combination of failure detection, timeouts and message numbering. In this section, we first outline Birrell's approach [1993], discuss the limitation of that presentation, and then make use of our graphical representation to map failure detection and associated remedial actions, in order to provide new insights into Birrell's algorithm. Birrell writes:

> [Section 2] To deal with [calls being delivered out of order], a *sequence number* is attached to each clean or dirty call. The sequence number must increase with each new operation from the client. (Some authors use the term 'timestamp' to refer to this sort of sequence number.) Let $seqno(O, P)$

be the largest sequence number seen at O's owner on a clean or dirty call for object O from process P. An incoming operation will be performed only if its sequence number exceeds this value; otherwise it has no effect.

Communication failures
[Section 2.3] When a dirty call fails, no surrogate is created. It would not be safe to create one, because the object's owner may not have received the dirty call. However, it is also possible that the owner *did* receive the dirty call, so the object and a sequence number for the clean call are added to the cleanup demon's queue. Note that this may cause an unnecessary clean call, but that does no harm. The effect of a clean call is to remove the client from the object's dirty set; if it is not in the set, the clean call is a no-op.
When a clean call fails, the cleanup demon merely leaves the request on its queue, keeping the same sequence number. The clean call will be repeated until it succeeds, or until the owner's termination is detected.

Process termination
[Section 2.4] [Birrell's] collector detects termination by having each process periodically ping the clients that have surrogates for its objects. If the ping is not acknowledged after sufficient time, the client is assumed to have died, and is removed from all dirty sets at that owner.

This presentation suffers from some weaknesses, similar to those we discussed in Section 2.4. More specifically:

(1) As before, the presentation depends on specific implementation techniques:
   (a) It relies on an RPC-based communication layer, in which the process receiving a procedure call has little opportunity to take action when a failure occurs during the return of a result; it is, for example, difficult for such a process to retry sending the result as the connection initiated by the caller has probably been lost.
   (b) The detection of process failures is driven by the owner pinging client processes. This is not the only approach, however: for instance, Java RMI relies on a mechanism of leases that clients must renew regularly, failing which they are deemed dead by the owner.
   (c) The failure handling mechanism is bound to the wireRep representation, and in particular, it uses the wireRep uniqueness to detect processes considered to be dead.

(2) Birrell does not specify the global properties that the algorithm must preserve. It is particularly hard to identify such properties without any knowledge of the application, and therefore it seems difficult to find a generic solution at the middleware level.

(3) Different methods of communication and different failure detectors can result in different algorithmic solutions, and therefore should be made explicit as parameters to the algorithm.

(4) Sections 2.3 and 2.4 of Birrell's algorithm discuss how a client should behave when communications with the owner fail and how the owner should react to an apparently dead client, respectively. These sections are presented independently, but it is likely that, in the presence of communication failures, both would be applicable at the same time; therefore, their interaction needs to be discussed.

Against this background, we believe that fault tolerance deserves a complete study that is beyond the scope of this article. Indeed, following our investigation, we estimate that a minimum of nine new rules would be necessary to formalise Birrell's approach, and their proof of correctness would require substantial reworking and extension of the current proof. Instead, our contribution is to shed some light on Birrell's approach through our graphical notation. This allows us to: *(i)* identify precisely when failures can be detected; *(ii)* define the states that are reached after failures have been detected; *(iii)* illustrate how remedial actions attempt to recover the system; and *(iv)* show which cases must be added to Birrell's algorithm to avoid deadlocks.

Figure 13 depicts two nested cubes: the inner one maps the transitions and states of the algorithm in the absence of failures (as in Figure 4), whereas the outer cube identifies states after failures have been detected. By convention, the states reached after detecting a failure are overlined; for clarity, the states 1, 2,. . . of Figure 4 that describe the sending of references have been omitted. All failure detection transitions move from a vertex of the inner cube to a vertex of the outer one (and vice-versa for failure-handling transitions).

The outer cube follows the same intuitive interpretation as that of Figures 5, 6 and 7. Horizontal slicing brings useful insights to failure detection. When communicating with the owner, the absence of acknowledgement can be detected, but the process cannot decide whether the owner has received the message. Therefore, the upper failure states (marked with a subscript u) identify states in which the owner (still) believes that the receiver has a reference, as opposed to the lower failure states (marked with a subscript l) in which the owner does not know that the receiver holds a reference.

In the failure-free life cycle, a process announces its possession of a reference by sending a dirty message to the reference's owner; this action results in the system moving from the state $\perp$ to the state nil. From this state, in the absence of failure, we would expect the receipt of a dirty_ack message from the owner. Alternatively, the process could observe a failure to deliver the dirty message, or could conclude after a timeout that the dirty acknowledgement message has not been delivered. These two conditions result in a transition to some state in the outer cube. Using the intuition of horizontal slicing from Figure 6 ($u$pper and $l$ower slices), two states need to be be considered: in the first one, $\overline{\mathsf{nil}}_u$, the owner has received the dirty message, whereas in the second, $\overline{\mathsf{nil}}_l$, the owner has not received a dirty message. If the cause of the transition is that no dirty_ack message has been received, the client is unable to decide in which state the owner is; therefore, from a client's perspective, there are two non-deterministic transitions to $\overline{\mathsf{nil}}_u$ and $\overline{\mathsf{nil}}_l$.

A similar failure detection occurs in states ccit and ccitnil, in which the client process can detect the failure to deliver a clean message, or alternatively that a

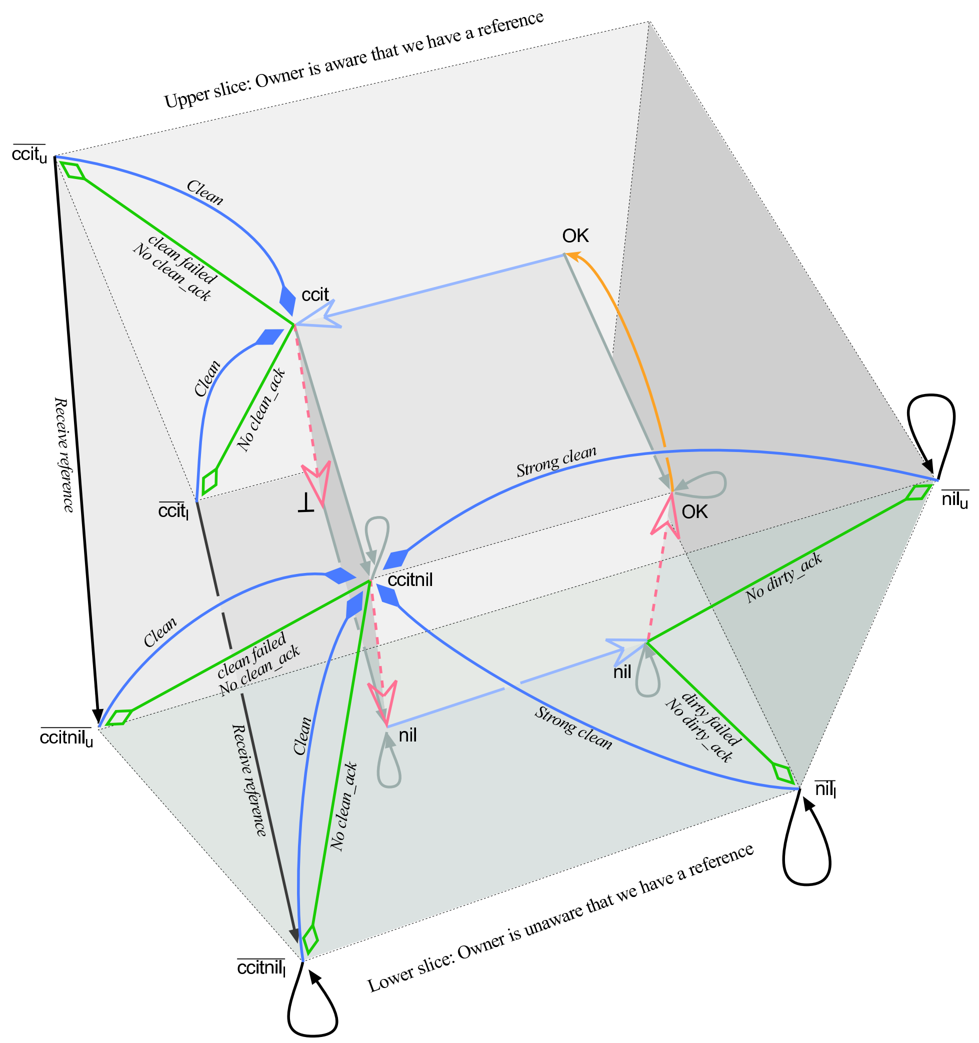

Upper slice: Owner is aware that we have a reference
$\overline{\text{ccit}}_u$
Clean
clean failed
No clean_ack
ccit
OK
Clean
No clean_ack
Receive reference
$\overline{\text{ccit}}_l$
⊥
Strong clean
OK
$\overline{\text{nil}}_u$
ccitnil
Clean
clean failed
No clean_ack
No dirty_ack
nil
$\overline{\text{ccitnil}}_u$
Receive reference
Clean
No clean_ack
nil
Strong clean
dirty failed
No dirty_ack
$\overline{\text{nil}}_l$
Lower slice: Owner is unaware that we have a reference
$\overline{\text{ccitnil}}_l$

Fig. 13. Failure Detection and Handling, according to Birrell. Note that, for clarity, the transition labels of the inner cube have been omitted and its orientation has been changed from that of Figure 4.

$\mathsf{clean\_ack}$ message has not been received before the timeout. From ccit and $\mathsf{ccitnil}$, we find non-deterministic transitions to $\overline{\mathsf{ccit}}_u$ and $\overline{\mathsf{ccit}}_l$, and to $\overline{\mathsf{ccitnil}}_u$ and $\overline{\mathsf{ccitnil}}_l$, respectively.

Our notation also helps us to explain what remedial actions should be performed to recover from failures. For the states $\overline{\mathsf{ccit}}_u$, $\overline{\mathsf{ccit}}_l$, $\overline{\mathsf{ccitnil}}_u$ and $\overline{\mathsf{ccitnil}}_l$, the solution is to re-attempt the sending of a clean message, bringing the system back to the inner cube. As noted by Birrell, a failure may have been detected, but the clean message may have been successfully processed by the owner: additional clean messages are harmless.

The remedial actions for the states $\overline{\mathsf{nil}}_u$ and $\overline{\mathsf{nil}}_l$ are more interesting: the client process must send a clean message to clean up a potential permanent dirty entry in the owner process. Such clean calls are flagged as *strong clean* calls in Birrell's algorithm to ensure that the dirty call that was thought to have failed is indeed cancelled no matter when it arrives. From a client's perspective, the reference is still potentially usable, even though it is not known whether it has been registered with the owner yet; hence, the remedial action leads to the ccitnil state.

Finally, at any stage of a reference life cycle, a new copy message containing the same reference may be received by the process: it needs to be handled, otherwise the algorithm would lead to a deadlock. Receiving such a message in $\overline{\mathsf{ccit}}_u$ and $\overline{\mathsf{ccit}}_l$ makes the reference potentially usable, which is exactly the meaning of $\overline{\mathsf{ccitnil}}_u$ and $\overline{\mathsf{ccitnil}}_l$, towards which we have introduced new 'receive reference' transitions. For all the other states of the outer cube, receiving a copy does not change the state of the system.

Our mapping of failure detection and remedial actions to the graphical notation brings new insights into the algorithm:

(1) Birrell's algorithm did not have a ccitnil state. We introduced this state in order to prove the correctness of the algorithm in the absence of failures. Figure 13 shows the pivotal role of this state in the handling of failures.

(2) A crucial consideration when designing an algorithm is to prevent the creation of deadlocks: our graphical notation helped us identify what action to take in the presence of incoming copy messages, resulting in new transitions $\overline{\mathsf{ccit}}_u \rightarrow \overline{\mathsf{ccitnil}}_u$ and $\overline{\mathsf{ccit}}_l \rightarrow \overline{\mathsf{ccitnil}}_l$.

(3) Our presentation of the algorithm focuses on the client's perspective, for which we identified the presence of non-determinism when failures are detected. This non-determinism reflects the client's inability to decide whether the owner has processed a message or not. After mapping the remedial actions to the graphical notation, we observe that, *from a client's perspective*, the upper states can be merged with their corresponding lower states, since the remedial actions are identical. However, it remains important to distinguish these states, because the owner is itself capable of deciding whether it has processed a message or not.

(4) Similarly to Birrell's algorithm, our presentation has focused only on the handling of failures for dirty and clean messages. Failures of copy_ack messages should also be handled because their processing affects the transient entries in dirty tables.

(5) Remedial actions attempt to bring the system back to states in the inner cube, but they can fail themselves. Our graphical notation shows that their failure would be handled in a similar manner, but this could result in infinite loops. The algorithm would have to use a timeout to perform a transition to a state, which we have not represented in our pictures, in which the process (or communication) is believed to be dead (or irremediably broken).

By using the graphical notations, we have identified six transitions to the outer cube, six transitions to handle failures (or three if the upper and lower states are collapsed), and two supplementary cases to handle incoming copy messages. If we were to formalise these transitions, a minimum of nine new rules would have to be added to our algorithm, and most of the other rules would have to be revisited in order to support message sequencing. The proof would have to be completely revised as some of the key invariants would no longer hold in the new system. For instance, the terms of Lemma 4 would no longer be mutually exclusive, because several identical clean messages could be in transit due to slow communication, timeouts and resending by the client. Finally, the key problem with formalising and proving the correctness of the fault-tolerant version of the algorithm is to specify the overall properties that need to be preserved. This study is beyond the scope of the current paper.

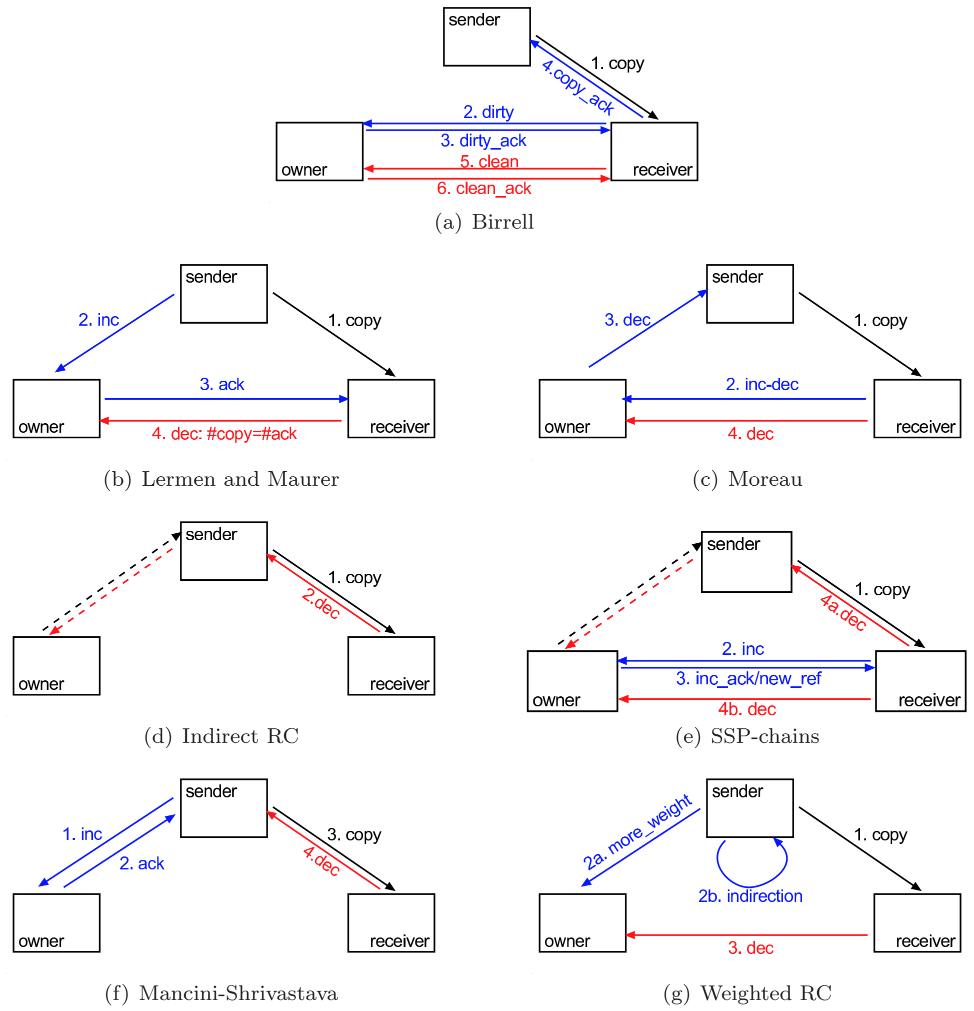
7. RELATED WORK

In this section, we start by comparing Birrell's algorithm with other techniques for distributed garbage collection before discussing other formalisms and proof techniques.

7.1 Reference Counting Algorithms for Garbage Collection

Distributed reference counting or listing algorithms are direct: they immediately (subject to batching of messages) identify those objects for which there are no longer global references. Consequently, reference counting scales well to large networks. A further advantage is that it is comparatively straightforward to implement. Designers of distributed reference counting algorithms face challenges of safety and efficiency: *(i)* to avoid race conditions between messages between reference owners, senders and receivers that may cause objects to be reclaimed prematurely (described in Section 2.2); *(ii)* to ensure that garbage objects are eventually reclaimed (liveness); and *(iii)* to minimise the number of messages that must be exchanged. Birrell's is one of many solutions to address the first problem.

The earliest algorithm, from Lermen and Maurer [1986], used a system of acknowledgements to prevent decrement (DEC) messages from overtaking increment (INC) messages. The sender of a reference must also notify the reference's owner, who will then send an acknowledgement (ACK) to the receiver. By waiting until the count of receipts of references and the count of their acknowledgements is equal, a receiving process can delay sending decrement messages until it knows that the increment has been received. Lermen and Maurer's algorithm is illustrated in Figure 14(b). Birrell's algorithm improves on Lermen and Maurer *in the context of RPC* as it performs fewer RPC calls when copying a reference (however, more messages are sent). Unlike Lermen and Maurer, it is the receiver rather the



(a) Birrell

(b) Lermen and Maurer

(c) Moreau

(d) Indirect RC

(e) SSP-chains

(f) Mancini-Shrivastava

(g) Weighted RC

Fig. 14. Simplified message exchanges in distributed reference counting protocols

sender that initiates the exchange of reference counting messages.

In a fashion similar to that of Birrell, Moreau's algorithm [2001b; 2001] requires the receiver of a reference to send an INC_DEC message to the owner, which in turn sends a DEC message to the reference sender (and, thus, the INC_DEC does not require an acknowledgement). The algorithm is illustrated in Figure 14(c). To reduce synchronisation between mutator and distributed memory manager, Moreau requires point-to-point FIFO ordering of messages, which has inspired the optimisation that we discussed in Section 5.1.

The triangular protocols used by these algorithms are necessary to avoid race conditions between reference counting messages that might lead to premature deletion of objects. Indirect Reference Counting (IRC) [Piquer 1991] and Weighted Reference Counting (WRC) [Bevan 1987; Watson and Watson 1987; Foster 1989;

Dickman 1992] can avoid such races by sending only decrement messages.

Piquer's IRC instead maintains a *diffusion tree* that represents the path along which distributed references have been propagated for the first time between processes. Each process increments a counter for a reference whenever it copies the reference to another process; when a reference is deleted, the remote process sends a decrement message back to the reference's parent in the diffusion tree where the counter is decremented (see Figure 14(d)). The disadvantages of this technique are that a process may have to preserve a locally unreachable *zombie* reference simply because it is required by a third process, and that diffusion tree algorithms are vulnerable to the failure of a process holding a parent node of the tree.

Moreau's [2001b; 2001] and Dickman's [2000] algorithms show how zombies can be removed by reorganising the diffusion tree by re-parenting child nodes, either with the root node (Moreau) or arbitrarily (Dickman). Indeed, IRC can be seen as a special case of Moreau's algorithm. The idea of short-cutting chains of pointers was introduced by Shapiro, Gruber and Plainfossé [1990], and subsequently by Shapiro, Dickman, and Plainfossé [1992; 1992; 1995], with SSP (scion/stub pair) chains (see Figure 14(e)). Like Birrell, their description of SSP chains is focused on implementation and would benefit from an implementation-independent description.

Mancini and Shrivastava [1991] address fault tolerance in diffusion trees through a sender-initiated triangular protocol that requires the sender to notify the owner, and wait for an acknowledgement, before posting a reference (see Figure 14(f)). As with Birrell, this introduces synchronisation between the mutator and the distributed memory manager.

WRC (Figure 14(g)) associates a weight with each object and with each remote reference. Initially, the weight of the first remote reference to an object is equal to the weight of the object. When the reference is copied, its weight is divided (equally) between the two copies, thereby maintaining the invariant that the weight of an object is equal to the sum of the weights of the references that point to it. In order to divide a reference of weight one, either an indirection can be introduced, which behaves as an IRC zombie ('message' 2b), or a message can be sent to the owner requesting more weight (message 2a) [Corporaal et al. 1990]. The 'send more weight' solution is reminiscent of dirty messages but sender rather than receiver initiated.

Although reference counting is only able to reclaim acyclic data structures, it may be combined with tracing techniques to provide a complete distributed collector. The strategy of such hybrid collectors is to combine the efficiency and immediacy of reference counting with less frequent traces, possibly restricted to a subset of the reference graph. Examples include the partial tracing techniques of Rodrigues and Jones [1996; 1998], Jones and Lins [1993; 1993] and Lang, Queinnec and Piquer [1992], timestamp propagation from Le Fessant, Piumarta, and Shapiro [1998], and Maheshwari and Liskov's back-tracing algorithm [1997].

7.2 Comparison with Other Formalisations

Garbage collection algorithms, particularly those with any aspect of concurrency, whether mutator-collector or collector-collector, are notoriously challenging. For example, Dijkstra described concurrent garbage collection 'as one of the more challenging — and hopefully instructive — problems' in parallel programming. In consequence, most effort at formalisation and proof of garbage collection algorithms has been directed at concurrent GC algorithms, and in particular at Dijkstra, Lamport, Martin, Scholten, and Steffens's tricolour marking scheme [1978]. The only formalisation of a distributed garbage collection algorithm of which we are aware is that of the first author [Moreau and Duprat 2001].

Fine-grained concurrent implementations have several traps for the unwary, of which possibly the best-known was that discovered in Dijkstra's algorithm by Woodger and Stenning [Stenning 1976]. In describing his proof of the algorithm, Gries reported that he had 'seen five purported solutions to this problem, either in print or ready to be submitted for publication' [Gries 1977]. Ramesh and Mehndiratta [1983] formalised the proof of termination and absence of live-lock by using Owicki and Lamport's proof procedure [1982]. The algorithm has also been formalised by Jackson [1998], who used a labelled transition system and proved its correctness using an embedding of linear temporal logic in PVS, and by Goguen, Brooksby and Burstall [1998], based on a graph-theoretic representation of memory and related operations. Other algorithms and proofs can be found in Kung and Song [1977], Francez [1978] and Müller [1976]. Ben-Ari's algorithms [1982; 1984] were based on Dijkstra's but intended to have much simpler proofs of correctness. Nevertheless, Ben-Ari's algorithm was also susceptible to the Woodger scenario [van de Snepscheut 1987; Pixley 1988; Russinoff 1994]; both van de Snepscheut [1987] and Pixley [1988] use invariant techniques for their proofs. Gonthier and Doligez [1994; 1996] use a sugared version of the Temporal Logic of Actions [Lamport 1991] to formalise and prove correct a multi-processor concurrent garbage collector for Caml-light.

Birrell's original account of the algorithm sketched a proof of correctness based on temporal reasoning. However, in this paper, we use a notation previously employed to formalise other algorithms for distributed reference counting [Moreau and Duprat 2001] and a directory service for mobile agents [Moreau 2001a; 2002]. Our formalisation denotes an abstract machine whose transitions describe how the distributed system is able to evolve. Correctness proofs are established through an invariant technique, showing that a property is true in the initial configuration of the machine, and remains true for every possible transition. This style of proof mechanises straightforwardly, using the Coq theorem prover [Moreau and Duprat 2001; Moreau 2001a; 2002], and avoids the complications of temporal reasoning. Russinoff [1994] has mechanised a proof of Ben-Ari's algorithm, using the Boyer-Moore prover; his proof system was based on the Manna-Pnueli model of concurrency. As we found in this work, Russinoff also found it necessary to identify implicit assumptions made by the algorithm designer and to discard implementation-dependencies. Havelund and Shankar [1997] use refinement techniques in their mechanised safety proof.

8. FURTHER WORK

In this paper, we have used a systematic notation to formally specify a version of Birrell's algorithm without fault tolerance and we have proved that the formalisation provides the required safety and liveness guarantees. In Section 6, we used our graphical notation to identify precisely where failures may occur and must be handled and showed the remedial actions that would be necessary. We intend to formalise this fault-tolerant version and prove its correctness. We remarked in Section 3 that our formal notation bears some similarity with executable pseudo-code. A further development of this work will automatically generate executable code from our formal description. While we do not necessarily expect the code output to be optimal, it would nevertheless serve as a reference version for more efficient implementations.

Finally, we hope to use our graphical and formal notations to describe other algorithms in the distributed reference counting family. Initial experiments show that the notation can capture these algorithms [Lermen and Maurer 1986; Moreau and Duprat 2001] and that, again, three dimensions are required. We intend to explore this further and believe that such an exercise would be instructive, not least because it would make more explicit the similarities and differences of the algorithms.

9. CONCLUSION

Using a systematic notation, we have formally specified Birrell's distributed reference listing algorithm. This specification also led to a graphical representation of the state transition diagram and an intuitive explanation of the reference life cycle. Our specification is independent of any implementation technique, and hence makes the algorithm reusable in other contexts, not necessarily tied to distributed garbage collection (such as distributed termination detection [Tel and Mattern 1993]). Specifically, it neither assumes a remote procedure call communication mechanism nor does it assume the use of stacks. Critical sections are made explicit in our algorithm through our rules, whose execution is assumed to be atomic.

Our formalisation also clarifies issues in Birrell's original account of the algorithm, namely that references passed as a result of a remote method invocation also need to be kept reachable (in our algorithm, using the dirty tables), and that attention should be paid to race conditions that may occur when a fresh copy of a reference is received while it is in the process of being cleaned. Thus, our algorithm differs from Birrell's in the following ways:

(1) We make systematic and uniform use of transient dirty tables for both owners and non-owners in order to record the fact that a reference has been sent to a remote process.

(2) We introduce an extra state ccitnil to allow a process that has already sent a clean call for a reference, but has not yet received an acknowledgement, to handle an incoming copy of that reference.

We have proved the correctness of the implementation by deriving a set of invariants: proofs in such a style are easier to establish than temporal-based reasoning and can naturally lead to their mechanisation using a theorem prover.

Our graphical notation also brings new insight into the fault tolerance of Birrell's algorithm. In particular, it is readily apparent precisely where failure can be detected, which new states must be introduced and how recovery may be attempted. Once again, this shows the pivotal role of our new state, ccitnil. A final benefit of our work is that use of a common notation to describe algorithms can help, in the longer term, to compare such algorithms in a more systematic, if not formal, manner.

Acknowledgement

This research is funded in part by the UK EPSRC Memory Management Network (reference GR/R57140), to whom we are most grateful. We also thank the anonymous referees for their useful feedback which has improved this paper significantly.

REFERENCES

Abdullahi, S. E. and Ringwood, G. A. 1998. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys 30,* 3 (Sept.), 330–373.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin, C., Saïbi, A., and Werner, B. 1997. The Coq Proof Assistant Reference Manual – Version V6.1. Tech. Rep. 0203, INRIA. August.

Ben-Ari, M. 1982. On-the-fly garbage collection: New algorithms inspired by program proofs. In *Automata, languages and programming. Ninth colloquium*, M. Nielsen and E. M. Schmidt, Eds. Springer-Verlag, Aarhus, Denmark, 14–22.

Ben-Ari, M. 1984. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems 6,* 3 (July), 333–344.

Bevan, D. I. 1987. Distributed garbage collection using reference counting. See de Bakker et al. [1987], 176–187.

Birrell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E. 1993. Distributed Garbage Collection for Network Objects. Tech. Rep. 116, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301. Dec.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E. 1994a. Network Objects. Tech. Rep. 115, Digital Systems Research Center. Feb.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E. 1994b. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles.* ACM Press, Asheville, NC, 217–230.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E. 1995. Network objects. *Software Practice and Experience 25,* 4 (Dec.), 87–130.

Collins, G. E. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM 3,* 12 (Dec.), 655–657.

Corporaal, H., Veldman, T., and van de Goor, A. J. 1990. Efficient, reference weight-based garbage collection method for distributed systems. In *PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications.* IEEE Press, Miami Beach, 463–465.

de Bakker, J. W., Nijman, L., and Treleaven, P. C., Eds. 1987. *PARLE'87 Parallel Architectures and Languages Europe.* Lecture Notes in Computer Science, vol. 258/259. Springer-Verlag, Eindhoven, The Netherlands.

Dickman, P. 1992. Optimising Weighted Reference Counts for Scalable Fault-Tolerant Distributed Object-Support Systems.

Dickman, P. 2000. Diffusion Tree Redirection for Indirect Reference Counting. In *Proceedings of the Second International Symposium on Memory Management*, T. Hosking, Ed. ACM, Minneapolis, MN.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21,* 11 (Nov.), 966–975.

Doligez, D. and Gonthier, G. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages.* ACM SIGPLAN Notices. ACM Press, Portland, OR.

Foster, I. 1989. A Multicomputer Garbage Collector for a Single-Assignment Language. *Intl J. of Parallel Programming 18,* 3, 181–203.

Francez, N. 1978. An application of a method for analysis of cyclic programs. *ACM Transactions on Software Engineering 4,* 5 (Sept.), 371–377.

GOGUEN, H., BROOKSBY, R., AND BURSTALL, R. 1998. An Abstract Formulation of Memory Management. Available from `http://www.dcs.ed.ac.uk/~hhg/`.

GONTHIER, G. 1996. Verifying the safety of a practical concurrent garbage collector. In *Computer Aided Verification CAV'96*, R. Alur and T. Henzinger, Eds. Lecture Notes in Computer Science. Springer-Verlag, New Brunswick, NJ.

GRIES, D. 1977. An exercise in proving parallel programs correct. *Communications of the ACM 20,* 12 (Dec.), 921–930.

HAVELUND, K. AND SHANKAR, N. 1997. A mechanized refinement proof for a garbage collector. Available from `http://www.cs.auc.dk/~havelund/`.

HIRZEL, M., DIWAN, A., AND HENKEL, J. 2002. On the usefulness of type and liveness for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems 24,* 6 (Nov.), 593–624.

JACKSON, P. 1998. Verifying a garbage collection algorithm. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98*. Lecture Notes in Computer Science, vol. 1479. Springer-Verlag, Canberra, 225–244.

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester. With a chapter on Distributed Garbage Collection by R. Lins.

JONES, R. E. AND LINS, R. D. 1993. Cyclic weighted reference counting without delay. In *PARLE'93 Parallel Architectures and Languages Europe*, A. Bode, M. Reeve, and G. Wolf, Eds. Lecture Notes in Computer Science, vol. 694. Springer-Verlag, Munich, 712–515.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6,* 1 (Jan.), 109–133.

KUNG, H. T. AND SONG, S. W. 1977. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*. IEEE Press, 120–131.

LAMPORT, L. 1991. The temporal logic of actions. Research Report 79, DEC Systems Research Center, Palo Alto, CA.

LANG, B., QUENNIAC, C., AND PIQUER, J. 1992. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGPLAN Notices. ACM Press, Albuquerque, NM, 39–50.

LE FESSANT, F., PIUMARTA, I., AND SHAPIRO, M. 1998. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices. ACM Press, Montreal, 152–161.

LERMEN, C.-W. AND MAURER, D. 1986. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN Notices. ACM Press, Cambridge, MA, 343–350.

LINS, R. D. AND JONES, R. E. 1993. Cyclic weighted reference counting. In *Procedings of WP & DP'93 Workshop on Parallel and Distributed Processing*, K. Boyanov, Ed. North Holland, Sofia, Bulgaria, 369–382. Also Computing Laboratory Technical Report 95, University of Kent, December 1991.

MAHESHWARI, U. AND LISKOV, B. 1997. Collecting cyclic distributed garbage by back tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*. ACM Press, Santa Barbara, CA, 239–248.

MANCINI, L. V. AND SHRIVASTAVA, S. K. 1991. Fault-tolerant reference counting for garbage collection in distributed systems. *Computer Journal 34,* 6 (Dec.), 503–513.

MANNA, Z. AND PNUELLI, M. 1991. *Temporal Logic or Reactive and Concurrent Systems: Specification*. Springer.

MOREAU, L. 2001a. Distributed Directory Service and Message Router for Mobile Agents. *Science of Computer Programming 39,* 2–3, 249–272.

MOREAU, L. 2001b. Tree Rerooting in Distributed Garbage Collection: Implementation and Performance Evaluation. *Higher-Order and Symbolic Computation 14,* 4 (Dec.), 357–386.

MOREAU, L. 2002. A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers. In *The 17th ACM Symposium on Applied Computing (SAC'2002) — Track on Agents, Interactions, Mobility and Systems*. ACM, Madrid, Spain, 93–100.

MOREAU, L. AND DUPRAT, J. 2001. A Construction of Distributed Reference Counting. *Acta Informatica 37*, 563–595.

MÜLLER, K. A. G. 1976. On the feasibility of concurrent garbage collection. Ph.D. thesis, Tech. Hogeschool Delft.

OWICKI, S. AND LAMPORT, L. 1982. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems 4,* 3 (July), 455–495.

PIQUER, J. M. 1991. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE'91 Parallel Architectures and Languages Europe*, Aarts et al., Eds. Lecture Notes in Computer Science, vol. 505. Springer-Verlag, Eindhoven, The Netherlands.

PIXLEY, C. 1988. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing 3,* 1, 41–50.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, H. Baker, Ed. Lecture Notes in Computer Science, vol. 986. Springer-Verlag, ILOG, Gentilly, France, and INRIA, Le Chesnay, France.

RAMESH, S. AND MEHNDIRATTA, S. L. 1983. The liveness property of on-the-fly garbage collector — a proof. *Information Processing Letters 17,* 4 (Nov.), 189–195.

RODRIGUES, H. C. C. D. AND JONES, R. E. 1996. A cyclic distributed garbage collector for Network Objects. In *Tenth International Workshop on Distributed Algorithms WDAG'96*, O. Babaoglu and K. Marzullo, Eds. Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Bologna, 123–140.

RODRIGUES, H. C. C. D. AND JONES, R. E. 1998. Cyclic distributed garbage collection with group merger. In *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, Brussels, 249–273. Also UKC Technical report 17–97, December 1997.

RÖJEMO, N. AND RUNCIMAN, C. 1996. Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *Proceedings of First International Conference on Functional Programming.* ACM Press, Philadelphia, PA, 34–41.

RUSSINOFF, D. M. 1994. A mechanically verified incremental garbage collector. *Formal Aspects of Computing 6*, 359–390.

SHAHAM, R., KOLODNER, E., AND SAGIV, M. 2002. Estimating the impact of liveness information on space consumption in Java. In *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, D. Detlefs, Ed. ACM SIGPLAN Notices. ACM Press, Berlin, 64–75.

SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. 1992. Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing.* ACM Press, Vancouver, Canada, 135–146. Superseded by [**?**].

SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. 1992. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Rapport de Recherche 1799, INRIA-Rocquencourt. Nov.

SHAPIRO, M., GRUBER, O., AND PLAINFOSS, D. 1990. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Inria-Rocquencourt. Nov.

STENNING, V. 1976. On-the-fly garbage collection. Unpublished notes, cited by [Gries 1977].

Sun Microsystems 1996. *Java Remote Method Invocation Specification.* Sun Microsystems. http://java.sun.com/products/jdk/rmi/.

TEL, G. AND MATTERN, F. 1993. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Trans. Program. Lang. Syst. 15,* 1 (Jan.), 1–35.

VAN DE SNEPSCHEUT, J. 1987. Algorithms for on-the-fly garbage collection revisited. *Information Processing Letters 24,* 4 (Mar.), 211–216.

WATSON, P. AND WATSON, I. 1987. An efficient garbage collection scheme for parallel computer architectures. See de Bakker et al. [1987], 432–443.

Received Month Year; revised Month Year; accepted Month Year