Recovering resources in the π-calculus (draft) [⋆]

David Teller – David.Teller@ens-lyon.fr

LIP (UMR CNRS, ENS Lyon, INRIA, Univ. Claude Bernard Lyon 1)

Abstract. Although limits of resources such as memory or disk usage are one of the key problems of many communicating applications, most process algebras fail to take this aspect of mobile and concurrent systems into account. In order to study this problem, we introduce the Controlled π-calculus, an extension of the π-calculus with a notion of recovery of unused resources with an explicit (parametrized) garbage-collection and dead-process elimination. We discuss the definition of garbage-collection and dead-process elimination for concurrent, communicating applications, and provide a type-based technique for statically proving resource bounds. Selected examples are presented and show the potential of the Controlled π-calculus.

1 Introduction

Virtually every piece of software or hardware in use nowadays relies on some form of communication. Whether a communication takes place between a program and the underlying operating system, between an application and a user, between the video board and the central processing unit or between several distant computers, it involves the concurrent emission and reception of information along a of *communication medium.* As no actual device has infinite resources, neither in memory limited-systems such as cellphones nor in enterprise-level webservers, only a finite number of communications may be performed simultaneously without failing, sometimes critically. Indeed, the problems of resource usage and resource awareness are crucial as this kind of failure may arise as a consequence of erroneous user-interaction, internal accidents or Denial of Service-like attacks.

A number of process algebras hold emissions and receptions as a primitive construction. Unfortunately, these calculi fail to address the problem of resource boundedness. This aspect of resource usage may be seen as closely related to the creation/restriction of fresh names, a construction shared between the class of process calculi known as *nominal process calculi* [3]. In this class of calculi, *names* are used to access resources while interaction is programmed by letting processes exercise some capabilities they have on names. Moreover, in nominal calculi, *names themselves may be seen as resources.*

In this paper, we study the crucial problem of resource boundedness using the π-calculus as a representative of nominal process calculi and as a base for

[⋆] This work is partially supported by IST Global Computing Profundis.

our work. To achieve our goal, we revisit the operator ν (the Greek letter 'nu', pronounced 'new') of name creation/restriction in nominal process calculi from two distinct points of view. In our setting, ν is both:

- the action of creating a new name whose identity is given by an agreed-upon identifier – hence using some of the available resources;
- an indication of scope, limiting the definition domain of a name *in space as well as in time* – hence also limiting the usage of resources.

Based on this notion of resources, in order to permit the design of resource-aware protocols, we enrich the π-calculus with the ability to wait for the recovery of now-unused resources by introducing a new capability $\daleth$ (the Hebrew letter 'daleth', to be pronounced 'delete'). In this Controlled π-calculus, for instance,

$$(\nu x)((\daleth x).P|Q) \ \longrightarrow^{*}\ P|Q \text{ when } x \text{ is not free in } P \text{ or } Q.$$

In other words, when x is not used anymore, its *finalizer* P may be triggered and the resources occupied by x may be recovered. This operator, somewhat dual to ν, is very close to the `Gc.finalise` function in OCaml or to finalizer methods in Java or C#. Although much safer than `free` or `delete` in traditional programming languages, the definition of finalization requires care to avoid problems such as resurrection of previously garbage-collectable resources – problems which may be easily witnessed in the aforementioned languages, and which is considered bad programming as it may lead to unpredictable behaviors, especially when in distributed or cross-language settings (see e.g. [1]).

In turn, resource recovery requires some form of garbage-collection mechanism, which may be more or less automated, to recover resources which cannot be accessed anymore although they may still appear syntactically in a term. For example, let us consider a process Q defined by

$$Q \ \stackrel{\triangle}{=}\ (\nu r)r(x).P \tag{1}$$

Q creates a new (secret) channel name r then immediately waits for an input on r. Since r is secret, no other process can possibly communicate using this channel, hence Q shall wait forever. Consequently, the resource r is unnecessary and could be recovered, as well as some of the resources which may appear in P, since P will never be actually executed.

In (1), a garbage-collector could rewrite Q into $\mathbf{0}$, possibly in several steps, hence releasing r. However, different garbage-collectors might take different approaches to remove Q and may or may not be able to analyze complex situations such as deadlocks or even livelocks. In order to deal with any possible garbage-collection method, we first define a relation $\mathcal{GC}$ between garbage-collectable processes and their garbage-collected counterparts using barbed simulations. We then introduce in the operational semantics of Controlled π a rule

$$\text{R-GC}\ \frac{P \longrightarrow_{GC} Q}{P \longrightarrow Q}$$

where $\longrightarrow_{GC}$ is a parametric relation on processes and included in $\mathcal{GC}$.

Although this enrichment of the π-calculus is not meant to add expressivity[1], it permits both dynamic reactivity to resources allocation and deallocation and easy reasoning the usage of resources. The possibility for processes to synchronize on resource recovery may be used to write resource-aware protocols which may be run with statically provable bounded amounts of resources. In order to prove these bounds, we introduce a resource-aware type system for the Controlled π-calculus, more powerful than our previous type systems of [10], for it does not require to replace replication by recursion.

In section 2, we introduce the Controlled π-calculus, starting with a simple language with only a simple form of garbage-collection (Core language) then expanding the definition to add garbage-collection mechanisms (Full language). We then present in section 3 a type system for resource-bounds guarantees. We conclude this paper by an overview of related works and future developments. Due to space limitations, proofs are not included in this document. They may be found in a companion technical annex [11].

2 The Controlled π-calculus

As the full definition of Controlled π-calculus (Cπ for short) requires a complex definition of dead-process elimination which in turn relies on the definition of the other parts of the calculus, we present Cπ in two steps. We start by defining the core of Cπ.

2.1 Core Cπ

Syntax The syntax of Core Cπ is presented on Figure 1. Although the syntax itself includes type annotations denoted by N in $(\nu x : N)$, we delay presenting a possible type system until Section 3. It is almost identical to that of the π-calculus, with the addition of $\urcorner$ and the special name $\diamond$ which denotes a special channel on which communications never actually occur and which cannot be bound. Process $(\urcorner x).P$, a finalizer for x, waits for name x to become unused then proceeds as P. Based on this syntax, we define the set of free names, fn and the set of bound names, bn, as in the π-calculus, with the addition of $(\urcorner x)$ – $(\urcorner x)$ is not a binder, rather x is a free name in $(\urcorner x).P$. Substitution, written $P\{a \leftarrow b\}$, is standard with the addition of rules for $\urcorner$: $((\urcorner x).P)\{a \leftarrow b\} = (\urcorner b).(P\{a \leftarrow b\})$ if $x = a$, $((\urcorner x).P)\{a \leftarrow b\} = (\urcorner x).(P\{a \leftarrow b\})$ otherwise.

In order to simplify the definition of the language, we shall only take into account terms in which no name is bound twice (i.e. there is no P such that $P \overset{\triangle}{=} a(x)Q$ or $P \overset{\triangle}{=} (\nu x : T)Q$ with x bound in Q). It is a standard result that, through appropriate renamings, we may keep the names unique, in any term, hence preventing the presence of names bound twice.

[1] Actually, we believe $\urcorner$ is encodable in π-calculus, although the encoding itself is way too complicated to permit the relatively simple proofs of Section 3.

Processes			Prefixes	
$P, Q ::= \mathbf{0}$	Terminated process		$\alpha ::= x(y)$	Input
$\mid\ P \mid Q$	Parallel composition		$\mid\ \overline{x}\langle y\rangle$	Output
$\mid\ \alpha.P$	Action			
$\mid\ !\alpha.P$	Guarded replication			
$\mid\ P + Q$	Non-deterministic choice			
$\mid\ (\nu x : N)P$	Resource creation/restriction			
$\mid\ (\urcorner x).P$	Resource finalization			

Fig. 1. Syntax of Controlled π-calculus.

S-PAR-ASSOC	$P\|(Q\|R) \equiv (P\|Q)\|R$	S-PAR-COMM	$P\|Q \equiv Q\|P$
S-PAR-NIL	$P\|\mathbf{0} \equiv P$	S-BANG	$!P \equiv P\|!P$
S-SUM-COMM	$P + Q \equiv Q + P$	S-SUM-NIL	$P + \mathbf{0} \equiv P$
S-SUM-ASSOC	$P + (Q + R) \equiv (P + Q) + R$		
S-NEW-REN	$(\nu x : N)P \equiv (\nu y : N).P\{x \leftarrow y\}$		if $y \notin fn(P)$
S-RCV-REN	$c(x).P \equiv c(y).P\{x \leftarrow y\}$		if $y \notin fn(P)$
S-NEW-COMM	$(\nu x : N)(\nu y : U)P \equiv (\nu y : U)(\nu x : N)P$		if $x \neq y$
S-NEW-PAR	$(\nu x : N)(P\|Q) \equiv P\|(\nu x : N)Q$		if $x \notin fn(P)$
S-FIN-PAR	$(\urcorner x).P\|(\urcorner x).Q \equiv (\urcorner x).(P\|Q)$		
S-STRUCT-PAR	$P \equiv Q \Longrightarrow P\|R \equiv Q\|R$		

Fig. 2. Structural congruence in Cπ

Structural congruence Structural congruence is the smallest equivalence verifying the relations of Figure 2. The rules are mostly identical to their counterparts from the π-calculus. Actually, rule S-NEW-COMM is slightly more precise than its counterpart in the standard π-calculus, as it preserves type information[2].

Also note that the only rule specific to $\urcorner$ is S-FIN-PAR, which states that triggering two processes by either one or two identical finalizations does not change the behavior of the system. The only other difference is the absence of the usual rule $(\nu x)\mathbf{0} \equiv \mathbf{0}$. In C$\pi$, this congruence is not true since $(\nu x : T)\mathbf{0}$ is a process holding resource x, although it does not use it, while $\mathbf{0}$ is a process holding no resource.

Similarly, we do not have $(\nu x : T)\alpha.P \equiv \alpha.(\nu x : T)P$, a rule found in several variants of the π-calculus, which states that the time of allocation of a resource is not important. In fact, this rule would not make sense in Cπ, for the time of allocation of a resource is important, as may be seen in the following process:

$$Q \triangleq c(x).(\nu y : T)P$$

Q acquires resource y only if reception $c(x)$ actually takes place. If no other process ever emits anything on channel c, the allocation will not take place.

Reduction rules Reduction rules of *Base Cπ* are defined in Figure 3.

R-COMM and R-SUM differ from their counterparts in the traditional π-calculus only insofar as communication cannot take place on the special channel

[2] This is actually not needed as long as we avoid names bound twice.

Communication	
R-COMM-SUM	$(\overline{a}\langle b\rangle.P + Q)\vert(a(x).R + S) \longrightarrow P\vert R\{x \leftarrow b\}\ (a \neq \diamond)$

Structure			
R-PAR	$\dfrac{P \longrightarrow Q}{P\vert R \longrightarrow Q\vert R}$	R-NEW	$\dfrac{P \longrightarrow Q}{(\nu n : T)P \longrightarrow (\nu n : T)Q}$
R-EQUIV	$\dfrac{P \equiv P'\ P' \longrightarrow Q'\ Q' \equiv Q}{P \longrightarrow Q}$		

Resource recovery	
R-AUTOCLEAN	$(\nu x : N)\mathbf{0} \longrightarrow \mathbf{0}$
R-FINALIZE	$(\nu x : N)(\urcorner x).P \longrightarrow P\{x \leftarrow \diamond\}$

Fig. 3. Reduction rules for Base Cπ

named $\diamond$. This may be seen as processes trying to dereference a `null` pointer: the process is immediately stopped by the operating system. In other words, the communication never occurs. R-PAR, R-NEW and R-EQUIV are standard.

R-AUTOCLEAN takes the place of $(\nu x : T)\mathbf{0} \equiv \mathbf{0}$ in the traditional π-calculus. It means that a terminated process may release the resources it is holding. R-FINALIZE states that a name which appears only as a finalizer may actually be finalized. This is comparable to the behavior of garbage-collectors in traditional languages: if the only reference to an object is a finalizer, then the object has become unreachable and should be finalized then garbage-collected. Finalization of name x ensures that x shall never be used again by substituting special name $\diamond$ (once again similar to a `null` pointer) to possible occurrences of x. Without this substitution, x could appear after $(\urcorner x)$, which would mean that x still lives after having been garbage-collected. Note that we could have specified either that $(\urcorner x).P$ is incorrect when x is free in P or that $(\nu x)(\urcorner x).P$ may only be reduced whenever x is free in P. We preferred our formulation with $\diamond$ as we may wish, at a later stage, to use $\urcorner$ and $\diamond$ to reason about some notions of secrecy (we briefly discuss this perspective in the conclusion) as well as about erroneous deallocations.

2.2 Example: a Bounded Resources Manager

Under most operating systems, in order for an agent (such as a Un*x-like process) to allocate a resource, it must request it through a system call such as `malloc`, `fork` or `fopen`. In turn, the operating system is responsible for limiting the actual amount of resources used by each agent.

One possible model for such a bounded resources manager is presented on Figure 4. Each $\Box$ offers a slot: at any time, the number of resources available for clients is equal to the number of concurrent $\Box$ in the resource manager. Conversely, each $\triangledown.P$ consumes a slot then proceeds as P. In order to allocate a resource, a client agent must request it on channel *alloc*. Whenever the resource manager receives a request, it waits for a slot to become available, consumes it by exerting $\triangledown$, creates the appropriate resource by exerting $(\nu c : N_c)$ and sends it to the client with $\overline{r}\langle c\rangle$. Whenever a resource c becomes unused and is garbage-collected, a new $\Box$ is created, hence keeping track of released resources.

$$\begin{array}{ll} \Box & \triangleq \bar{l}\langle\diamond\rangle \qquad \triangledown.P \triangleq l(x).P \text{ with } x \notin fv(P) \\ BRM_n & \triangleq (\nu l : N_l)\ (\ !alloc(r).\triangledown.(\nu c : N_c)(\overline{r}\langle c\rangle|(\urcorner c).\Box)\ \ |\ \ \underbrace{\Box|\Box|\ldots|\Box}_{n \text{ resources}}\ ) \end{array}$$

Fig. 4. Bounded resources manager

This bounded resources manager models the behaviour of operating systems' memory allocation, process creation, file opening, network access opening... Also note that several bounded resources managers may coexist, each handling a different kind of resources. We shall return to this example to prove several of its properties in the following sections.

Do note that this service only works whenever c is garbage-collected, which may not happen if a client is faulty. Let us consider

$$ROGUE \triangleq (\nu r)\overline{alloc}\langle r\rangle.r(c).c(x).\mathbf{0}$$

A purely syntactic examination of $ROGUE$ leads to believe that c is used as a channel to receive some value and should not be recovered. However, in $ROGUE|BRM_n$, no process will ever emit anything on channel c.

Other circumstances, such as deadlocks or communication attempts on channel $\diamond$, may hide the fact that a channel c is, in fact, unused. In each case, c falsely appears to be used because the name c appears syntactically in a process which will actually never be executed. Discovering and removing these occurences so as to recover unneeded resources is the task of the dead-processes eliminator.

2.3 Full Cπ

As in programming languages, for most systems, the actual set of dead processes not only changes during execution but also depends on non-deterministic choices made during the reduction of processes. For example, let us consider

$$(\nu a)(\nu c)(\ \overline{c}\langle\diamond\rangle.\mathbf{0}\ |\ c(x).P\ |\ c(x).Q\ )$$

where P uses a, Q does not and neither P nor Q ever use c. Since c is only used once, either P or Q will be triggered while the other process will end up dead. If P is triggered, a will be used, possibly infinitely often. If Q is triggered, a will become unused.

Therefore, we cannot consider dead process elimination (DPE for short) a static task. Instead, we must define it as a way to rewrite some processes whenever they are not needed anymore. Our intuition is that a process P may be replaced by $\mathbf{0}$ whenever P is triggered by some name a and the behavior of the whole scope of a is unaffected by the removal of P. We now define formally "P is triggered by some name a" (formally, "process P is guarded by a") and "the behavior [...] is unaffected" (formally, this is a barbed simulation).

$$\text{GC-N}{\scriptstyle\text{ULL}}\ \frac{Q_{[\diamond]}}{(Q,\mathbf{0})\in\mathcal{GC}} \qquad \text{GC-R}{\scriptstyle\text{EMOVE}}\ \frac{(\nu a:N)(P|Q)\ll(\nu a:N)P \qquad Q_{[a]}}{(\ (\nu a:N)(P|Q),\ (\nu a:N)P\ )\in\mathcal{GC}}$$

$$\text{GC-S}{\scriptstyle\text{UM}}\ \frac{(\ (\nu a:N)(P|Q),\ (\nu a:N)P\ )\in\mathcal{GC}}{(\ (\nu a:N)(P|(Q+R)),\ (\nu a:N)(P|R)\ )\in\mathcal{GC}}$$

$$\text{GC-R}{\scriptstyle\text{EPL}}\ \frac{(\ (\nu a:N)(P|Q),\ (\nu a:N)P\ )\in\mathcal{GC}}{(\ (\nu a:N)(P|!Q),\ (\nu a:N)P\ )\in\mathcal{GC}}$$

Fig. 5. Definition of $\mathcal{GC}$

Definition 1 (Guard). *We write $P_{[a]}$ ("P is guarded by a") whenever*

- *$P = a(x).Q$, $P=\overline{a}\langle x\rangle.Q$, $P=(\urcorner a).Q$,*
- *$P = !P'$ where $P'_{[a]}$ or*
- *$P = (\nu b:N)P'$ where $P'_{[a]}$ and $a\neq b$.*

Definition 2 (Barb). *We write $P\downarrow_{\overline{a}}$ ("P has a barb $\overline{a}$") whenever $P\equiv(\nu b_1: N_1)\dots(\nu b_n:N_n)(\overline{a}\langle b\rangle.Q+R|S)$ and $P\downarrow_a$ ("P has a barb a") whenever $P\equiv(\nu b_1:N_1)\dots(\nu b_n:N_n)(a(x).Q+R|S)$ (with $a\notin\{b_1,\dots,b_n\}$ in both cases).*

Definition 3 (Barbed Simulation). *Let $\mathcal{R}$ be a relation on processes. If, for any processes P and Q such that $P\mathcal{R}Q$ and for any guard ζ we have $P\downarrow_\zeta\Rightarrow Q\downarrow_\zeta$ and $P\longrightarrow P'\Rightarrow Q\longrightarrow Q'$ with $P'\mathcal{R}Q'$, then $\mathcal{R}$ is a barbed simulation. If $P\mathcal{R}Q$ for some barbed simulation we say that Q simulates P, written $P\ll Q$.*

Figure 5 contains our definition for the Elimination of Dead Processes. Recall that we examine the scope of a name and only remove processes guarded by that name – with the exception of $\diamond$. Rule GC-Null states that any process guarded by $\diamond$ can be safely removed, while rule GC-Remove formalizes our intuition: within the scope of name a, if Q is guarded by a and the process obtained by removing Q from the scope can simulate the original process, then Q is dead and may be removed. Rules GC-Sum and GC-Repl extend removing respectively to sums and replications. Note that, since replicated processes are always guarded, $!(\overline{a}\langle x\rangle.P+a(y).Q)$ is not a valid process and therefore causes no garbage-collection problems.

Definition 4 (Elimination of dead processes). *Let $\mathcal{GC}$ be the smallest relation defined by the rules of Figure 5. If $(A,B)\in\mathcal{GC}$, we say that A is* garbage-collectable *into B by Dead Process Elimination.*

Note that, in defining $\mathcal{GC}$, we chose to focus on a relation which only requires the examination of a given term. Although some more powerful relations exist, this restriction actually corresponds to the fact that examining a whole network is not an option for a garbage-collector. Also note that we chose not to encompass all mechanisms for rewriting processes by removing bits without affecting the outcome. While it would have been relatively easy to define a larger relation $\mathcal{GC}$, we consider that, say, removing prefixes or allocations or removing processes before they get a chance to become active, rather than being garbage-collectors, are other forms of compile-time or run-time optimizations.

A simple example Let us consider $A \triangleq (\nu a : N)!a(x).B$. Since B may hinder resource recovery, we may wish to garbage-collect A into $\mathbf{0}$. From GC-Remove, we see that a term such as A is garbage-collectable into $A' \triangleq (\nu a)\mathbf{0}$ by dead process elimination. Since $A' \longrightarrow \mathbf{0}$ by R-Autoclean, this term is, in fact, "as good as $\mathbf{0}$". Hence, replacing A by A' corresponds to what we wish to do.

Garbage-collecting a deadlock Similarly, let us define $A \triangleq a(x).\overline{b}\langle x\rangle.P$ and $B \triangleq b(x).\overline{a}\langle x\rangle.Q$ and let us consider $C \triangleq (\nu a : N_a)(\nu b : N_b)(A|B)$. Since C is deadlocked, we may also wish to garbage-collect it into $\mathbf{0}$.

Let us then define a relation $\mathcal{R}$ by $(\nu b : N_b)(A|B)\mathcal{R}(\nu b : N_b)A$. Since the only ζ such that $(\nu b : N_b)(A|B) \downarrow_\zeta$ is a, since we also have $(\nu b : N_b)A \downarrow_a$ and since there is no process D such that $(\nu b : N_b)(A|B) \longrightarrow D$, $\mathcal{R}$ is a barbed simulation. Hence, by GC-Remove, we have $((\nu b : N_b)(A|B), (\nu b : N_b)A) \in \mathcal{GC}$. Similarly, we have $((\nu a : N_a)A, (\nu a : N_a)\mathbf{0}) \in \mathcal{GC}$. Hence, once again, using $\mathcal{GC}$ to provide a garbage-collected version of C corresponds to what we wish to do.

As we have seen, $\mathcal{GC}$ gives us a notion of garbage-collectable terms. However, as this relation is undecidable, no compiler or runtime support could track $\mathcal{GC}$-related terms. Therefore, we now present a mechanism to include a number of simple rules whose role is to approximate $\mathcal{GC}$.

Definition 5 (Garbage-collector). *A garbage-collector $\longrightarrow_{GC}$ is a relation between processes such that $P \longrightarrow_{GC} Q \Longrightarrow (P, Q) \in \mathcal{GC}$.*

Rather than specifying a garbage-collection algorithm, we leave the actual garbage-collection mechanism as a parameter of the language. Let us write $\longrightarrow_{GC}$ for this parameter, which must be a garbage-collector. To obtain full Cπ, with respect to core Cπ, the only addition is the following rule:

$$\text{R-GC}\ \frac{P \longrightarrow_{GC} Q}{P \longrightarrow Q}$$

Do note that, informally, compositional type systems with a subject reduction property for core Cπ in which $\mathbf{0}$ is always typable, with any type, also have a subject reduction property in full Cπ.

2.4 Examples

In these examples, we write $P \hookleftarrow x$ whenever all occurences of name x in P have the form $(\urcorner x).R$. Let us consider the smallest relation $\longrightarrow_{GC1}$ defined by the set of rules of Figure 6. Relation $\longrightarrow_{GC1}$ is a simple garbage-collector which could be easily and efficiently implemented and which, although limited, is powerful enough to collect some simple examples.

$$\diamond(x).P \longrightarrow_{GC1} \mathbf{0} \quad (1) \qquad (\nu a : N)(P|a(x).Q) \longrightarrow_{GC1} (\nu a : N)P \text{ if } P \hookleftarrow a \quad (3)$$

$$\overline{\diamond}\langle x\rangle.P \longrightarrow_{GC1} \mathbf{0} \quad (2) \qquad (\nu a : N)(P|\overline{a}\langle x\rangle.Q) \longrightarrow_{GC1} (\nu a : N)P \text{ if } P \hookleftarrow a \quad (4)$$

$$\frac{(\nu a : N)(P|Q) \longrightarrow_{GC1} (\nu a : N)P}{(\nu a : N)(P|!Q) \longrightarrow_{GC1} (\nu a : N)P} \quad (5)$$

Fig. 6. A garbage-collector.

GC1 is a garbage-collector Due to space constraints, we shall only give an outline of the proof. We prove by induction that for all A and B, $A \longrightarrow_{GC1} B$ implies $(A, B) \in \mathcal{GC}$. Let us start by proving this property for rule (3).

Let us consider $\mathcal{R}$, the smallest relation such that, for all P such that $P \hookleftarrow a$, if $(\nu a : N)(P|a(x)Q) \longrightarrow^* R$ and $R \equiv (\nu a_1 : N_1, \dots, a : N, \dots, a_n : N_n)(P'|a(x)Q)$ then $(R, (\nu a_1 : N_1, \dots, a : N, \dots, a_n : N_n)P') \in \mathcal{R}$. One shows that $\mathcal{R}$ is a barbed simulation. Consequently, for a, b, x P and Q such that rule (3) holds, we have $((\nu a : N)(P|a(x).Q), (\nu a : N)P) \in \mathcal{R}$, hence $(\nu a : N)(P|a(x).Q) \ll (\nu a : N)P$. By GC-REMOVE, $((\nu a : N)(P|a(x).Q), (\nu a : N)P)$ is in $\mathcal{GC}$. This proves that only garbage-collectable terms are garbage-collected.

Proof for rule (4) is almost identical. The other proofs are trivial.

Putting GC1 to work In the following example, we shall use an instanciation of Cπ using rule GC1 as our garbage-collection parameter.

Let us consider the bounded memory manager BRM_n and its bad client $ROGUE$ as defined in Section 2.2. $BRM_n|ROGUE$ may be reduced in several steps into $(\nu c)(BRM'_n|c(x))$, where BRM'_n is the bounded memory manager with one resource locked and awaiting release, namely c. By definition of BRM_n, we have $BRM'_n \hookleftarrow c$. Hence, by rule (3), $(\nu c)(BRM'_n|c(x)) \longrightarrow_{GC1} (\nu c)BRM'_n$. By definition of BRM_n, we then have $(\nu c)BRM'_n \longrightarrow^* BRM_n$. In other words, although the bad client froze while holding resource c, GC1 was powerful enough to eventually recover c.

3 Type-based proofs for resource-bounds

3.1 The type system

Now that we have extended the π-calculus so as to make it aware of resources being released, we may define a type system which permits us to prove that resources are properly used. Namely, we intend to prove that systems which properly balance allocation and deallocation of resources have finite resource usage. Remember that, in Cπ, $(\nu c : N)$ stands for the allocation of some of the available resources while $(\urcorner c)$ permits to wait for the recovery of these resources. We will thus write $(\nu c : K, e)$ where K is some type information on the created name and e is the number of allocated resources. This lets us define formally the resource usage of a process:

Definition 6 (Resource usage). *We write $Res(P)$ for the number of resources process P currently uses, as defined by*

$$\begin{array}{llll}\text{Processes } T ::= t & t \in \mathbb{N} & \text{Communications } K ::= [N, z] & z \in \mathbb{Z} \\ \text{Names} \quad N ::= K, e & e \in \mathbb{N} & \qquad\qquad\qquad\qquad\quad \mid \ \mathit{Ssh} & \end{array}$$

Fig. 7. Grammar for the resource types

– *if* $P = (\nu c : K_c, e_c)Q$ *then* $Res(P) = e_c + Res(Q)$
– *if* $P = Q + R$ *then* $Res(P) = max(Res(Q), Res(R))$
– *if* $P = Q|R$ *then* $Res(P) = Res(Q) + Res(R)$
– *in all other cases,* $Res(P) = 0$

Do note that the resource usage of $c(x).(\nu d)P$ is zero as this process has not allocated any memory yet.

Full grammar for the type system is given on Figure 7. It includes entries for the type of processes and names. *Typing environments*, ranged over with Γ, are lists of associations of the form $c : N$, where c is a name and N a type. We write $\Gamma(c) = N$ to represent the fact that name c is associated to type N while $\Gamma, c' : N'$ stands for environment Γ extended with $c' : N'$. By convention, we have $\Gamma(\diamond) = N$ for any N. For any name c, $\Gamma(c) = K, e$ expresses the fact that the nature of c is K and that c occupies e resources until it is deallocated. K may be either $[V, z]$ if c is a channel which may be used to communicate names of type V or Ssh if c is not a channel. In the first case, z is used to balance the accounting of the effects of processes triggered by the communication.

Deallocation strategies, ranged over with Λ, are lists of names. The typing judgement for a process is of the form $\Gamma; \Lambda \vdash P : t$ and expresses the fact that P may use up to t resources under assumptions Γ and provided the deallocation in P of names appearing in Λ is taken into account. Keeping track of deallocations prevents erroneously counting that two finalizers registered for the same name deallocate the resources twice.

The typing rules are given on Figure 8. While T-Nil specifies that **0** may always be typed and may take into account the deallocation of any number of names, T-Repl specifies that $!P$ may be typed if and only if P does not require any resource and does take into account any deallocation – in other words, if the effect of P is accounted for somewhere else, presumably by taking advantage of communication cost balancing. T-New may only be applied whenever the deallocation of the allocated name has somehow already been taken into account and it specifies the fact that $(\nu x : K, e).P$ requires e more resources than P. T-Sum is applicable only when both processes may be typed under the same environments and strategies, and the number of resources occupied corresponds to the worst case scenario. T-Par, on the other hand, sums the contributions of both processes to the amount of resources allocated, as well as the list of names whose deallocation has been taken into account – the deallocation of one name may be taken into account only once. T-Finalize1 and T-Finalize2 permit the typing of finalizations. While the first of these rules does take into account the deallocation of a name, both by decreasing the number of necessary resources and by specifying into Λ that the deallocation has been taken into account, the second rule does neither, so as to allow the typing of several finalizers for the same name

$$\text{T-NIL } \Gamma; \Lambda \vdash \mathbf{0} : t \qquad \text{T-NEW } \frac{\Gamma, x : (K, e); \Lambda, x \vdash P : t_P}{\Gamma; \Lambda \vdash (\nu x : K, e)P : t_P + e}\ x \notin \Lambda$$

$$\text{T-REPL } \frac{\Gamma; \emptyset \vdash P : 0}{\Gamma; \Lambda \vdash !P : t} \qquad \text{T-SUM } \frac{\Gamma; \Lambda \vdash P : t \qquad \Gamma; \Lambda \vdash Q : t}{\Gamma; \Lambda \vdash P + Q : t}$$

$$\text{T-PAR } \frac{\Gamma; \Lambda_P \vdash P : t_P \qquad \Gamma; \Lambda_Q \vdash Q : t_Q}{\Gamma; \Lambda_P \cup \Lambda_Q \vdash P|Q : t_P + t_Q}\ \Lambda_P \cap \Lambda_Q = \emptyset$$

$$\text{T-FINALIZE1 } \frac{\Gamma; \Lambda \vdash P : t_P \qquad \Gamma(x) = \_, e}{\Gamma; \Lambda, x \vdash (\urcorner x).P : t_P - e}\ t_P \geq e, x \notin \Lambda$$

$$\text{T-FINALIZE2 } \frac{\Gamma; \Lambda \vdash P : t_P \qquad \Gamma(x) = \_, \_}{\Gamma; \Lambda \vdash (\urcorner x).P : t_P}\ x \notin \Lambda$$

$$\text{T-READ } \frac{\Gamma(c) = [N, z], \_ \qquad \Gamma, x : N; \Lambda \vdash P : t_P}{\Gamma; \Lambda \vdash c(x).P : t_P + z}\ x \notin \Lambda, t_P + z \geq 0$$

$$\text{T-WRITE } \frac{\Gamma(c) = [N, z], \_ \qquad \Gamma; \Lambda \vdash Q : t_Q \qquad \Gamma(y) = N}{\Gamma; \Lambda \vdash \overline{c}\langle y \rangle.Q : t_Q - z}\ t_Q - z \geq 0$$

Fig. 8. Typing rules for resource-bounds checking

or of a finalizer for a name which will only be known at runtime. For instance, T-FINALIZE2 will be used to type $a(x).(\urcorner x).P$ as the actual name which will be deallocated depends on the context. Similarly, when typing $(\urcorner a).P \mid (\urcorner a).Q$, using T-FINALIZE1 on either $(\urcorner a).P$ or $(\urcorner a).Q$ will force a into Λ, hence forcing us to use T-FINALIZE2 to type the rest of the process – hence counting only once the resources set free during $(\urcorner a)$. Rules T-READ and T-WRITE permit the typing of communications. Note first that T-READ may only be applied when the deallocation of x has not been taken into account as x is dynamically bound and we do not know whether its deallocation may already have been accounted for under another name. Let us now consider a communication between $A = c(x).P$ and $B = \overline{c}\langle y \rangle.Q$. As either A or B may be enclosed within a replication, we may use z to make sure $t_P + t_Q$ appears in the typing of A (if $z = -t_P$), in that of B (if $z = t_Q$), or balanced between these processes. Note that T-SUM, T-FINALIZE1, T-READ and T-WRITE rely on a resource expansion property (Theorem 1) to ensure that effects remain positive.

Properties We then have the following properties:

Theorem 1 (Resource expansion). *If we have $\Gamma; \emptyset \vdash P : t$, then for any $u \geq t$, we also have $\Gamma; \emptyset \vdash P : u$.*

Theorem 2 (Resource control). *If we have $\Gamma; \emptyset \vdash P : t$ and $P \longrightarrow^* Q$ then we also have $Res(Q) \leq t$.*

In other words, this type system permits us to find bounds on resources used by processes. This results are comparable to Resource control theorems in [10, 12] or to the Absence of over/under-flow theorem in [2].

3.2 Examples

Balancing costs Let us consider once again processes $A \triangleq c(x).P$ and $B \triangleq \overline{c}\langle y\rangle.Q$. Let us suppose we have $\Gamma(c) = [N_c, z_c], \_$, $\Gamma(y) = N_c$, $\Gamma, x : N_c; \emptyset \vdash P : t_P$ and $\Gamma; \emptyset \vdash Q : t_Q$.

Depending on the exact process we wish to type, we may need different values of z_c. For instance, if we wish to type $A \mid B$, with $z_c = 0$, we simply have $\Gamma; \emptyset \vdash A \mid B : t_P + t_Q$. Let us now consider rather process $A \mid !B$. Although B is enclosed under a replication, Q will probably not be triggered an infinite number of times. Actually, Q will be triggered at most as many times as $\overline{c}\langle y\rangle.Q$ may be actually reduced, in other words as many times as there are receptions on channel c. Therefore, in this case, instead of accounting for the cost of Q in B, it is preferable to consider that each reception on channel c costs an additional t_Q. To do this, it is sufficient to have $z = t_Q$. We then have $\Gamma; \emptyset \vdash A : t_P + t_Q$ and $\Gamma; \emptyset \vdash B : 0$. Hence, $\Gamma; \emptyset \vdash A \mid !B : t_P + t_Q$.

Note that z does not have actually to be exactly equal to t_Q: the operation is also possible with $z \geq t_Q$, albeit at the cost of a greater approximation, as we have $\Gamma; \emptyset \vdash A \mid !B : t_P + z$. Of course, by symmetry, we could have handled $!A \mid B$ in the same fashion.

Bounded Memory Manager Recall that the expression of the bounded resources manager BRM_n uses names l (the placeholder for resources, with type N_l), c (the actual resource being created, with type N_c) and $alloc$ (the channel used to communicate requests, with type $\Gamma(alloc)$). By convention, we will assume that both c and l occupy 1 resource. The actual value of N_l, N_c and $\Gamma(alloc)$, however, depends on the property we wish to prove.

For example, a desirable property is that BRM_n must be bounded in its resource usage while the action of requesting a name through $alloc$ does not induce any cost for a client. To prove this, we may use the types as in the *First typing* of Figure 9. With such types, we have $\Gamma; \emptyset \vdash BRM_n : n+1$, which proves that BRM_n is bounded indeed. The fact that requesting a name through $alloc$ is "free" is specified in the type of $alloc$ by the fact that $z = 0$.

We may also wish to prove that BRM_n actually occupies at most 1 resource while requesting a name through $alloc$ costs 1 resource to the client. This property corresponds to the *Second typing* of Figure 9. With such types, we have $\Gamma; \emptyset \vdash BRM_n : 1$, which proves that BRM_n occupies at most 1 resource. The fact that requesting a name through $alloc$ costs 1 resource is specified in the type of $alloc$ by the fact that $z = -1$.

	Resource usage is bounded	Each request costs 1
N_l	$[\_, \mathbf{1}], 1$	$[\_, \mathbf{0}], 1$
N_c	$\_, 1$	$\_, 1$
$\Gamma(alloc)$	$[[N_c, 0], \_, \mathbf{0}], \_$	$[[N_c, 0], \_, \mathbf{-1}], \_$

Fig. 9. Typing BRM_n.

4 Conclusion

We have introduced the Controlled π-calculus in order to permit the design and modelling of resource-aware protocols. Beyond adding to the π-calculus the ability to wait for the recovery of resources, we have presented a definition of garbage-collection and dynamic dead processes elimination in parallel systems. We have enhanced this calculus with a type system created to allow simple proofs on resource-bounds – a type system which may handle traditional replication, not just recursion as in our earlier works [10, 12].

Note that, although our type grammar only permits integer costs, it should be quite easy to rework the type system so as to permit variables. We could then obtain results such as "this process requires $3 \cdot e_a + 7 \cdot e_b - 2 \cdot e_c$ resources", where e_a, e_b and e_c are the resources required by the allocation of a, b and c. This would permit us to refine the information on the costs of protocols and systems.

We are also currently trying to expand Cπ to handle explicit distribution, as this kind of extension seems natural. In order to obtain semantics of finalization close to that of Java, OCaml or C$\sharp$, we are investigating the use of type systems with causality [4, 13] so as to prevent the appearance of name $\diamond$. We also wish to try and apply our results to existing implementations of the π-calculus.

Another aspect we are planning to study is the possibility of using finalization as an element of specification for secrecy: as $(\urcorner x).P$ guarantees that P will not be triggered as long as name x is still present somewhere in the system, this kind of property may be used to model protocols which must guarantee that they do forget informations (e.g. unless the user specifically asks that her password must be remembered, her webbrowser must forget it).

Related works Many works consider names as resources in the π-calculus and offer different mechanisms for protecting resources from being exposed or misused, without trying to account for allocation or deallocation [8, 9]. Other works [6] use linear types to prove bounds on the number of communications on channels in the π-calculus without considering allocation, deallocation or garbage-collection.

Garbage-collection for functional languages has been investigated in [7], although without finalization or guarantees on resource-bounds. On the other hand, a primitive similar to $\urcorner$ has been proposed along with a type system to design memory-bounded functional programs [5], with no extension to concurrency. This $@_\_$ primitive, however, is related to manual deallocation rather than automatic garbage-collection. Process algebras have also been designed by us and others [2, 10, 12] with explicit allocation and deallocation of cells (ambients, agents, ...) and type systems to offer resource-bounds guarantees. However, these calculi are specifically designed for this purpose, the main resource entity is a cell rather than a name and there is no notion of dead processes or finalization. Note that the main idea of BoCa [2] is actually quite close to our bounded resources manager with the addition of distribution. It is, however, built in the language and seems limited to only one kind of resources. Also note BoCa uses a different, dynamically typed, mechanism of guarded replication for preventing uncontrolled spawning of processes.

Our work is also related to some attempts at designing and modelling garbage-collectors for distributed calculi. Among these, a work on groups in π-calculus [14] offers a definition of dead process elimination close to ours. However, to the best of our knowledge, no such work provides either synchronization and reuse of resources or resource-bounds guarantees.

Acknowledgements We would like to thank Daniel Hirschkoff and Tom Hirschowitz for their time and their insightful suggestions during this work.

References

1. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1998.
2. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A calculus of bounded capacities. In *ASIAN'03*, number 2896 in LNCS, pages 205–223. Springer-Verlag, 2003.
3. A. Gordon. Notes on nominal calculi for security and mobility. In R. Focardi and R. Gorrieri, editors, *FOSAD*, volume 2171 of *LNCS*, pages 262–330. Springer-Verlag, 2002.
4. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *POPL*, pages 128–141, 2001.
5. S. Jost. `lfd_infer`: an implementation of a static inference on heap space usage. In *Proceedings of SPACE 2004*, 2004.
6. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
7. Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of FPCA 1995*, pages 66–77. ACM Press, 1995.
8. N.Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *CONCUR'99*, volume 1664 of *LNCS*, pages 557–572. Springer-Verlag, 1999.
9. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *8th IEEE Logics in Computer Science*, pages 376–385, Montreal, Canada, 1993.
10. D. Teller. Formalisms for mobile resource control. In *FGC'03*, volume 85 of *ENCS*. Elsevier, 2003.
11. D. Teller. Resource recovery in pi-calculus – technical annex. 2004. available at `http://perso.ens-lyon.fr/david.teller/recherche/Publications/cpita.pdf`.
12. D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*. Springer-Verlag, 2002.
13. N. Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoretical Computer Science*, pages 371–386, 1996.

14. S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. In *MFCS 2000: 25th ISMFCS*, 2000.