A Construction of Distributed Reference Counting

Luc Moreau\textsuperscript{1}, Jean Duprat\textsuperscript{2}

\textsuperscript{1} University of Southampton, UK, e-mail: L.Moreau@ecs.soton.ac.uk
\textsuperscript{2} Ecole Normale Supérieure, Lyon, France, e-mail: Jean.Duprat@ens-lyon.fr

Received: date / Revised version: date

Abstract. Distributed reference counting is a general purpose technique, which may be used, e.g., to detect termination of distributed programs or to implement distributed garbage collection. We present a distributed reference counting algorithm and a mechanical proof of correctness carried out using the proof assistant Coq. The algorithm is formalised by an abstract machine, and its correctness has two different facets. The safety property ensures that if there exists a reference to a resource, then its reference counter will be strictly positive. Liveness guarantees that if all references to a resource are deleted, its reference counter will eventually become null.

1 Introduction

Reference counting is a general purpose technique that is able to count the number of references to a given resource. Collins [5] was the first to use it in order to determine when list cells were no longer needed. Operating systems rely on this technique in order to decide when files may be deleted or when file descriptors may be closed. Reference counting is also a method for implementing garbage collection, a memory management technique that automatically determines when objects may be deallocated. We refer the reader to Jones and Lins’ book [19, section 2.1] for a discussion of the pros and cons of this technique for garbage collection purpose.

Distributed reference counting is an extension of reference counting where a resource and its users may be located at different positions. The difficulty of a distributed environment is that the decision
of whether a resource is used can no longer be taken locally, but must involve a collaboration with the different locations participating in the computation. Distributed reference counting may be used to implement distributed garbage collection; a variant of this technique is in particular used in Java and RMI [27,18]. Even though distributed reference counting is not able to deal with distributed cycles, it has been a popular implementation technique of distributed garbage collection because it is simple to implement and can nicely be integrated with sequential garbage collectors [3,27,32,41]. More generally, it may be used for tracking references to resources [15]. A possible use is to detect termination of distributed programs [40]; reference counting may be used for such an application because processes form a hierarchy. Groups [31] also have a hierarchical organisation and can be reference counted.

The first author recently published a new algorithm for distributed reference counting [27]. It has the property that all references may be found at any time, which can be useful when the owner of a resource wishes to propagate information to the resource users. In fact, this algorithm describes a family of implementations, according to the policy adopted to propagate messages. In particular, Piquer’s Indirect Reference Counting [32] can be seen as a particular instance of our algorithm.

The purpose of this paper is to present this algorithm and to prove its correctness. The correctness of a reference counting algorithm has two different facets. Safety guarantees that if there exists a reference to a resource, then its reference counter will be strictly positive. Liveness guarantees that if all references to a resource are deleted, its reference counter will eventually become null.

The contribution of this paper is the description of a mechanical proof that has been carried out using the calculus of inductive constructions and the proof assistant Coq [1]. We have also studied some optimisations and have considered two algorithm variants. In particular, we present reference listing, which is a variant of the algorithm that not only counts references to a resource, but also remembers where those references were passed. Reference listing is a useful technique to assist in building a fault tolerant version of the algorithm [4].

The motivation for this work is threefold. First, even with the best intentions, it is easy to skip reasoning steps in paper proofs, or to overlook non-trivial properties. Parallel and distributed algorithms are by nature difficult to verify, and we felt that a mechanical proof would help us in understanding the algorithm deeply. Second, the
proof assistant Coq requires constructive proofs, which forced us not only to state properties, but also to provide a mechanical way to derive them. Such an exercise has proved to be successful because we managed to specify very precisely the notions of alternate queue and diffusion tree, which are central to the proof of safety. Third, we see this work as part of a larger activity aiming to certify distributed software systems; the hope is that our formalisation may be reused as a module for more complex systems.

The source code for the proof in Coq is available from [30]. The proof is about 13000 lines long, plus an extra 3000 lines for algorithm variants. We present here a selection of definitions, lemmas and theorems, in a notation that is very close to the one in our Coq proof. For the sake of conciseness, proofs are only sketched, but complete proof details may be obtained from [30].

This paper is organised as follows. First, we set the context in which the algorithm was developed and present its intuition (Section 2). The algorithm is then formally described as an abstract machine, which we call the DRC-machine (Section 3). General properties of the machine are defined, including some basic invariants and a notion of diffusion tree that represents the path by which references are propagated in a computation (Section 4). Correctness is established, involving both safety and liveness aspects (Section 6). Then, optimisations and algorithm variants are investigated (Sections 6 and 7). Finally, we conclude the paper with related work.

2 The Algorithm: Informal Presentation

The initial motivation for this work was the design and implementation of a distributed language [29], based on the message-passing library Nexus [10]. This library essentially provides a notion of global pointer (GP), which is a reference to a remote object, and a form of remote procedure call, which allows the programmer to activate a computation on an object pointed at by a GP; any data, including global pointers, may be passed as argument to a remote procedure call.

We assume that several locations participate to a computation and we call them sites. During the course of a computation, GPs are created and communicated by remote procedure calls. The site where a GP is initially created is called its owner; the owner contains some data that a GP is referring to. Newly created global pointers must be unique; in practice, a global pointer contains a unique address representing its host and a locally unique identifier. In this algorithm, we
adopt the following failure assumptions: there exists a reliable message delivery, i.e. messages cannot be lost, corrupted or duplicated; machines never crash and are never taken out of service; there is trust across the entire domain.

The purpose of distributed reference counting is to keep track of the different GPs. More precisely, each GP will be associated with a reference counter. On a GP’s owner, a reference counter is expected to be strictly positive whenever a copy of the GP is accessible remotely.

We use tables to maintain associations between counters and global pointers that were sent to remote sites. We call these tables send-tables as they are used whenever GPs are sent remotely. Each site contains such a send-table.

---

**Fig. 1** Copying and Deleting a Reference

Let us consider two sites $s_1$ and $s_2$, some data on $s_1$, and a global pointer $GP$ pointing at this data. Initially, the counter of $GP$ is set to zero in the send-table of $s_1$. Every time a $GP$ is sent to a remote site, its associated counter is incremented by one. The reader will note that reference counters are used for counting references between sites; other mechanisms may be used for counting references locally.

The middle picture in Figure 1 shows that copying $GP$ has increased its reference counter in the send-table of $s_1$. To a first approximation, the send-table indicates the number of times a global pointer was sent remotely. The middle picture indicates that a copy of $GP$ is accessible on $s_2$ and the send-table on $s_1$ is strictly positive.

Knowing that sending, propagating and receiving a message are events that do not occur simultaneously, we adopt the following con-
ventions. Each picture represents a snapshot of the system, at a given point in time. A bold arrow from \( s_1 \) to \( s_2 \) indicates that a message \( s_1 \) sent by \( s_1 \) and received by \( s_2 \); the snapshot represents the state of the system after the message has been received and processed.

In order to keep reference counters up to date, each site has to be able to determine whether a \( GP \) has already been received. For this purpose, each site maintains a second table, called \( \text{receive-table}^1 \), which contains the global pointers that have already been received. By construction, a \( GP \) belongs to its owner’s receive table. According to the middle picture of Figure 1, \( GP \) is in the receive-tables of both \( s_1 \) (its owner) and \( s_2 \).

In addition to reference counters, the distributed reference counting algorithm uses \( \text{control messages} \), whose purpose is to update counters. A \( \text{decrement message} \) is aimed at a site and contains a global pointer \( GP \). When the destination site receives such a message, it decrements the counter associated with \( GP \) in its send-table; if the counter reaches 0, the object associated with the pointer is then unreferenced by remote sites.

We use decrement messages in two different situations. First, when a \( GP \) is no longer needed by a site, \( GP \) is removed from the receive table and a decrement message is sent to \( GP \)'s owner. In Figure 1, as soon as \( GP \) is unneeded on \( s_2 \), a decrement message is sent to \( s_1 \), which in the present case has the effect of resetting its counter in the send-table of \( s_1 \). A \( GP \) can be declared unneeded on a site if it is not required by the local computation and its associated counter in the send-table is null.

Second, when a \( GP \) is received by a site that already owns a copy of the \( GP \) (as indicated by its receive table), a decrement message has to be sent back to the emitter so as to maintain accurate reference counters. Now, we can refine the counter description: a counter in a send-table represents the number of \textit{different} remote copies of a \( GP \) plus the number of messages related to it in transit.

Let us now consider three sites. Figure 2 illustrates a scenario that follows the middle picture of Figure 1, where \( GP \) has been copied from \( s_2 \) to \( s_3 \). Using the same principle, the counter for \( GP \) on \( s_1 \) and \( s_2 \) has a value 1, and the \( GP \) is also in the receive-tables of \( s_2 \) and \( s_3 \).

In fact, the mechanism we describe here bears some resemblance with Indirect Reference Counting [32], where the sum of reference

---

1 We call our tables \( \text{send} \) and \( \text{receive} \) because they are used when sending or receiving global pointers, respectively. Other names may be found in the literature: entry and exit items [21,33], scions and stubs [34], or Incoming and Outgoing reference tables [9].
counters across the diffusion tree of a \( GP \) is the number of its remote copies. The analogy does not extend further because decrement messages are used differently.

Let us recall that, when a \( GP \) is no longer needed, a message is sent to its owner. This design decision is motivated by the fact that a Nexus GP only refers to its owner site, and has no information about the sites it transited by. Unfortunately, untimely decrement messages may be the consequence as illustrated in Figure 3. If \( s_3 \), which received \( GP \), deletes its reference to \( GP \), then \( s_3 \) sends a decrement message to \( s_1 \), that is, the \( GP \)'s owner. The effect of the decrement message is to reset the reference counter on \( s_1 \). This clearly results in an inconsistent situation as \( GP \) may still be active on \( s_2 \), while the reference counter on \( s_1 \) is null.

Besides the incorrectness related to the decrement message, such an indirect reference counter technique may keep some pointers active longer than expected; in other words, this results in a form of memory leak. Indeed, \( GP \) remains needed by \( s_2 \) in Figure 2 because the counter for \( GP \) in \( s_2 \) send-table is not null, even if the local computation does not use this pointer any longer.

Our solution to both the untimely arrival of messages and memory leaks involves a new type of message, called increment-decrement, written inc_dec. An increment-decrement message involves three different sites: \( s_1, s_2, s_3 \), respectively, the owner, the emitter and the receiver of a \( GP \). When \( GP \) reaches the receiver for the first time, an increment-decrement message is sent to its owner. When the owner \( s_1 \) receives an increment-decrement message, it increments \( GP \)'s reference counter, and then sends a decrement message to the emitter \( s_2 \).
Fig. 3 Untimely Decrement

considering GP (Figures 4 and 5). The increment-decrement message can be seen as a form of registration, which has to be performed the first time a GP is received; as a consequence, this allows the owner to be aware of all the sites that have received copies of a GP.

Fig. 4 Diffusion Tree Reorganization (1)

Introducing the increment-decrement message is not sufficient to avoid untimely message arrivals. The increment-decrement message from the receiver $s_3$ should arrive at the owner $s_1$ before any decrement message from the receiver $s_3$ about the same GP. This can be enforced by adding a further constraint, in the form of FIFO trans-
mission of messages. We therefore assume in-order message delivery of messages between any pair of sites (in Sections 6 and 7, we discuss how such a constraint may be partially relaxed).

In Figure 5, we can observe that if GP is no longer needed on s2, its owner s1 may be informed by a dec message. Such a property is particularly important in the presence of mobile computations jumping from sites to sites. The diffusion tree reorganisation provided by the increment-decrement message prevents the formation of chains of pointers abandoned by mobile computations.

**Remark** We have presented distributed reference counting as a general purpose technique. It may be used to implement a distributed garbage collector. The send-table must be defined as a root of the local garbage collector. A GP will be entered in a send-table only if its counter is strictly positive. As a result, by its presence in the send-table, GP remains reachable from the local collector roots, which ensures that the space used by the data referenced by GP cannot be reclaimed. As soon as a reference counter reaches zero, its entry may safely be removed from the send-table. In contrast, the receive table must not be defined as a root of local collector.

3 The Algorithm: The DRC-Machine

Let us now present the algorithm, following our encoding in the Coq proof assistant [30]. (A tutorial is available from [16].) The algorithm is formalised by an abstract machine, called the DRC-machine, whose
state space is displayed in Figure 6. In the DRC-machine, we only model messages exchanged by the distributed reference counting algorithm, and we do not model any form of computation which it would be used in.

\[
\begin{align*}
S &= \{s_0, s_1, \ldots, s_n\} & \text{(Set of Sites)} \\
G &= \{gp_0, gp_1, \ldots, gp_m\} & \text{(Set of Global Pointers)} \\
M &= \text{copy} : G \to M \mid \text{dec} : G \to M & \text{(Set of Messages)} \\
K &= S \times S \to \text{Queue}(M) & \text{(Set of Message Queues)} \\
ST &= S \times G \to \mathbb{Z} & \text{(Set of Send Tables)} \\
RT &= S \times G \to \text{Bool} & \text{(Set of Receive Tables)} \\
C &= ST \times RT \times K & \text{(Set of Configurations)}
\end{align*}
\]

Characteristic variables:
\[s \in S, \ GP \in G, \ m \in M, \ k \in K, \ send \mathcal{T} \in ST, \ rec \mathcal{T} \in RT, \ c \in C\]

**Fig. 6** State Space of the DRC-machine

A finite number of sites are involved in a DRC-machine, and we consider a finite number of global pointers. The set of messages is defined by an inductive type, whose three constructors are named according to the messages presented in Section 2, namely \text{copy}, \text{dec} and \text{inc\_dec}. Communication channels are represented by queues of messages between pairs of sites. We use the following notations and operations on queues:

- \(q, q_1, \ldots\): denote queues;
- \(\emptyset\): the empty queue;
- \(\text{first}(q)\): head of a non-empty queue \(q\);
- \(\text{tail}(q)\): non-empty queue \(q\) except its head;
- \(q \oplus \{m\}\): queue \(q\) after adding a message \(m\) at its tail;
- \(q_1 \oplus q_2\): queue obtained after concatenating \(q_1\) and \(q_2\).

Send and Receive Tables are represented by functions associating sites and global pointers with numbers or booleans, respectively. Counters are represented by integers; we shall establish that counters are always positive or null. A DRC-configuration is given by a tuple of send tables, receive tables, and message queues. This abstract machine is a suitable abstraction of a distributed system as send-tables and receive-tables may easily be distributed across several sites.

We assume that each \(GP\) has been created on a site (and associated with some local data). This site is called the \(GP\)'s owner. We
define a function

\[ \text{owner} : GP \rightarrow \text{Site}, \]

which maps each global pointer onto its owner site.

The distributed reference counting algorithm itself is encoded by transitions of the DRC-machine displayed in Figure 7. Transitions are defined as inductive types, whose constructors are \texttt{make\_copy}, \texttt{receive\_copy}, \texttt{receive\_inc\_dec}, \texttt{receive\_dec} and \texttt{delete}. A transition function maps a configuration \( c \) and a transition \( t \) to a new configuration \( c' \):

\[ c \mapsto^t c', \]

where \( t \) is any of the five permitted transitions. In a concise form, Figure 7 displays the definitions of transitions and the transition function. We used some notations such as \textit{post}, \textit{receive} or table updates, which give an imperative look to the algorithm, and whose definitions are as follows.

- \( \text{send}_T(s, GP) := V \) denotes \( \langle \text{send}_T', \text{rec}_T, k \rangle \), such that \( \text{send}_T'(s, GP) = V \) and \( \text{send}_T'(s, GP') = \text{send}_T(s, GP') \) for any \( GP' \neq GP \).
- \( \text{rec}_T(s, GP) := V \) is similar.
- \( \text{post}(s_1, s_2, m) \) denotes \( \langle \text{send}_T, \text{rec}_T, k' \rangle \), with \( k'(s_1, s_2) = k(s_1, s_2) \cap \{m\} \), and \( k'(s_i, s_j) = k(s_i, s_j), \forall (s_i, s_j) \neq (s_1, s_2) \).
- \( \text{receive}(s_1, s_2) \) denotes \( \langle \text{send}_T, \text{rec}_T, k' \rangle \), with \( k'(s_1, s_2) = \text{tail}(k(s_1, s_2)), \) and \( k'(s_i, s_j) = k(s_i, s_j), \forall (s_i, s_j) \neq (s_1, s_2) \).

In each rule of Figure 7, the conditions that appear to the left hand side of an arrow are guards that must be satisfied in order to perform the transition. The right-hand side denotes the configuration that is reached after transition.

The first transition denotes the transition that is performed when a GP is copied from \( s_1 \) to \( s_2 \). We assume here that the two sites are different. Furthermore, it is a requirement for \( s_1 \) to “have access” to \( GP \), otherwise sending \( GP \) to \( s_2 \) would be impossible; such a condition is modeled by \( GP \)'s presence in the receive-table of \( s_1 \).

The resulting configuration sees the send-table of \( s_1 \) increased and a message \texttt{copy} sent between \( s_1 \) and \( s_2 \).

The second transition is concerned with \( s_2 \) handling an incoming \texttt{copy}(\( GP \)) message from \( s_1 \). The following cases are possible: (i) If \( s_2 \) has access to the global pointer \( GP \), i.e. \( GP \) is present in \( s_2 \) receive-table, then a \texttt{dec} message is sent back to the emitter \( s_1 \). (ii) Otherwise, \( s_2 \) receive table is set to true; furthermore, if \( s_1 \) and \( s_2 \) are different from the owner, then an \texttt{inc\_dec} message should be sent
Given a configuration \( c = (send_T, rec_T, k) \), five transitions are permitted:

\[
\text{make\_copy}(s_1, s_2, GP):
\]

\[
s_1 \neq s_2 \land rec_T(s_1, GP)
\rightarrow \begin{cases}
\text{send}_T(s_1, GP) := send_T(s_1, GP) + 1 \\
\text{post}(s_1, s_2, \text{copy}(GP))
\end{cases}
\]

\[
\text{receive\_copy}(s_1, s_2, GP):
\]

\[
first(k(s_1, s_2)) = \text{copy}(GP)
\rightarrow \begin{cases}
\text{receive}(s_1, s_2) \\
\text{if } rec_T(s_2, GP) \text{ then } \\
\{ \text{post}(s_2, s_1, \text{dec}(GP)) \} \\
\text{else} \\
\{ \text{rec}_T(s_2, GP) := \text{true} \\
\text{post}(s_2, \text{owner}(GP), \text{inc\_dec}(GP, s_1)) \text{ if } s_1, s_2 \neq \text{owner}(GP) \}
\end{cases}
\]

\[
\text{receive\_inc\_dec}(s_1, s_2, GP, s_3):
\]

\[
first(k(s_1, s_2)) = \text{inc\_dec}(GP, s_3)
\rightarrow \begin{cases}
\text{receive}(s_1, s_2) \\
\text{send}_T(s_2, GP) := send_T(s_2, GP) + 1 \\
\text{post}(s_2, s_3, \text{dec}(GP))
\end{cases}
\]

\[
\text{receive\_dec}(s_1, s_2, GP):
\]

\[
first(k(s_1, s_2)) = \text{dec}(GP)
\rightarrow \begin{cases}
\text{receive}(s_1, s_2) \\
\text{send}_T(s_2, GP) := send_T(s_2, GP) - 1
\end{cases}
\]

\[
\text{delete}(s, GP):
\]

\[
send_T(s, GP) = 0, \text{ rec}_T(s, GP), \text{ owner}(GP) \neq s
\rightarrow \begin{cases}
\text{rec}_T(s, GP) := \text{false} \\
\text{post}(s, \text{owner}(GP), \text{dec}(GP))
\end{cases}
\]

**Fig. 7** Transitions of the DRC-Machine
to the owner as displayed in Figure 4. Consequently, a necessary condition to send an inc, dec message is to have received a GP that is not locally accessible\(^2\). Let us note that the received message has been “consumed” and is no longer present in the resulting configuration.

The third transition deals with an incoming inc, dec(GP, s\(_3\)) message: the send-table is increased and a dec message is sent to site s\(_3\). The fourth transition reacts to an incoming dec message by decreasing the send-table for the concerned global pointer.

Deciding when a reference is lost is application dependent. For instance, a distributed garbage collector may use a local garbage collector to detect such an event; in distributed termination [40], the loss of a reference is triggered by the end of a local computation. As a result, we cannot model such criteria, but we can establish the conditions that must hold in the distributed reference counting algorithm when a reference is deleted, as formalised by the fifth transition. This transition is typically fired when the application decides to release a reference. It can only be fired if the site is not the GP’s owner, if the send-table is null and if the receive-table contains the GP. The transition sets the receive table to false and sends a dec message as in the right-hand side of Figure 1.

The initial configuration is defined as follows. Receive-tables contain false entries except for GP owners; Send-tables are set to 0; Communication channels are empty. Formally, the initial configuration \(c_i\) is defined by the tuple \((\text{recv}_i, \text{send}_i, K_i)\).

\[
\begin{align*}
\text{recv}_i = \lambda s \lambda GP \text{ if } (s = \text{owner}(GP)) \text{ then } true \text{ else } false \\
\text{send}_i = \lambda s_1 \lambda s_2 \lambda GP 0 \\
K_i = \lambda s_1 \lambda s_2.0
\end{align*}
\]

A configuration \(c\) is said to be legal if there is a sequence of transitions \(t_1, t_2, \ldots, t_n\) such that \(c\) is reachable from the initial configuration:

\[
c_i \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \ldots \xrightarrow{t_n} c.
\]

4 Algorithm Properties

Our goal is to prove the correctness of the distributed reference counting algorithm, which has two different facets. Safety is the property

\(^2\) Note that the decision of sending an inc, dec message is based on the accessibility of the GP at the time a copy message is received, independently of the previous history. A site, different from the owner, may therefore receive a first copy of a GP, delete the reference, and then receive a second copy, which may be followed by an inc, dec message.
according to which the reference counter of a GP on its owner is guaranteed to be strictly positive whenever a copy of the GP is available on a remote site. Liveness is the property that guarantees that if all references to a global pointer are deleted, the owner’s send-table will eventually become null.

These properties will be established in Section 5, but beforehand we present some general properties of the algorithm. First, we establish some invariants relating send-tables, receive-tables and messages in transit. Second, we analyse the use of inc, dec messages, which are only found on channels aimed at a GP’s owner; we show that these channels have a regular structure. Third, we investigate the notion of diffusion tree, which is, we previously claimed, reorganised by the inc, dec message.

4.1 Invariants

Messages may be assigned a weight, as a measure of their overall absolute effect on reference counters. We assign 1 to dec and copy messages because their effect is respectively to decrease or increase counters. On the other hand, we assign a null weight to an inc, dec message, because it increases the owner’s send-table, but it is followed by a dec message which decreases another counter.

\[
\begin{align*}
\text{Weight}(\text{dec}(GP)) &= 1 \\
\text{Weight}(\text{copy}(GP)) &= 1 \\
\text{Weight}(\text{inc, dec}(GP, s)) &= 0
\end{align*}
\]

Similarly, we can convert the boolean value stored in a receive table into an integer.

\[
\begin{align*}
\text{INT}(\text{true}) &= 1 \\
\text{INT}(\text{false}) &= 0
\end{align*}
\]

The first invariant establishes that the counters stored in send-tables are directly linked to receive table values and the weight of messages in transit.

**Lemma 1.** Let \( c = (\text{send}, \text{rec}, k) \) be a legal configuration. The following equality holds. For any GP \( \in G \):

\[
\begin{align*}
\sum_{s_i \in S} \text{send}_{T}(s_i, GP) &= \sum_{s_i \in S} \text{INT}(\text{rec}_{T}(s_i, GP)) - 1 \\
&+ \sum_{m \in K_{GP}} \text{Weight}(m),
\end{align*}
\]
where \( m \in K \downarrow GP \) denotes the set of messages in \( K \) that are related to \( GP \).

**Proof.** The detailed proof appears in file invariant1.v. It uses an induction on legal transitions and a case analysis on the different types of transitions. \( \square \)

---

**Fig. 8** Messages Under Control of \( s_2 \)

The second invariant defines the value of reference counters on sites that differ from the owner. In Figure 8, we identify messages that update the send-tables of \( s_2 \), or which result from a change in the send-table of \( s_2 \). Indeed, the send-table of \( s_2 \) is increased every time a \( \text{copy}(GP) \) message is sent to a remote site; such a \( \text{copy} \) message may be followed by a \( \text{dec} \) message or an \( \text{inc \_ dec} \) message (towards the owner); the latter is followed by a \( \text{dec} \) message back to \( s_2 \). In reality, we have to consider all sites \( s_1 \) to which \( s_2 \) sends such \( \text{copy} \) messages.

**Definition 1.** Let \( k \) be a set of queues of a DRC-machine configuration. Let \( s_i \) be a site of \( S \). The set of messages under control of \( s_i \), written \( \text{control}(GP, s_i) \), is defined as:

\[
\text{control}(GP, s_i) = \{ m \mid \begin{array}{l}
m = \text{copy}(GP), m \in k(s_i, s_j) \\
m = \text{dec}(GP), m \in k(s_j, s_i) \text{ or} \\
m = \text{inc \_ dec}(GP, s_i), m \in k(s_k, s_j) \\
\end{array}\}
\]

for any \( s_j, s_k \).

The second lemma is stated as follows: the value of a send-table on a site \( s_i \) that differs from the owner is given by the number of messages under control of \( s_i \).
Lemma 2. Let \( \langle \text{send}_T, \text{rec}_T, k \rangle \) be a legal DRC configuration. The following property holds. For any \( GP \in G \), for any \( s_i \in S \) such that \( s_i \neq \text{owner}(GP) \):
\[
\text{send}_T(s_i, GP) = \#(\text{control}(GP, s_i)),
\]
where \# denotes the cardinality of a set.

Proof. The equality is initially true and is preserved by each transition. The case analysis is available in file invariant.2.v. □

Both invariants may be combined together in order to obtain the value of the owner’s send-table in terms of the messages in transit and receive-tables. We will then be able to derive the safety property by proving that the owner’s send-table is positive whenever a global pointer is accessible remotely. However, we need to establish further properties about contents of messages queues with inc, dec messages and the notion of diffusion tree.

4.2 Alternate Queues

A message inc, dec is sent if a site \( s \) receives a message copy(GP) and the receive table for the GP is empty on \( s \). Site \( s \) will send again an inc, dec message only after it has performed a delete transition, which cleared the receive-table for that GP. Consequently, we can find two messages inc, dec(GP, si) and inc, dec(GP, sj) in a same queue only if there is (at least) one dec message between them.

We characterise such a behaviour by the notion of alternate queue, which specifies how inc, dec and dec messages must be interleaved.

Definition 2 (Alternate). An alternate queue for a given GP is defined inductively as follows:

- \( q \) is alternate for GP if it does not contain messages related to GP;
- \( q \parallel \{ \text{inc, dec}(GP, s) \} \) is alternate for GP if \( q \) does not contain messages related to GP;
- if \( q \) is alternate for GP, so is \( q \parallel \{ m \} \) provided that \( m \) is not an inc, dec message related to GP;
- if \( q \) is alternate for GP, so is \( q \parallel \{ \text{dec}(GP) \} \parallel q_1 \parallel \{ \text{inc, dec}(GP, s) \} \), provided that \( q_1 \) is a queue of messages not related to GP.

We can prove that any queue of messages between a site and a GP’s owner is alternate.
Lemma 3. Let \( \langle \text{send}_T, \text{rec}_T, k \rangle \) be a legal DRC-configuration. For any \( GP \in \mathcal{G} \) and for any \( s \in S, \quad k(s, \text{owner}(GP)) \) is alternate for \( GP \).

Proof. The proof appears in file invariant5.v and proceeds by induction on the legal transitions, and by a case analysis of the different transitions. \( \square \)

4.3 Diffusion Tree

In a distributed application, global pointers are exchanged between sites taking part in the computation. Such operations are modeled by copy messages in the DRC-machine. One can easily derive a graph structure whose nodes are sites and edges represent the presence of a copy message between two sites.

However, our motivation is not so much about understanding where copy messages are sent to, which is application-specific, but to investigate the role of inc\_dec messages in the algorithm. An inc\_dec message indicates that a site \( s \) has received a new global pointer, i.e. \( s \) has received a global pointer that was not accessible on \( s \). From this idea, we can derive a notion of diffusion tree, which formalises the path taken by global pointers to reach new sites.

We define the root of the diffusion tree as the owner of a global pointer. A direct child is a site that receives a new global pointer \( GP \), directly from its owner. An indirect child is a site that receives a new \( GP \) from a site different from its owner. According to the algorithm, as soon as an indirect child receives a new \( GP \), an inc\_dec message is posted to its owner.

We can therefore define a relation \( \text{diffuse}(c, GP, s_1, s_2) \), read as \( s_1 \) has diffused \( GP \) to \( s_2 \) in configuration \( c \), indicating that \( s_2 \) has received the new \( GP \) from \( s_1 \).

Definition 3 (Diffuse). Given a configuration \( c \) and a \( GP \), \( \text{diffuse}(c, GP, s_1, s_2) \) holds if \( \text{rec}_T(s_2, GP) = \text{true} \) and the last inc\_dec message related to \( GP \) in the queue between \( s_2 \) and \( \text{owner}(GP) \) is inc\_dec\((GP, s_1)\).

Let us note that we could find several inc\_dec messages for a given \( GP \) in a given queue, but the diffuse relation is defined by the most recent inc\_dec message for the \( GP \) that was posted in that queue. The relation diffuse changes over time as inc\_dec messages are processed or new inc\_dec messages are generated. We can now formally define direct and indirect children.
Definition 4 (Indirect Child). Given a configuration \( c \), a global pointer \( GP \), a site \( s_2 \) is an indirect child if there is a site \( s_1 \) such that \( \text{diffuse}(c, GP, s_1, s_2) \) holds.

Definition 5 (Direct Child). A site \( s \) that has access to a \( GP \) is a direct child if there is no \( s_i \) such that \( \text{diffuse}(c, GP, s_i, s) \) holds.

We define an ancestor as the transitive closure of the relation \( \text{diffuse} \). An important property of the ancestor relation is its non-reflexivity, which ensures that this relation may be used to define a tree, and will not result in a graph.

Lemma 4 (Not Reflexive). For any legal configuration \( c \), for any global pointer \( GP \), and for any sites \( s_1, s_2 \), if \( \text{ancestor}(c, GP, s_1, s_2) \), then \( s_1 \neq s_2 \).

Proof. The proof, available in invariant6.v, proceeds by induction on the legal transitions and by case analysis on the different kinds of transitions. \( \Box \)

---

Fig. 9 Diffusion Tree Reorganisation

In the left-hand side of Figure 9, \( GP \) was diffused from \( s_1 \) to \( s_2 \), as visualised by the \text{inc\_dec} message towards the \( GP \)'s owner. The effect of an \text{inc\_dec} message is to “register” a site that has received a new \( GP \). As soon as the \text{inc\_dec} message is received by the owner, \( s_2 \) becomes a direct child, as described in the right-hand side of Figure 9.
When all \texttt{inc,dec} messages have been processed, all sites will be direct children. The effect of the \texttt{inc,dec} message is therefore to flatten the diffusion tree.

More importantly for our proof, we can prove that for any site, one can find an ancestor that is a direct child.

**Lemma 5.** For any legal configuration \(c\), any global pointer \(GP\), and any site \(s\), if \(s\) is an indirect child of \(GP\)'s owner, then there exists a site \(s_1\) such that \(s_1\) is a direct child and \(s_1\) is an ancestor of \(s\).

**Proof.** This is a long proof by induction on the legal transitions and by case on the possible transitions. In particular, the transitions that produce or consume \texttt{inc,dec} messages have the ability to change the diffusion tree; they need a careful case analysis. The proof also relies on Lemma 4 to guarantee that we deal with a tree and not a graph. \(\square\)

Intuitively this Lemma specifies that if a site \(s\) receives a new \(GP\) from a site that is not the owner, this global pointer had to be propagated from a site \(s_1\) that is a direct child of the owner.

5 Correctness

We are now ready to establish the safety and liveness of the algorithm.

5.1 Safety

The safety property guarantees that the reference counter of a \(GP\) on its owner is strictly positive if \(GP\) is accessible remotely. A \(GP\) is said to be accessible on a site if it is present in a site's receive-table or if it is present in a \texttt{copy} message in transit.

It is now rather straightforward to derive the safety property. Substituting Lemma 2 into Lemma 1, we obtain the value of the owner's send-table.

**Lemma 6.** Let \(\langle \text{send}\_T, \text{rec}\_T, k \rangle\) be a legal DRC-configuration. The following property holds:

\[
\text{send}\_T(\text{owner}(GP), GP) = \sum_{s_i \in S, s_i \neq \text{owner}(GP)} \text{site}\_\text{weight}(s_i, GP)
\]

with \(\text{site}\_\text{weight}\) defined as:

\[
\text{site}\_\text{weight}(s_i, GP)
\]
= \{ \text{INT}(\text{rec}_T(s_i, \text{GP})) \\
+ \#(\{m \mid m = \text{copy}(\text{GP}), m \in k(\text{owner}(\text{GP}), s_i)\}) \\
+ \#(\{m \mid m = \text{dec}(\text{GP}), m \in k(s_i, \text{owner}(\text{GP}))\}) \\
- \#(\{m \mid m = \text{inc}\_\text{dec}(\text{GP}, s_i), m \in k(s_j, \text{owner}(\text{GP})), \forall s_j\}) \}\).

\textbf{Proof.} The proof can be found in file \texttt{invariant4.v}. It is immediately derived from Lemmas 1 and 2. \hfill \Box

We can see that the owner’s send-table depends on the number of remote sites that have access to the pointer, on the number of \texttt{copy} messages leaving the owner, on the number of \texttt{dec} messages aimed to the owner, and on the number of \texttt{inc}\_\texttt{dec} messages in transit.

Lemma 3 established that every queue \(k(s_i, \text{owner}(\text{GP}))\) is alternate for \(\text{GP}\). It follows that the owner send-table is always positive or null.

\textbf{Lemma 7.} Let \(\langle \text{send}\_T, \text{rec}_T, k \rangle\) be a legal DRC-configuration. For any global pointer \(\text{GP}\), \(\text{send}\_T(\text{owner}(\text{GP}), \text{GP}) \geq 0\).

\textbf{Proof.} The proof appears in file \texttt{invariant5.v}. Lemma 6 defines the owner’s send-table value as a sum, for which we prove here that each summand is positive or null. Using Lemma 3, we can derive that the number of \texttt{inc}\_\texttt{dec} messages in a queue \(k(s_i, \text{owner}(\text{GP}))\) is at most equal to the number of \texttt{dec} messages plus 1. Furthermore, it is at most equal to the number of \texttt{dec} messages when \(\text{rec}_T(s_i, \text{GP})\) is false. We therefore conclude that \texttt{site\_weight} is always positive or null. \hfill \Box

We are now ready to establish the safety property.

\textbf{Theorem 1 (Safety).} Let \(\langle \text{send}\_T, \text{rec}_T, k \rangle\) be a legal DRC-configuration.

\[\forall \text{GP} \in \mathcal{G}, \text{let } s = \text{owner} (\text{GP}), \forall s_i \neq s,\]

\[\text{if } \text{rec}_T(s_i, \text{GP}), \text{ then } \text{send}\_T(s, \text{GP}) > 0.\]

\textbf{Proof.} The proof of this theorem may be found in file \texttt{invariant8.v}. First, \(\text{site\_weight}(s_i, \text{GP}) > 0\) for any site \(s_i\) that is a direct child; indeed, by definition, the receive-table of a direct child is true and there is no \texttt{inc}\_\texttt{dec} message in the queue \(k(s_i, \text{owner}(\text{GP}))\) of a direct child \(s_i\). From Lemma 7, we know that \texttt{site\_weight} is always positive or null. We therefore have to prove that, if there is a site \(s_i\) such that \(\text{rec}_T(s_i, \text{GP})\), then there exists at least one site that is a direct child. Using Lemma 5, we know that if \(s_i\) is an indirect child, there is a direct child, which concludes the proof. \hfill \Box
The purpose of the safety property is to guarantee that the owner’s send-table is strictly positive when a reference is available in the distributed system. Theorem 1 proved such a property when a GP is explicitly present in a site’s receive-table. We still have to consider the case where the reference is in transit in a copy message.

**Theorem 2 (Safety 2).** Let \( \langle \text{send}_T, \text{rec}_T, k \rangle \) be a legal DRC-configuration.

\[ \forall GP \in \mathcal{G}, \forall s_i, s_j \in \mathcal{S}, \]

if \( \text{copy}(GP) \in k(s_i, s_j) \), then \( \text{send}_T(\text{owner}(GP), GP) > 0 \).

**Proof.** The proof of this theorem may be found in file \texttt{v8.invariant}. We can prove that if a copy message is in transit between two sites \( s_i \) and \( s_j \), then the send-table of \( s_i \) is strictly positive, which implies that its receive table is also true. Using Theorem 1, we conclude that the owner’s send table is strictly positive. \( \square \)

### 5.2 Liveness

Liveness guarantees that if all references to a GP are deleted, its owner’s send table will eventually become null. In order to establish liveness, we first show that whenever there is a message in a queue, a transition may be fired to consume this message.

**Lemma 8.** Let \( c \) be a legal configuration \( \langle \text{send}_T, \text{rec}_T, k \rangle \), such that \( k(s_1, s_2) = \{m\}^q \), for some \( m, s_1, s_2 \) and \( q \). Then, there exist a transition \( t \) and a configuration \( c' = \langle \text{send}_T', \text{rec}_T', k' \rangle \) such that \( c \rightarrow_t c' \), with \( k'(s_1, s_2) = q \).

**Proof.** The proof appears in file \texttt{liveness.v}. It proceeds by case analysis on the type of the message \( m \) known to be in a queue. \( \square \)

Lemma 8 ensures that the algorithm itself does not prevent the processing of messages.

Our next step is to prove that the distributed reference counting activity generates a finite number of transitions. We however need to be very clear about what we mean by distributed reference counting activity. The transition \texttt{make_copy} is initiated by the application, which is beyond this algorithm. So, we show that there can only be a finite number of transitions that do not involve a transition \texttt{make_copy}.

For this purpose, we introduce a new measure, called termination measure, which gives an indication of how far the DRC-machine
is from terminating its transitions related to distributed reference counting. The termination measure is defined in terms of a measure of the receive table and a measure of messages.

**Definition 6 (Termination Measure).** The termination measure of a configuration \( c = (\text{send}_T, \text{recv}_T, k) \) is defined as:

\[
\text{termination measure}(c) = \sum_{GP \in S} \sum_{s \in S} \text{rt measure}(\text{recv}_T(s, GP)) \\
+ \sum_{s_i \in S, s_j \in S} \sum_{m \in k(s_i, s_j)} \text{msg measure}(m),
\]

with

\[
\text{msg measure}(\text{copy}(GP)) = 5 \quad \text{rt measure}(\text{true}) = 2 \\
\text{msg measure}(\text{inc}_\text{dec}(GP, s)) = 2 \quad \text{rt measure}(\text{false}) = 0 \\
\text{msg measure}(\text{dec}(GP)) = 1
\]

Intuitively, a copy message can update a receive table and create a new inc_dec message, which itself may create a new dec message. The termination measure of these events was designed in such a way that the measure of an event is bigger that the cumulative measure of causally dependent events.

**Lemma 9.** For any legal configurations \( c, c' \) and for any transition \( t \), such that \( c \xrightarrow{t} c' \), and \( t \neq \text{make}_\text{copy}(s_1, s_2, GP) \), the following inequality holds:

\[
0 \leq \text{termination measure}(c') < \text{termination measure}(c).
\]

**Proof.** The proof can be found in file liveness.v. It proceeds by an analysis of the different possible cases for transition \( t \). \( \square \)

Knowing that the termination measure is positive or null, and having proved that it decreases for every non make_copy transition, we can derive the following termination Lemma.

**Theorem 3 (Termination).** For any legal configuration, all transition paths that do not involve make_copy transitions terminate.

**Proof.** The proof appears in file liveness.v. Let us define a relation successor on the set of legal configurations; \( lc_2 \) is a successor of \( lc_1 \) if \( lc_2 \) is obtained from \( lc_1 \) by a transition that differs from make_copy. Using the termination measure (Definition 6) and the fact that it decreases (Lemma 9), we can establish that the successor relation is well-founded. Therefore, we can derive that, for any legal configuration, there exists a configuration without successor, fixed point of the successor relation, which concludes the proof. \( \square \)
Let us consider a given global pointer $GP$. Using Theorem 3 and Lemma 6, a terminal state of the DRC-machine does not contain any message related to $GP$, which implies that the owner’s send-table value is equal to the number of remote sites that have a receive-table set to true. In addition, if all sites have fired the delete transition, because the global pointer was no longer needed, the owner’s send-table becomes zero. Consequently, if we assume fairness [26] of message delivery, and if all references to a $GP$ are lost, then its owner’s send-table becomes null, which proves liveness of our algorithm.

6 Local Optimisations

In this section, we present two local optimisations, which give new insights to the algorithm. The first optimisation relaxes the FIFO constraint for copy messages, whereas the second optimisation shows that our algorithm describes a family of distributed reference counting, including Piquier’s Indirect Reference Counting [32].

6.1 Unordered Copy Messages

The distributed reference counting algorithm was formalised by an abstract machine, which assumes FIFO communication queues between any pair of sites. We relied on such a property to characterise the regular structure of a queue between a site and a $GP$’s owner (Definition 2). In addition, we know that if dec messages were allowed to overtake inc_dec messages, send-tables may prematurely be decremented, which would break the safety property.

However, copy messages have a different nature than dec and inc_dec messages. A copy message represents the application activity which communicates references to remote sites, for instance through remote procedure call, whereas the latter messages represent real distributed reference counting activity.

Depending on the specific need of the application, it may be of primary importance to process application messages faster than distributed reference counting messages. For instance, it is generally admitted that garbage collection activity should not slow down the mandatory application.

The FIFO handling of messages forces the distributed reference counting activity to proceed synchronously with the application. As such a behaviour may not be acceptable to some applications, we investigate here the possibility of decoupling copy messages from the rest of the reference counting activity. We could re-design the abstract
machine and introduce queues whose specific purpose is to transport \texttt{copy} messages. Instead, we prefer to introduce a new rule that allows any \texttt{copy} message to be propagated individually by any strategy.

\[
\text{propagate\_copy}(s_1, s_2, GP, q_1, q_2, q_3, q_4) : \\
k(s_1, s_2) = q_1 \xi \{\text{copy}(GP)\} \xi q_2 \land q_1 \xi q_2 = q_3 \xi q_1 \\
\rightarrow \{ k(s_1, s_2) := q_3 \xi \{\text{copy}(GP)\} \xi q_4 \}
\]

Rule \text{propagate\_copy} should be read as follows. If there is a \texttt{copy} message between two sites \(s_1\) and \(s_2\) with \(q_1\) and \(q_2\) the sequences of messages respectively preceding and following the \texttt{copy} message, the message \texttt{copy} may be positioned at any location in the queue between \(s_1\) and \(s_2\); the concatenation of \(q_3\) and \(q_4\), the sequences of messages respectively preceding and following the \texttt{copy} message in the transformed queue, must be equal to the concatenation of \(q_1\) and \(q_2\).

Rule \text{propagate\_copy} allows any \texttt{copy} message appearing in a queue to be put at any other position in that queue, provided the order of the other messages remains unchanged. Such a transition allows \texttt{copy} messages to be processed at a different speed than other messages. Note that this transition is not intended to be easily implementable, but its purpose is to specify a range of possible behaviours for \texttt{copy} messages.

After adding a new transition to the abstract machine, all proofs that use an induction on the type of transition had to be extended to support the new case. No major difficulty was encountered, except for the alternate queues (Definition 2). The definition had to be revised so that \texttt{copy} messages may be allowed at any position.

**Definition 7 (Alternate 2).** An alternate queue is defined inductively as follows:

- \(q\) is alternate for \(GP\) if it does not contain messages related to \(GP\);
- if \(q\) is alternate for \(GP\), so is \(q \xi \{m\}\) provided that \(m\) is not an \texttt{inc\_dec} message related to \(GP\);
- if \(q\) is alternate for \(GP\), so is \(q \xi \{\text{inc\_dec}(GP, s)\}\), provided that there is a \texttt{dec}(GP) message after the last occurrence of an \texttt{inc\_dec} message related to \(GP\) in \(q\), if any.

We conjecture that other similar local optimisations may be proved. For instance, \texttt{inc\_dec} messages are allowed to overtake any message, or messages related to different \(GP\)s may be safely swapped.
6.2 Indirect Reference Counting

Let us consider a scenario where a copy message was received by $s_1$ from $s_2$, followed by $s_1$ posting an inc\_dec message to the owner; shortly afterwards, let us assume that $s_1$ deleted the global pointer reference, which resulted in a dec message from $s_1$ to the owner, immediately following the inc\_dec message. There is room for a local optimisation in such circumstances. Indeed, according to the current algorithm, the inc\_dec message would be delivered, would increase the owner’s send-table, would be followed by a dec message that would decrease the send-table on $s_2$; on the other hand, the other dec message would decrease the owner’s send-table. In other words, the net effect of these three messages is to decrease the send-table of $s_2$.

A similar effect may be achieved by a single dec message from $s_1$ to $s_2$ directly. This optimisation may be formalised by a new transition rule.

$$\text{redirect\_inc}(s_1, s_2, GP, q_1):$$
$$k(s_1, owner(GP)) = q_1 \; \& \; \{\text{inc\_dec}(GP, s_2) \; \& \; \{\text{dec}(GP)\}\}$$
$$\rightarrow \{ k(s_1, owner(GP)) := q_1;$$
$$k(s_1, s_2) := k(s_1, s_2) \; \& \; \{\text{dec}(GP)\} \}$$

The new rule satisfies the invariants formalised in Lemmas 1 and 2; furthermore, it is also safe because the safety Theorems 1 and 2 are still valid. However, this innocent change in surface had quite a deep repercussion on the proof. Indeed, rule redirect\_inc potentially changes the diffusion tree as it consumes the last inc\_dec message of a queue. Rule redirect\_inc is unique in the algorithm because it extracts messages from the end of the queue and not its beginning.

In particular, Lemma 4, and consequently Lemma 5, could not be derived immediately in presence of the new rule. We had to generalise Definition 3 and introduce a notion of multiple diffusion.

**Definition 8 (Multiple Diffusion).** Given a legal configuration $c$ and a $GP$, the predicate $\text{multiple\_diffuse}(c, GP, s_1, s_2)$ holds if $\text{rec\_T}(s_2, GP) = true$ and there is a message $\text{inc\_dec}(GP, s_1)$ in the queue $k(s_2, owner(GP))$.

Definition 8 differs from Definition 3 because it regards all inc\_dec messages as indicators of the diffuse relationship, as opposed to the last one only. We define an $\text{multiple\_ancestor}$ as the transitive closure of the relation $\text{multiple\_diffuse}$. The $\text{multiple\_ancestor}$ relation is also non reflexive.
Lemma 10 (Not Reflexive 2). For any legal configuration c, for any sites $s_1, s_2$, if $\text{multiple\_ancestor}(c, GP, s_1, s_2)$, then $s_1 \neq s_2$.

Proof. Proof appears in file invariant6.v and proceeds by induction on the legal transitions and by case on the possible transitions. □

The multiple ancestor relation is a superset of the ancestor relation. Therefore, from Lemma 10, we can derive that Lemma 4 is still valid in the presence of rule redirect\_inc.

Let us observe again that redirect\_inc is not intended to be easily implementable, but its purpose is to specify a new behaviour of the abstract machine. Indeed, in terms of implementation, it seems difficult to redirect messages that were already sent.

More realistically, this rule may be implemented as follows. Instead of sending an inc\_dec message when a new GP is received, one can associate the GP with a “redirecting information”, containing the site that sent it. When a dec message has to be sent to the owner, it has to be redirected if some redirecting information is available.

In reality, such a systematic avoidance of inc\_dec messages is Piquer’s Indirect Reference Counting algorithm [32]. We can see our algorithm as an abstract specification of a family of distributed reference counting algorithms. At one end of the spectrum, we find Piquer’s Indirect Reference Counting (IRC) that does not use inc\_dec messages at all. At the other end of the spectrum, we find an algorithm that eagerly sends inc\_dec messages in order to flatten the diffusion tree. In between these extremes, there is a range of implementation strategies, which combine both IRC and diffusion tree flattening.

Indirect Reference Counting forces each parent to maintain a sendable entry for each global pointer passed to its children, until children have completely released the references to this pointer. This may result in “zombie pointers” [33], where the pointer is only kept live on a site because it is needed in a send-table. This in fact results in a form of memory leak, which may be avoided by the use of the inc\_dec message.

7 Algorithm Variants

In this Section, we consider two variants of the algorithm. (i) The first one handles messages to the owner differently, so that dec messages do not have to be sent back to the emitters. It is not an local optimisation as the ones described in Section 6, because it changes
some fundamental properties of the algorithm, which we discuss here. 

(ii) The second variant of the algorithm uses reference listing, which 
not only counts the number of times references are copied to remote 
sites, but also remembers the sites where the references were copied. 
Reference listing is a technique that is useful to assist in defining a 
fault-tolerant version of the algorithm.

7.1 No Copy to the Owner

A make_copy transition increases the emitter's send-table. If the copy 
message is emitted to the owner, it will be followed by a dec message 
back to the emitter, which will decrease its send-table. This scenario 
could be optimised: if we do not increment the send-table before 
sending a copy-message to the owner, we can avoid sending a dec 
message back to the emitter.

We have investigated this approach, which requires an extra pre-
condition in the guard of rule make_copy.

make_copy(s1, s2, GP) :
    s1 ≠ s2 ∧ s2 ≠ owner(GP) ∧ rec_T(s1, GP)
    ⇒ { send_T(s1, GP) := send_T(s1, GP) + 1
         post(s1, s2, copy(GP))  }

Rule make_copy may be fired only when s2 is different than the owner. 
Let us observe that this rule is more radical than the description we 
just gave. Indeed, in this algorithm, we no longer send copy messages 
to the owner at all. Let us remember that copy messages represent 
the information that must be communicated to our algorithm when 
references are copied between sites. The absence of copy messages 
to the owner does not prevent an implementation from performing 
remote procedure calls to the owner, but it simply indicates that no 
information has to be passed to the distributed reference counting 
module in such circumstances. We decided to adopt such a rule be-
cause it facilitates the proof; if we had accepted copy-messages to the 
owner without increasing the send-table, we would have had to in-
troduce a null weight for these messages, which would have required 
longer case analyses in the proofs.

The invariant Lemmas 1 and 2 and the safety Theorems 1 and 2 
are all valid for this algorithm, without any major difference in the 
proofs themselves.

Now that copy messages have disappeared from queues to the 
owner, rule propagate_copy will no longer be applicable for such queues.
However, \texttt{propagate\_copy} indicated that application messages carrying references did not have to be synchronised with distributed reference counting messages. This property is no longer valid with the current algorithm, and we give a counter example.

Let us consider two sites: the owner of a \texttt{GP} and \texttt{s}. Let us assume that the send-table of \texttt{s} is null. Site \texttt{s} sends a \texttt{copy} \texttt{GP} with a remote procedure call to the owner, and immediately afterwards deletes its reference of \texttt{GP}, which generates a \texttt{dec} message to the owner. If the remote procedure call is delayed, the \texttt{dec} message can decrease the owner’s send-table, which becomes null, whereas a reference is still in transit. Such a scenario would have been impossible in the original algorithm, because \texttt{s} had to increase its send table when sending the \texttt{copy} message, which prevented \texttt{s} to fire the \texttt{delete} transition.

It does not imply that this variant of the algorithm is less useful than the previous one. FIFO order must be strictly followed in order to preserve safety, and the application will dictate if such a constraint is acceptable. We conjecture that some asynchronism is still permitted; it is always safe to process a \texttt{copy} message early, because it increases reference counters; symmetrically, \texttt{dec} messages may be processed later.

\textit{7.2 Reference Listing}

In order to define a fault tolerant version of the algorithm, it is convenient to maintain not only a counter representing the number of times references were copied, but also the sites to which they were sent.

The state space has to be changed accordingly. Send-tables require an extra argument representing the site where a global pointer is sent to. In addition, \texttt{dec} and \texttt{inc\_dec} message constructors take one more argument, which is the site-entry of a send-table they operate on.

\[
\mathcal{M} = \text{copy}: \mathcal{G} \to \mathcal{M} \mid \text{dec}: \mathcal{G} \times S \to \mathcal{M} \quad \text{(Set of Messages)} \\
\mid \text{inc\_dec}: \mathcal{G} \times S \times S \to \mathcal{M} \\
\mathcal{ST} = S \times \mathcal{G} \times S \to \mathcal{Z} \quad \text{(Set of Send Tables)}
\]

Figure 10 displays the transitions. Rule \texttt{make\_copy} updates the table on site \texttt{s1}, for an entry identified by \texttt{GP} and \texttt{s2}. Similarly, rules \texttt{receive\_inc\_dec} and \texttt{receive\_dec} update the entry of a send-table indexed by the new site contained in the received message. Other changes are similar.

Lemma 2 may be refined for the reference listing algorithm, which requires us to update Definition 1.
Given a configuration \( c = (\text{send}_T, \text{rec}_T, k) \), five transitions are permitted:

\[
\text{make\_copy}(s_1, s_2, GP) :
\begin{align*}
& s_1 \neq s_2 \land \text{rec}_T(s_1, GP) \\
& \rightarrow \{ \text{send}_T(s_1, GP, s_2) := \text{send}_T(s_1, GP, s_2) + 1 \\
& \text{post}(s_1, s_2, \text{copy}(GP)) \}
\end{align*}
\]

\[
\text{receive\_copy}(s_1, s_2, GP) :
\begin{align*}
& \text{first}(k(s_1, s_2)) = \text{copy}(GP) \\
& \rightarrow \{ \text{receive}(s_1, s_2) \\
& \quad \text{if } \text{rec}_T(s_2, GP) \text{ then} \\
& \quad \{ \text{post}(s_2, s_1, \text{dec}(GP, s_2)) \} \\
& \quad \text{else} \\
& \quad \{ \text{rec}_T(s_2, GP) := \text{true} \\
& \quad \text{post}(s_2, \text{owner}(GP), \text{inc\_dec}(GP, s_1, s_2)) \\
& \quad \quad \text{if } s_1, s_2 \neq \text{owner}(GP) \} \}
\end{align*}
\]

\[
\text{receive\_inc\_dec}(s_1, s_2, GP, s_3, s_1) :
\begin{align*}
& \text{first}(k(s_1, s_2)) = \text{inc\_dec}(GP, s_3, s_4) \\
& \rightarrow \{ \text{receive}(s_1, s_2) \\
& \quad \text{send}_T(s_2, GP, s_1) := \text{send}_T(s_2, GP, s_1) + 1 \\
& \quad \text{post}(s_2, s_3, \text{dec}(GP, s_1)) \} 
\end{align*}
\]

\[
\text{receive\_dec}(s_1, s_2, GP, s_3) :
\begin{align*}
& \text{first}(k(s_1, s_2)) = \text{dec}(GP, s_3) \\
& \rightarrow \{ \text{receive}(s_1, s_2) \\
& \quad \text{send}_T(s_2, GP, s_3) := \text{send}_T(s_2, GP, s_3) - 1 \}
\end{align*}
\]

\[
\text{delete}(s, GP) :
\begin{align*}
& \forall s_j, \ \text{send}_T(s, GP, s_j) = 0, \ \text{rec}_T(s, GP), \text{owner}(GP) \neq s \\
& \rightarrow \{ \text{rec}_T(s, GP) := \text{false} \\
& \text{post}(s, \text{owner}(GP), \text{dec}(GP, s)) \}
\end{align*}
\]

Fig. 10 Reference Listing Variant of the DRC-Machine
Definition 9. Let \( k \) be a set of queues of a DRC-machine configuration. Let \( s_i, s_j \) be two site of \( S \). The set of messages under control of \( s_i \) via \( s_j \), written \( \text{control}(GP, s_i, s_j) \), is defined as:

\[
\text{control}(GP, s_i, s_j) = \{ m \mid m = \text{copy}(GP), m \in k(s_i, s_j), \\
m = \text{dec}(GP, s_j), m \in k(s_k, s_i) \text{ or} \\
m = \text{inc_dec}(GP, s_i, s_j), m \in k(s_k, \text{owner}(GP)) \\
\text{for any } s_k \}.
\]

The number of messages under control of \( s_i \) via \( s_j \) is precisely the value of the send-table of \( s_i \), for messages sent to \( s_j \).

Lemma 11. Let \( \langle \text{send}_T, \text{rec}_T, k \rangle \) be a legal DRC configuration. The following property holds. For any global pointer \( GP \in \hat{G} \), for any sites \( s_i, s_j \in S \):

\[ \text{send}_T(s_i, GP, s_j) = \#(\text{control}(GP, s_i, s_j)). \]

Proof. The equality is initially true and is preserved by each transition. The case analysis is available in invariant2.v. \( \square \)

Other properties such as safety and liveness still hold for this algorithm. The algorithm presented here combines reference counters and reference listing. By using reference listing, Birrel et al. [4] and Plainfosse and Shapiro [34] made messages idempotent and therefore resistant to message failure.

8 Related Work

8.1 Comparison with Other Related Mechanical Proofs

Jackson [17] has verified the correctness of a garbage collection algorithm using the PVS theorem prover. The algorithm that was studied is a stop-and-collect, non-copying collector. It uses Dijkstra, Lamport, Martin, Scholten, and Steffens' [7] tricolour marking scheme, but no concurrency (or distribution) was allowed in the algorithm. The algorithm was formalised as a labelled transition system. An embedding of linear temporal logic in PVS was used for reasoning. Safety and liveness properties, similar to ours, were derived for his algorithm.

Goguen, Brooksbysy and Burstall [11] present an abstract formulation of memory management based on a graph-theoretic representation of memory and related operations. They also formalised Dijkstra and Lamport's three-colour tracing algorithm and its correctness. Their proof has been encoded in Coq.
Russinoff [36] used the Boyer-Moore theorem prover to verify the safety and liveness property of Ben-Ari’s [2] mark-and-sweep garbage collection algorithm. Ben-Ari’s algorithm is a two colour solution to Dijkstra et al’s initial problem. He proves that a state predicate remains invariant, i.e. true for all reachable states. Havelund and Shankar [14] use refinement techniques to prove the safety of Ben-Ari’s algorithm, in PVS.

Gonthier and Doligez [8,13] proved the safety of a concurrent garbage collector used in Caml-light. The proof was carried out with the Larch Prover.

8.2 Reference Counting Algorithms for Garbage Collection

Reference-counting garbage collection was initially developed for uniprocessor systems [5]. Its principle is as follows: every time a pointer is copied or deleted, a reference counter is respectively incremented or decremented. It might seem that this algorithm can be extended straightforwardly to distribution by using two types of messages. A decrement message is sent to GP’s owner when GP is discarded; an increment message is sent to GP’s owner when GP is duplicated. However, this naïve extension fails to behave properly because non-causal [20] message delivery may reset the counter even though remote references may still be active.

Numerous solutions to this problem have been proposed. The most famous are weighted reference counting [3,41,9] and its optimised version [6], generational reference counting [12], or Piquer’s Indirect Reference Counting [32], which we have already discussed in Section 6.2. However, Lermen and Maurer’s [23,40] and Birfel’s [4] solutions are the closest to our work; we present them in the next two paragraphs.

In Lermen and Maurer’s algorithm [23,40], when a GP is duplicated, a create message is sent to its owner. The owner then sends an acknowledgement to the GP’s receiver. When a GP is discarded a decrement message is sent only after the acknowledgement has been received for this pointer. Lermen and Maurer’s technique also involves three sites (emitter, receiver, and owner), but it differs from ours: (i) The owner is involved every time the emitter duplicates a GP to the receiver in Lermen and Maurer’s algorithm, whereas it is involved only if the GP is not accessible on the receiver in our algorithm. (ii) Lermen and Maurer’s schema requires the receiver to maintain a count of both the number of copies made and the number
of acknowledgements received. Decrement messages can only be sent when both are equal.

Birrel et al. [4] present network objects, a distributed object-based language with a garbage collector. The owner of an object maintains a “dirty” set, which contains identifiers for all the processes that have GPs to the object. When a client first receives a GP, it makes a dirty call to the owner. When the GP is no longer reachable, as determined by the client’s local gc, the client makes a clean call and deletes GP. With the dirty calls, Birrel et al. reinstate the equivalent of an increment message. In order to avoid conflicts between dirty and clean calls, an acknowledgement message from the receiver of a GP to its emitter guarantees the impossibility of freeing the pointer on the emitter; the actual implementation prevents the method call from terminating on the emitter till the acknowledged message is received.

In Birrel’s algorithm, distributed reference counting activity is synchronous with the application. In particular, unmarshalling may be suspended by dirty calls. Furthermore, the emitter of a GP is only allowed to free its reference after the method invocation has terminated on the receiver; this may potentially create a zombie pointer for the duration of the computation. Our algorithm requires less synchronisations with the application; it is more flexible since, fully lazy, it behaves as indirect reference counting, and fully eager it behaves more like Birrel’s; the only difference is that our acknowledgement is sent by the owner in the form of a decrement message and not by the recipient of the reference.

The distributed variant of the Train GC [15] is also able to collect cycles; it combines a reference-counting style pointer-tracking mechanism with a substitution protocol. The latter algorithm bears some resemblance with Birrel’s but minimizes the number of exchanged messages: as a consequence, the owner of a GP may not be able to find (directly or indirectly) all the sites that have a copy of the GP.

Our algorithm has a major benefit as it is able to reorganise diffusion trees: when GC messages are all processed, the diffusion tree is completely flattened, and every site owning a GP directly “depends” from its owner. In the presence of mobile computations jumping from site to site, this allows sites to reclaim the space that was occupied by a mobile program, hereby avoiding zombie references as in indirect reference counting [32]. To the best of our knowledge, Shapiro, Gruber and Plainfossé [39], and subsequently Shapiro, Dickman, and Plainfossé [37, 38, 34] were the first to address the issue of short-cutting chains of pointers. They introduce the notion of SSP chains. A chain starts its existence by a single SSP (Scion/Stub pair); it increases
when sending the reference of a local object, or when migrating an object to some other site. In addition, they propose a technique to short-cut SSP-chains, hereby avoiding the equivalent of zombie references. They regard migration as a primitive notion to be supported by the GC; in this paper, we do not deal with migration, however, we have showed that support for mobility could be added as an extra layer, like a library, on top of the our garbage collection algorithm [27]. In Shapiro’s algorithm, chains of pointers are collapsed in a safe fashion by side-effect on remote invocations; specifically, piggybacking new location information onto invocation results, and location exception raising are used to this end. Garbage collection takes care of cleaning obsolete indirect chains.

Weighted reference counting (WRC) [3,41,9] associates a weight with each object and pointer. It maintains the invariant that the weight of an object is equal to the sum of weights of pointers pointing to it. When a pointer is copied, its weight is (equally) divided between the two copies. When the weight of a pointer reaches one, several solutions are possible. (i) An indirection cell may be introduced, but it behaves as a “zombie pointer” as in Piquer’s IRC. (ii) A message may be sent to the owner in order to request for more weight. Such a message may be regarded as a form of inc\_dec message, and we could see Weighted Reference Counting as a systematic method to decide when inc\_dec messages must be sent. Whenever a pointer is deleted, the object weight must be updated, which involves sending a “decrement” message to the owner.

Mancini and Shrivastava [25] investigate a fault-tolerant version of distributed reference counting. They also consider a triangular protocol, between the owner, the sender and the receiver of a reference, which differs from ours. It is an open question whether their fault-tolerant extension are applicable to our algorithm.

In [28], we investigate the scalability of reference listing in the presence of massively distributed computations. The size of send-tables is proportional to the number of sites participating to the computation. In order to reduce the burden on individual sites, we present a hierarchical organisation of the reference listing algorithm by which we are able to give a bound to the size of send-tables.

JAVA Remote Method Invocation comes with a distributed garbage collector [18]. It extends Birrel’s reference listing technique with a new approach to fault tolerance, where remote pointers are leased for a period of time. Sites having pointer copies must regularly renew their lease. Our approach can be extended without problem to reference
listing so that send-tables contain the sites to which GPs were sent, and a similar lease technique could also be adopted.

Reference counting garbage collection is only able to reclaim acyclic data structures. Several authors have combined distributed reference counting with other algorithms to provide cyclic garbage collection; for instance, Le Fessant, Piumarta, and Shapiro [22], Rodrigues and Jones [35], Lins and Jones [24], Lang, Quinmec and Piquer’s [21], or Hudson et al’s Distributed Train GC [15].

9 Conclusion

We have presented an algorithm for distributed reference counting and its proof of correctness, which involves both safety and liveness. We used the Coq proof assistant to formalise this proof. This work was a major undertaking, but gave valuable insights to the proof, which had been overlooked in the first place, during the paper proof.

A number of related issues are worth considering now. Support for mobile objects in conjunction with distributed reference counting would provide an excellent specification that could be used to certify mobile agents. Extending the reference listing algorithm with timestamps would make the algorithm resilient to faults. Finally, proving the hierarchical variant of the algorithm would be a useful exercise in order to build correct massively distributed computing environments.

Acknowledgements. The coding of the proof in Coq was initiated during Luc Moreau’s stay at École Normale Supérieure, Lyon; Luc Moreau would like to thank Luc Bougé for making this visit possible, as well as École Normale Supérieure, Lyon and INRIA Rhônes-Alpes for their support. Thanks to Andy Gravell for his comments on the paper.

References

