Checking Global Usage of Resources Handled with Local Policies *

Chiara Bodei, Viet Dung Dinh and Gian-Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy
{chiara,dinh,giangi}@di.unipi.it

We present a methodology to reason about resource usage (acquisition, release, revision,...) and, in particular, to predict *bad* usage of resources. Keeping in mind the interplay between local and global information that occur in application-resource interactions, we model resources as entities with local policies and we study global properties that govern overall interactions. Formally, our model is an extension of π-calculus with primitives to manage resources. To predict possible bad usage of resources, we develop a Control Flow Analysis that computes a static over-approximation of process behaviour.

1 Introduction

We live in a world where mobility and distribution are part of the standard everyday (digital) devices, resources seem unlimited and always available. Scalable and elastic do not mean exactly this. Sooner or later people experiment that the available resources are not sufficient or that their existence can arise between their service requests and their satisfaction. One can add a 4 GB folder on her/his private data repository (e.g. Dropbox) and realises that the synchronisation with the Dropbox servers takes too long, especially for big amounts of data. The suggested solution is therefore to upload less big folders at a time, because the upload time is lower than the download one. Another one, let us say in Europe, wants to download some free software and suffers from the very slow downloading until realising that the chosen web-site is in USA and at that time most of the people have just woken up and began to surf on the web.

When designing a web-based distributed application the first focus is on the way of rendering the required functionalities into suitable tasks. This phase often abstracts away from the other side of the coin, i.e. the operational way of formalising the tasks and therefore of managing the assigned computational resources. Resource awareness, instead, should be part of the game from the very beginning, without being simply delegated to low-level supports. Standard programming metaphors consider resources as entities geographically distributed (typically available over the Internet) and with their own states, costs and access mechanisms. Furthermore, resources are not created nor destroyed by applications, but directly acquired on the fly when needed from suitable resource rental services, without investing in new infrastructures. Clearly, resource acquisition is subject to availability and requires the agreement between client requirements and service guarantees (Service Level Agreement – SLA).

The dynamic acquisition of resources increases the complexity of software since software capabilities strictly depends on resource availability. *Ubiquitous computing* [1] and *Cloud computing* [26, 73, 4] provide illustrative examples of applications where resources awareness is an essential concern. A further step towards ubiquitous computing is when software pervades the objects of our everyday life, e.g. webTV, cars, smartphones, ebook readers, etc. Often, these heterogeneous entities have a limited

*Research supported by the European FET Project "ASCENS", by the Italian MIUR project "Security Horizons" and by project *PRA*_2016_64 "Through the fog", funded by the University of Pisa.

Submitted to:
Science of Computing Programming

computational power, but are capable of connecting to the Internet, coordinating and interacting each other, in the so-called "plug&play" fashion. The real objects, as well as others of virtual nature (programs, services, etc.), which are connected in this way, form the Internet of Things (IoT) [5]. Similarly, cloud systems offer through the network a hardware and software infrastructure on which end-users can run their programs on-demand. In addition, a rich variety of dynamic resources, such as networks, servers, storage systems, applications and services are made available. The key point is that resources are "virtualised" so that they appear to their users as fully dedicated to them, and potentially unlimited.

In our programming model processes and resources are distinguished entities. Resources are computational entities with their own life cycle. They can range from computational infrastructures, storage systems and data services to special-purpose devices. Processes are instead thin entities that can dynamically acquire the required resources when available, but that cannot create any resource. This programming model abstracts some of the features of the systems discussed above. Indeed, our challenge consists in the metaphor of designing applications on top of heterogeneous sets of resources, by ensuring interoperability among them and providing transparency with respect to the actual resource implementation. As an example, let us consider a cloud system offering computing resources. The available resources are the CPU units of a given power and processes can only acquire the CPU time, when available, to run some specialised code. Similar considerations apply to storage services, where client processes can only acquire slots of the available storage. In our programming model, the deployed resources can be dynamically *reconfigured* to deal with resource upgrade, resource unavailability, security intrusion and failures. However, the reconfiguration steps that update the structure of the accessible resources are not under the control of client processes. Therefore, clients establish SLAs for having the necessary guarantees on the availability of the required resources.

This paper introduces the formal basis of our programming model. Specifically, we introduce the G-Local π-calculus, a process calculus with explicit primitives for the distributed ownership of resources, which can be either virtual or physical. In our calculus, resources are not statically granted to processes, but they are dynamically acquired on the fly, when required. We start from the π-calculus [62] and we extend it with primitives to represent resources and with the operations for acquiring and releasing resources on demand. Central to our approach is the identification of an abstract notion of resource. In our model, resources are *stateful* entities available in an active and dynamic environment, and processes continuously interact with them. A resource is described through the declaration of its interaction endpoint, its *local* state and its *global* properties. Global properties establish and enforce the SLA to be satisfied by any interaction the resource engages with its client processes. We do not address here the precise nature of these properties. The definition of the global interaction properties may involve several kinds of trade-offs. We assume that the global interaction properties can be expressed by means of a suitable resource-aware logic in the style of [11], or contract-based logic as in [30, 13]. The interplay between the local operations over the state and the global SLA enforcement that occur in the process-resource interactions motivates the adjective *G-Local* given to our extension of the π-calculus.

Since we start from π-calculus, name-passing is the basic communication mechanism among processes. Beyond exchanging channel names, processes can pass resource identifiers as well. Resource acquisition is instead based on a different abstraction. To acquire the ownership of a certain resource, a process issues a suitable request. Such request is routed in the network environment to the resource. The resource is granted only if it is available. Conceptually, the process-resource interaction paradigm adheres to the *publish-subscribe* model: resources act as publishers, while processes act as subscribers. Actually, the publish-subscribe paradigm not only is a natural choice to represent distributed resources, but it also emphasises the fact that resources have to be published by external parties and therefore have to be available to everyone through appropriate requests. Notice that processes issue their requests with-

out being aware of resource availability. When they have completed their task on the resource, acquired in an exclusive but limited usage, they release it and make it available for new requests. Furthermore, whenever the usage of the acquired resource does not respect the global SLA policy at hand, the release operation is forced. We argue that this approach relaxes the inter-dependencies among computational components thus achieving a high degree of loose coupling among processes and resources. Under this regard, our model resembles coordination models based on the notion of tuple space [40] and seems to be particularly suitable to manage distributed systems in which the set of published resources is subject to frequent changes and dynamic reconfigurations.

In summary, our approach combines the basic features of π-calculus with the distributed acquisition of stateful resources equipped with global SLA policies. This is our first contribution and also constitutes an original feature of the proposal since it covers both aspects of process-resource interactions.

A second contribution consists in the development of a *Control Flow Analysis* (CFA) for our calculus. The analysis computes a safe approximation of resource usage. Hence, it can be used to statically check whether or not the global properties of resource usage are respected by process interactions. In particular, it helps detecting possible *bad usage* of resources, due to policy violations. The analysis identifies the sensible points in the code that need dynamic checks in order to avoid policy violations. Our analysis also manages iterative behaviour of processes acting over resources. The analysis constructs a *finite-state* model that approximates the executable behaviour of processes capable of performing unbounded iterative actions over shared resources.

We adopt a top-down approach to present the G-Local π-calculus. We first informally discuss the programming abstractions at the basis of our approach. The remaining part of the paper is devoted to introduce the formal description of the calculus and the static analysis for predicting resource usage. Finally, we discuss related work, and we conclude by presenting plans for further research. The proofs can be found in Appendix.

This article is the full version of the extended abstract in [21]. The conference paper addressed the problem of managing finite sequential processes only. We include here extended definitions and explanations, and a simpler, yet more complete static analysis. In the paper we add several examples to better illustrate our proposal. Finally, we include proofs for our key results.

2 Programming Abstractions for Resource-awareness

This section intuitively introduces the programming abstractions of the G-Local π-calculus with the help of some simple, yet illustrative examples.

2.1 Resource Rental Service

In our first example, we consider a resource rental service making a number of identical computational resources available over the Internet. Several distributed applications have this shape, especially those that make resources accessible to a variety of external dynamic entities. Furthermore, a number of issues gives support and advice to the development of applications, equipped with many copies of the same kind of resources including high availability, load balancing and fault tolerance.

Here, we assume to provide a storage service as our resource. For simplicity, we suppose to have many copies of this kind of resource, i.e. to have a *resource pool* containing a certain number of storage resources. On a resource *storage*, clients can perform some access actions like opening, writing

and releasing ($open(storage)$, $write(storage)$, and $release(storage)$, respectively). The publication of a storage resource in G-Local π-calculus is as follows:

$$(storage, \varphi_{storage}, \eta)\{\mathbf{0}\}$$

The declaration introduces a stateful resource called *storage*, whose state information is stored in the *history* variable η that collects the sequence of all the resource access actions. Furthermore, $\varphi_{storage}$ describes the global policy (SLA) that any interaction with the storage has to satisfy. The declaration, also, introduces an explicit resource boundary $\{\}$ for representing the scope of availability of the resource *storage*. In the initial configuration, the encapsulated behaviour, specified by the nil process $\mathbf{0}$, is null. Finally, the resource pool is implemented as the parallel composition of the given storage resources.

Policies are modelled as regular safety properties of the traces that collect all the resource accesses, hereafter called *histories*. In this simple example, the adopted policy $\varphi_{storage}$ expresses that the correct way to operate over the storage resource is that the action *open* must occur *before* action *write*. Bad prefixes in traces, which correspond to policy violations, can be easily detected by suitable kinds of run time reference monitors such as security automata [63], subsequently extended to edit automata [53] and usage automata [8], or to the guardians introduced in [35].

A client process may acquire the temporary ownership of the published resource *storage*, by means of an explicit request, codified by the following primitive

$$req(storage)\{P\}$$

Only if the required resource is available, i.e. if $(storage, \varphi_{storage}, \eta)\{\mathbf{0}\}$ occurs in the common pool, a binding between the client and the resource can be established. More precisely, the process can enter the corresponding resource boundary, where the encapsulated process becomes the code P, in the resulting process $(storage, \varphi_{storage}, \eta)\{P\}$. At this point, the client process can access the resource *storage*, according to its code P, as long as its access actions on the resource do not violate the SLA policy at hand. Therefore, in our framework, resources are dynamically allocated to the client on demand, but their usage must adhere to explicit global policies.

Clients could also find the name of a resource through a discovery process, which relies on the existence of a process acting as *pool-front-end*. The pool-front-end accepts process discovery actions of clients and communicates the name of the copy of the resource of interest. The common pool can have a further dimension of dynamics since the membership of the copy of the resource to the pool can be dynamic as well. The pool-front-end taking care of the resource discovery can offer output actions in pure π-calculus style, such as

$$\bar{d}\langle storage\rangle$$

The pool-front-end sends the identifier of the resource (in this case the *storage* resource) over the channel d. The client process, connected with the pool-front-end, waits for the identifier of the resource on the channel d before trying to acquire the resource indicated by the received identifier. We show below the code describing the *main* of a possible client process, where "..." stands for further internal activities on data performed by the process, not of interest here.

$$d(s).req(s)\{open(s).write(s). \ldots .release(s)\}$$

The client process receives the actual resource identifier, e.g. *storage*, and binds it with the local variable s. Therefore, a request can initially include a resource variable s, provided that the request occurs after

an input, where the variable gets its binding value. After the binding, the process can require the resource $storage$, by becoming

$$req(storage)\{open(storage).write(storage). \ldots .release(storage)\}$$

Afterwards, if the required resource is available, it becomes

$$(storage, \varphi, \eta)\{open(storage).write(storage). \ldots .release(storage)\}$$

The state η is updated at each access action of the client process, by simply appending the action name. For instance, here, after the release it amounts to $\eta' = \eta.open.write.release$. After its usage, the resource becomes available again:

$$(storage, \varphi_{storage}, \eta')\{\mathbf{0}\}$$

but now its local state (η') stores the updated (log of) the activities performed during the process interactions. Since the action $open$ occurs before the action $write$, in the trace $open.write.release$, the client process respects the policy φ on the resource $storage$. Instead, the client process

$$d(s).req(s)\{\ldots .write(s).release(s)\}$$

would not obey to the policy, because no action $open$ occurs before the action $write$.

We can observe that the client process is organised in a loosely coupled fashion with respect to the resource. Loose coupling has the main advantage that when the resource is upgraded or modified, the impact on the client process is minimised.

This simple example illustrates a distinctive feature of our approach. Resources are entities available in the digital environment that surrounds processes. For example, we may consider a fridge in the so-called Internet of Things [5]. The fridge can be modelled as a suitable resource storage, ready to be queried for its contents to discover perishable goods. Notice that our model abstracts from heterogeneous communication facilities, i.e. we assume a programming interface for communicating with the available resources. We argue that this is essential for the integration of distributed resources into the digital environment.

2.2 Processes Coordination

We now consider a simple refinement of the setting presented above useful to show how the notion of stateful resource allows the definition of coordination policies among processes. To this purpose, we equip the storage resource with the *Chinese Wall* policy [24]. It is a classical security policy used in commercial and corporate business services. In such scenarios, all the objects related to the same corporation are usually grouped together into a so-called company data set, e.g. bankA, bankB, insuranceA, insuranceB, and so on. Furthermore, all the company data sets whose corporations are in competition are grouped together. Each group represents a *conflict of interest* class, e.g. the Bank conflict class includes bankA and bankB and Insurance contains insuranceA and insuranceB. Intuitively, the Chinese Wall policy imposes that access data cannot be performed on the *wrong* side of the wall. More precisely, a resource can be obtained only in two cases: either the resource is in the same company data set of the resources previously accessed, or the resource belongs to a different class of conflict of interest. To clarify the Chinese-Wall policy with an example, let us consider the following trace,

$$read(insuranceA).read(bankA).read(insuranceB)$$

P, P'	$::=$		*processes* $\in$ *Proc*	π, π'	$::=$	*action prefixes*
	$\mathbf{0}$		empty process		$\bar{x}y$	name output
	$\mid$	$\pi.P$	prefix action	$\mid$	$\bar{x}z$	resource output
	$\mid$	$(\nu x)\, P$	restriction	$\mid$	$x(y)$	name input
	$\mid$	$P + P'$	choice	$\mid$	$x(s)$	resource input
	$\mid$	$P \parallel P'$	parallel composition	$\mid$	$\alpha(z)$	access action
	$\mid$	$(r, \varphi, \eta)\{P\}$	resource joint point	$\mid$	$rel(z)$	release action
	$\mid$	$req(z)\{P\}$	resource request point			
	$\mid$	$!P$	replication			

Figure 1: The syntax of G-Local π-calculus.

It easy to see that the above trace violates the expressed policy, because reading an object in the data set insuranceB is not permitted after having accessed insuranceA, which is in the same conflict class Insurance.

In this paper, we will exploit static analysis techniques, specifically Control Flow Analysis, to verify properties like those described above to guarantee that resources are requested and released, without violating the stated policies.

3 The G-Local π-Calculus

In this section, we formally present the syntax of our calculus and we define its operational semantics. We further motivate the constructs of our calculus via a simple running example.

3.1 Syntax

G-Local π-calculus is an extension of the monadic version of π-calculus [62], with primitives to declare, access and dispose resources. The syntax is displayed in Figure 1, where we assume a countably infinite set $\mathscr{N}$ of channel names (ranged over by a, b, t, x, y, w), a set $\mathscr{R}$, composed by resource identifiers (ranged over by $r, r', ...$), and resource variables $\mathscr{S}$ (ranged over by $s, s', ...$). We assume that these sets are pairwise disjoint. For the sake of simplicity, we require that all the resource variables that initially occur in a process are distinct.

Furthermore, we assume to have a set $\mathscr{A}$ of actions (ranged over by α, β) for accessing resources, a distinguished action $rel \notin \mathscr{A}$ that denotes releasing of the acquired resource, and a set Φ of policies over $\mathscr{A}$, ranged over by φ, φ'. We use $z, z', ...$, which we simply call resources, to range over both resource identifiers and resource variables. We denote with $(\mathscr{A} \cup \{rel\})(r)$ the resource actions on the resource r. From now on, for the sake of simplicity, we often omit the trailing $\mathbf{0}$.

The output prefixed process $\bar{x}y.P$ sends the name y along channel x and then continues as P. Analogously, the resource output prefixed process $\bar{x}z.P$ sends the resource z along channel x and then continues as P. The input prefixed process $x(y).P$ ($x(s).P$, respectively) receives a channel name (a resource identifier, respectively) via the channel x, to which y (s, respectively) is bound, and then behaves like P. Note that resource names can be communicated, however they *cannot* be used as private names or as channels; therefore they can be objects but not subjects of communications. The operator $+$ describes non deterministic choice, while the operator $\parallel$ denotes parallel composition of processes. The operator (νx) acts as a static binder for the name x in the prefixed process that makes x different from all the external names.

Our extension introduces the notion of resource and some resource-aware constructs. Resources are stateful entities, equipped with policies that constrain their usage. More precisely, a resource is a triple (r, φ, η), where $r \in \mathscr{R}$ is a resource identifier, $\varphi \in \Phi$ is the associated policy and $\eta \in (\mathscr{A} \cup \{rel\})^*$ is the history that represents the resource state (ε denotes the empty state).

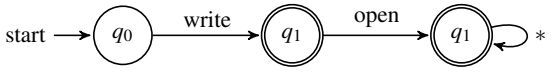
The access prefix $\alpha(z)$[1], as in [9], models the invocation of the operation $\alpha \in \mathscr{A}$ over the resource z. Access labels specify the kind of access operation, e.g. *rd* for read, *wr* for write, and so on. In our calculus, an action α that accesses a resource can be seen as an event observed by resource monitors and the operation represents a basic component of the resource usage. The special action $rel \notin \mathscr{A}$ denotes the release of the acquired resource. A finite sequence η of juxtaposed access and release events in $\mathscr{A} \cup \{rel\}$ is called *history* or *trace* (ε denotes the empty trace).

Policy Checking We discuss here our machinery for verifying whether a history is compliant with respect to a policy φ on the resource r. Policies $\varphi \in \Phi$, which specify the required properties of resource usage, are modelled as regular safety properties [43] of histories that express that nothing bad will occur during a computation. A policy φ can be represented by the set of its *bad* prefixes, which can be recognised by resorting to suitable finite state automata, such as the usage automata of [8, 11, 10]. Note that this approach follows the so-called *default-accept paradigm*, where only the unwanted behaviour is explicitly mentioned.

As a consequence, the language denoted by a policy φ is the set of *unwanted traces* and accepting states are offending: entering in an accepting state corresponds to a policy violation. Let $L(\varphi)$ denote the language of φ. We write $\eta \models \varphi$ if the history η does not lead to offending states in the corresponding usage automaton. We are now ready to define the notion of policy compliance of a history.

Definition 3.1 (Policy compliance) *Given a history η, η is* compliant *with the policy φ, in symbols, $\eta \vDash \varphi$ if and only if $\eta \notin L(\varphi)$.*

For instance, the following automaton (where we remove release prefixes, for improving readability) can abstractly represent the storage policy $\varphi_{storage}$, discussed in the previous section, that requires that an action $open$ must occur *before* an action $write$.



It is easy to see that $open.write \models \varphi_{storage}$, while $write \not\models \varphi_{storage}$ and, also, $write.open \not\models \varphi_{storage}$.

Policy checking can be mechanically implemented by resorting to standard automata-based model checking techniques [69]. We refer to [8, 12, 10] for a more detailed discussion. Note, in passing, that our notion of stateful resource behaves much like a *reference monitor* that acts as the manager of the resource. The reference monitor also observes the operations carried over the resource and stops execution if one of them violates the resource usage policy [34, 43].

Resource boundaries To cope with resource-awareness, we introduce two primitives that manage resource boundaries: resource joint point $(r, \varphi, \eta)\{P\}$ and resource request point $req(z)\{P\}$. Instead of considering resources as an extension of private names with a set of traces of access events like in [51], we introduce a specific construct (r, φ, η) to represent a notion of resources with explicit boundaries.

[1]This notation recalls the functional style and is not to be confounded with the input notation, where, e.g., in $a(s)$, s is still a bound name, as will be illustrated next, but where $a \in \mathscr{N}$ represents the receiving channel, while in $\alpha(z)$, $\alpha \in \mathscr{A}$ is the access action. Names are Latin letters like a, b, c, while access actions names range over Greek letters, apart from the action for releasing resources, which uses the special keyword 'rel'.

Unlike the dynamic scope of private names, where "private names" (i.e. resources) can be concurrently accessed by processes, our resources can be accessed, in mutual exclusion, only by the processes in the scope of the corresponding resource boundaries. Resource boundaries define indeed the portion of the code where resources are visible and accessible. More in detail, a process $(r,\varphi,\eta)\{P\}$ behaves like P, in the resource boundary (r,φ,η), where r can be accessed according to the policy φ. The state η is updated at each access action. Intuitively, the process P when plugged inside the resource boundary $(r,\varphi,\eta)\{P\}$ can fire actions that act over the resource r, provided that the policy that regulates its usage is satisfied. Processes of the form $(r,\varphi,\eta)\{\mathbf{0}\}$ represent *available resources*. These processes are idle: they cannot perform any operation. Therefore, resources can only react to requests. A process $req(z)\{P\}$ represents a process that asks the resource z. Note that z can be associated either to a resource identifier or to a resource variable. In the second case, the process resource request point comes after an input in which the variable can be instantiated, as in a process like $a(s)...req(s)\{P\}$. Only if the request is fulfilled, i.e. the required resource is available, the process P can enter the required resource boundary (r,φ,η), and use r, according to the policy φ.

The notions of free names $\mathsf{fn}()$, bound names $\mathsf{bn}()$, and substitution $\{-/-\}$ are defined as expected. The set of names $\mathsf{n}(P)$ of a process P is defined as the union of $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$. Similarly, we define the notion of free resource identifiers $\mathsf{fr}()$, bound resource identifiers $\mathsf{br}()$, and resource identifiers $\mathsf{r}()$, as described in the table below.

Case	$\mathsf{fn}()$	$\mathsf{bn}()$	$\mathsf{fr}()$	$\mathsf{br}()$
$\mathbf{0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\bar{x}y.P$	$\{x,y\}\cup\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$
$\bar{x}z.P$	$\{x\}\cup\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\{z\}\cup\mathsf{fr}(P)$	$\mathsf{br}(P)$
$x(y).P$	$\{x\}\cup(\mathsf{fn}(P)\setminus\{y\})$	$\{y\}\cup\mathsf{bn}(P)$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$
$x(s).P$	$\{x\}\cup\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\mathsf{fr}(P)\setminus\{s\}$	$\{s\}\cup\mathsf{br}(P)$
$\alpha(z).P$	$\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\{z\}\cup\mathsf{fr}(P)$	$\mathsf{br}(P)$
$rel(z).P$	$\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\{z\}\cup\mathsf{fr}(P)$	$\mathsf{br}(P)$
$(\nu x)P$	$\mathsf{fn}(P)\setminus\{x\}$	$\{x\}\cup\mathsf{bn}(P)$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$
$P\|P'$	$\mathsf{fn}(P)\cup\mathsf{fn}(P')$	$\mathsf{bn}(P)\cup\mathsf{bn}(P')$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$
$P+P'$	$\mathsf{fn}(P)\cup\mathsf{fn}(P')$	$\mathsf{bn}(P)\cup\mathsf{bn}(P')$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$
$(r,\phi,\eta)\{P\}$	$\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\{r\}\cup\mathsf{fr}(P)$	$\mathsf{br}(P)$
$req(z)\{P\}$	$\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\{z\}\cup\mathsf{fr}(P)$	$\mathsf{br}(P)$
$!P$	$\mathsf{fn}(P)$	$\mathsf{bn}(P)$	$\mathsf{fr}(P)$	$\mathsf{br}(P)$

Also resource substitution $\{r/s\}_R$, which replaces each occurrence of the resource variable s with the resource identifier r, is defined as expected. Note that the tag R distinguishes this substitution from the standard one for channel names. We describe below its application.

Case	*Application*
$\mathbf{0}\{r/s\}_R$	$\mathbf{0}$
$(\bar{x}y.P)\{r/s\}_R$	$\bar{x}y.P\{r/s\}_R$
$(\bar{x}s.P)\{r/s\}_R$	$\bar{x}r.P\{r/s\}_R$
$(\bar{x}z.P)\{r/s\}_R$	$\bar{x}z.P\{r/s\}_R$ if $z \neq s$
$(x(y).P)\{r/s\}_R$	$x(y).P\{r/s\}_R$
$(x(s').P)\{r/s\}_R$	$x(s').P\{r/s\}_R$
$(\alpha(s).P)\{r/s\}_R$	$\alpha(r).P\{r/s\}_R$
$(rel(s).P)\{r/s\}_R$	$rel(r).P\{r/s\}_R$
$((\nu x)P)\{r/s\}_R$	$(\nu x)P\{r/s\}_R$
$(P_1+P_2)\{r/s\}_R$	$P_1\{r/s\}_R+P_2\{r/s\}_R$
$(P_1 \parallel P_2)\{r/s\}_R$	$P_1\{r/s\}_R \parallel P_2\{r/s\}_R$
$((r',\varphi,\eta)\{P\})\{r/s\}_R$	$(r',\varphi,\eta)\{P\{r/s\}_R\}$
$(req(s)\{P\})\{r/s\}_R$	$req(r)\{P\{r/s\}_R\}$
$(!P)\{r/s\}_R$	$!(P\{r/s\}_R)$

A process is *well-formed* when all the constructs that involve a resource variable occur after an input, where the variable gets its binding value. As a consequence, at run time, the corresponding actions are closed with respect to resource identifiers. From now on, we only deal with well-formed processes.

Example 3.2 *We consider a small example that represents a green cloud computing environment, where computing the energy cost is crucial to save energy. In this scenario, we can associate an integer cost c_α to each access $\alpha(r)$ to a resource r and fix a threshold cost value v_r that cannot be passed for each resource r. We further suppose that $rel(r)$ has null cost. Intuitively, policies are respected when the sum of the costs of accesses do not pass the value fixed for each resource.*

Suppose to have a couple of resources r_1 and r_2, with v_{r_1} and v_{r_2} as threshold values. Users receive resources on channels x_i (for $i = 1,\ 2,\ 3$) and y.

The initial configuration is given below. Resources (r_1 and r_2) are initially available and have empty state. The access action α comes associated with the cost c_α, and the action β with the cost c_β. Suppose, for instance, that c_α is quite expensive and that the threshold of r_2 is not very high, while that of r_1 is. Suppose, in particular, that $3 \cdot c_\alpha < v_{r_1}$, while $3 \cdot c_\alpha + c_\beta > v_{r_1}$ and that $c_\alpha + c_\beta > v_{r_2}$. To reflect these constraints, the policy φ_1 can be represented by the set of traces that begin with the bad *prefix $\alpha.rel.\alpha.rel.\alpha.rel.\beta$, while φ_2 by the set of traces that begin with the* bad *prefixes $\alpha.rel.\beta$ and $\beta.rel.\alpha$, as shown in Figure 2, where we remove release prefixes, for improving readability.*

$$
\begin{array}{rl}
Res ::= & (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\} \\
Users ::= & x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\} \parallel x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\} \parallel \\
& x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel y(t).req(t)\{\beta(t).rel(t)\} \\
Plan ::= & \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0} \\
System ::= & Res \parallel Users \parallel Plan
\end{array}
$$

Note that, in particular, Users is a well-formed process, since all the resource requests and access actions that refer to a resource variable occur after an input, where the variable can obtain its binding identifier.

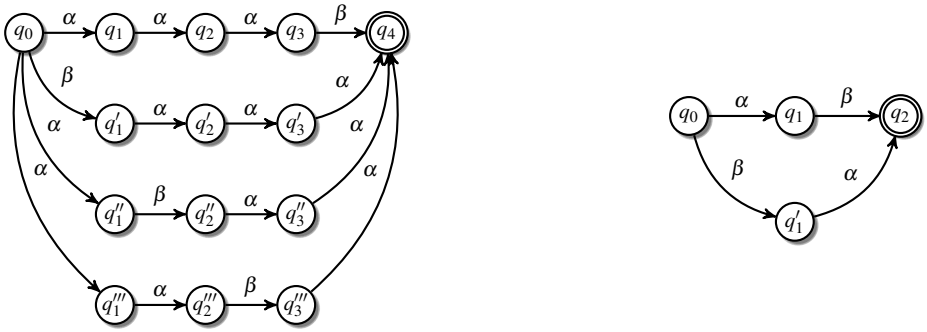


Figure 2: On the left: the policy ϕ_1 on r_1 expresses that a history composed by three actions α and one action β is forbidden. On the right: the policy ϕ_2 on r_2 expresses that a history composed by an action α followed by an action β, or composed by an action β followed by an action α is forbidden.

3.2 Operational semantics

The semantics of the calculus is given in terms of a transition system on well formed processes, defined up to structural congruence. The transition relation defined in Table 1, extends the standard semantics of π-calculus with suitable rules to deal with resource constructs.

We will use μ, μ' to indicate generic labels of transitions, τ for silent actions, $\bar{x}y$ for free name output, $x(y)$ for name input, $\bar{x}y$ for free output, $\bar{x}(y)$ for bound output, $\bar{x}r$ for free resource output, $a(s)$ for resource input, $\alpha(r)$, $\alpha?r$ and $\overline{\alpha(r)}$ ($rel(r)$ and $rel?r$, resp.) for closed, open, and faulty access (or release actions) over resource r, respectively, and whose precise meaning will be clear in a while. As in the standard π-calculus, the effect of bound output is to extrude the sent name from the initial scope to the external environment. Again, we extend the definitions of free and bound names for labels, to deal with resources identifiers, by introducing $\mathsf{fr}(\mu), \mathsf{br}(\mu)$, and $\mathsf{r}(\mu)$.

Label	μ	$\mathsf{fn}(\mu)$	$\mathsf{bn}(\mu)$	$\mathsf{fr}(\mu)$	$\mathsf{br}(\mu)$
Silent action	τ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Free name output	$\bar{x}y$	$\{x,y\}$	$\emptyset$	$\emptyset$	$\emptyset$
Free resource output	$\bar{x}r$	$\{x\}$	$\emptyset$	$\{r\}$	$\emptyset$
Name Input and Bound output	$x(y), \bar{x}(y)$	$\{x\}$	$\{y\}$	$\emptyset$	$\emptyset$
Resource Input	$x(s)$	$\{x\}$	$\emptyset$	$\emptyset$	$\{s\}$
Resource Access	$\{\alpha(r), \alpha?r, \overline{\alpha(r)}\}$	$\emptyset$	$\emptyset$	$\{r\}$	$\emptyset$
Resource Release	$\{rel(r), rel?r\}$	$\emptyset$	$\emptyset$	$\{r\}$	$\emptyset$

The relation of structural congruence on processes, denoted by $\equiv$, is defined as the least congruence satisfying the clauses in Figure 3. This includes the standard laws of the π-calculus, such as the monoidal laws for the parallel composition and the choice operator and the rules for restrictions and replication. To simplify the definition of our CFA in Section 4, we replace the standard notion of alpha-conversion with a notion of *disciplined alpha-conversion* that, without loss of generality, imposes a discipline in the choice of fresh names. We postpone the explanation of this notion to the next section.

In addition to the standard equational laws, we introduce specific laws for managing the resource-aware constructs. Resource request and resource joint points can be swapped with the restriction boundary since restriction is not applied to resource identifiers but only to channel names. The last law is crucial for managing the discharge of resources. This law allows rearrangements of available resources, e.g. an available resource is allowed to enter or escape within a resource boundary. The intuition is that available

- $P \equiv Q$ if P and Q are alpha-equivalent (in the *disciplined* sense explained in the text)
- $(Proc/_{\equiv}, +, \mathbf{0})$ and $(Proc/_{\equiv}, \|, \mathbf{0})$ are commutative monoids
- $(\nu x)\mathbf{0} \equiv \mathbf{0}$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$, $(\nu x)(P \parallel Q) \equiv P \parallel (\nu x)Q \;\; if \;\; x \notin \mathsf{fn}(P)$,
- $!P \equiv P \parallel !P$
- $(\nu x)(r,\varphi,\eta)\{P\} \equiv (r,\varphi,\eta)\{(\nu x)P\}$
- $(\nu x)req(r)\{P\} \equiv req(r)\{(\nu x)P\}$
- $(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel (r_1,\varphi_1,\eta_1)\{P\} \equiv (r_1,\varphi_1,\eta_1)\{(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel P\}$

Figure 3: Structural congruence.

resources can freely float in the digital environment. Note that if two processes P_1 and P_2 are equivalent, also the resource boundaries that include them are equivalent, i.e. $(r,\varphi,\eta)\{P_1\} \equiv (r,\varphi,\eta)\{P_2\}$ and $req(r)\{P_1\} \equiv req(r)\{P_2\}$.

Finally, we need the following auxiliary function *SubProc* that, given a resource variable s and a process P, returns the subprocess prefixed by an input on s. We recall that since the resource variables are all distinct, there is at most one input prefix with s as a variable in a given process. The function is inductively defined as follows.

$$\begin{array}{rcl}
SubProc(s,\mathbf{0}) & = & \mathbf{0} \\
SubProc(s,x(s).P') & = & P' \\
SubProc(s,\pi.P') & = & SubProc(s,P') \\
SubProc(s,(\nu x)P) & = & SubProc(s,P) \\
SubProc(s,P_1+P_2) & = & SubProc(s,P_1) \cup SubProc(s,P_2) \\
SubProc(s,P_1 \parallel P_2) & = & SubProc(s,P_1) \cup SubProc(s,P_2) \\
SubProc(s,(r,\varphi,\eta)\{P\}) & = & SubProc(s,P) \\
SubProc(s,req(r)\{P\}) & = & SubProc(s,P) \\
SubProc(s,!P) & = & SubProc(s,P)
\end{array}$$

We now comment on the rules of the operational semantics. Rules (Act), (Par), (Res), $(Comm)$, $(Cong)$, $(Choice)$, $(Open)$ and $(Close)$, in Table 1, are the standard π-calculus ones. The rules in *Act* describe actions of processes, e.g. the free input and the free output. Concretely, $\bar{x}y.P$ sends the name y along the channel x and then behaves like P, while $x(y).P$ receives a channel name via the channel x, to which y is bound, and then behaves like P. Rules in (Act_R) include the analogous output and input axioms for resource identifiers: $\bar{x}r.P$ sends the resource identifier r and then behaves like P, while $x(s).P$ receives a resource identifier via the channel x, to which s is bound, and then behaves like P. Note that our semantics adopts a *late* approach, e.g. variables are actually bound to values when communications occur.

Rule (Par) expresses the parallel computation of processes, while the rule *Choice* represents a choice among alternatives. Rule $(Comm)$ is used to communicate free channel names. Rules (Res) and $(Open)$ are rules for restriction. The first ensures that an action of P is also an action of $(\nu x)P$, provided that the restricted name x is not in the action. In the case of x in the action, the rule $(Open)$ transforms a free output action $\bar{a}x$ into a bound output action $\bar{a}(x)$, which basically expresses opening scope of a bound name. Rule $(Close)$ describes communication of bound channel names, which also closes the scope of a bound name in communication.

We are now ready to comment on the semantic rules that correspond to the treatment of resources. To help the intuition, we illustrate the more distinctive rules in Figures 4 and 5. Besides the communication

$$\text{(Act)} \quad \begin{cases} \bar{x}y.P \xrightarrow{\bar{x}y} P \\ x(y).P \xrightarrow{x(y)} P \end{cases} \qquad \text{(Cong)} \quad \frac{P_1 \equiv P_1' \;\; P_1' \xrightarrow{\mu} P_2' \;\; P_2' \equiv P_2}{P_1 \xrightarrow{\mu} P_2}$$

$$\text{(Par)} \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2} \; \mathsf{bn}(\mu) \cap \mathsf{fn}(P_2) = \emptyset \qquad \text{(Choice)} \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'}$$

$$\text{(Res)} \quad \frac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'} \; z \notin \mathsf{n}(\mu) \qquad \text{(Open)} \quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} \; y \neq x$$

$$\text{(Comm)} \quad \frac{P_1 \xrightarrow{\bar{x}y} P_1' \;\; P_2 \xrightarrow{x(w)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{y/w\}} \qquad \text{(Close)} \quad \frac{P_1 \xrightarrow{\bar{x}(y)} P_1' \;\; P_2 \xrightarrow{x(w)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} (\nu y)(P_1' \parallel P_2'\{y/w\})}$$

$$\text{(Act}_\text{R}\text{)} \quad \begin{cases} \bar{x}r.P \xrightarrow{\bar{x}r} P \\ x(s).P \xrightarrow{x(s)} P \\ \alpha(r).P \xrightarrow{\alpha?r} P \\ rel(r).P \xrightarrow{rel?r} P \end{cases} \qquad \text{(Comm}_\text{R}\text{)} \quad \frac{P_1 \xrightarrow{\bar{x}r} P_1' \;\; P_2 \xrightarrow{x(s)} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{r/s\}_R} \; r \notin \mathsf{fr}(SubProc(s, P_2))$$

$$\text{(Acquire)} \quad req(r)\{P\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} (r, \varphi, \eta)\{P\} \qquad \text{(Release)} \quad \frac{P \xrightarrow{rel?r} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{rel(r)} (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P'}$$

$$\text{(Policy}_1\text{)} \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\alpha(r)} (r, \varphi, \eta.\alpha)\{P'\}} \qquad \text{(Policy}_2\text{)} \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \not\models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\overline{\alpha(r)}} (r, \varphi, \eta.\overline{\alpha})\{\mathbf{0}\} \parallel P'}$$

$$\text{(Local)} \quad \frac{P \xrightarrow{\mu} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{\mu} (r, \varphi, \eta)\{P'\}} \; \mu \notin (\mathscr{A} \cup \{rel\})(r)$$

Table 1: Operational semantics.

rules, the rules in (Act_R) model a process that tries to perform an action α (or *rel*, resp.) on the resource r. This attempt is seen as an *open action*, denoted by the label $\alpha?r$ ($rel?r$, resp.). Intuitively, an access action on r is successful only if fired inside the boundary of r, and the action α satisfies the related policy (see $(Policy_1)$ and $(Policy_2)$ rules). Similarly, releasing a resource r succeeds only if performed inside the boundary of r (see $(Release)$). Rule $(Comm_R)$ explicitly models the communication of resource identifiers between processes. The rule is similar to rule $(Comm)$ (we use two rules to emphasise the different nature of the exchanged data). The only differences are the input premise, which is derived by the rules in (Act_R), for the input of resource identifiers, and the side condition. Since resource identifiers refer to concrete resources, renaming seems inadequate to avoid captures of identifiers. Nevertheless, processes could require a resource r, inside a boundary on r, or before a resource boundary on r. As a solution, we put a constraint on the use of resources, by preventing nested uses of resources with the same identifier. More precisely, we require that the input prefix $a(s)$, inside P_2, willing to receive the resource identifier r is (i) neither in the scope of a resource boundary devoted to r and still in use, (ii)

nor it prefixes a process that include resource boundaries on r. To perform this check, we verify whether r does not belong to the free resource identifiers of the subprocess of P_2 prefixed by the input $a(s)$, subprocess which is returned by the function *SubProc* called on parameters s and P_2.

A process P can acquire a resource r, when available, by entering the corresponding resource boundary (r, φ, η), as stated by rule $(Acquire)$ (see Figure 4). Symmetrically, according to rule $(Release)$, the process P can release an acquired resource r and update the state of its resources, by appending the label rel to η (see Figure 4). In the resulting process $(r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P$, P is outside the resource boundary, while the resource is available again.

Rules $(Policy_1)$ and $(Policy_2)$ (see Figure 5) check whether the execution of the action α on the resource r does not violate the policy φ, i.e. whether the updated state $\eta.\alpha$, obtained by appending α to the current state η is compliant with respect to φ, written as $\eta.\alpha \models \varphi$. If the policy is obeyed, then the updated trace $\eta.\alpha$ is stored in the resource state according to the rule $(Policy_1)$ and the action becomes *closed*. When this is not the case, i.e. when an invalid access α is tempted on r, the resource is forcedly released according to rule $(Policy_2)$, the corresponding *faulty* action $\overline{\alpha(r)}$ is fired, and the updated trace $\eta.\overline{\alpha}$ is recorded in the resource state. We assume here that $\mathscr{A}$ also includes faulty actions such as $\overline{\alpha}, \overline{\beta}$. Invalid accesses to resources are therefore identified by transitions of faulty actions. In the continuation P' of the process, all the actions over the resource r remain *open* and without any effect on the resource. Rule $(Policy_2)$ has been introduced to manage the recovery from bad access to resources, in case of policy violation. We recall that we can check the consistency of a trace, by resorting to suitable automata, whose languages represent the set of unwanted traces.

Finally, the side condition of rule $(Local)$ expresses that actions that do not access or release the resource r can bypass resource boundaries.

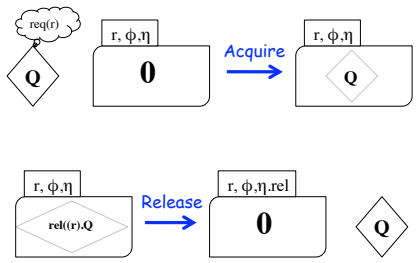


Figure 4: *Rules* $(Acquire)$ *and* $(Release)$

Remark 3.3 *Rule* $(Acquire)$ *is not inductively given in the SOS style. We could rephrase it in the SOS style by the following rules, where rule* $(ResReq)$ *guesses the available resources, in an early style:*

$$(\text{ResReq}) \quad req(r)\{P\} \xrightarrow{(r,\varphi,\eta)} (r,\varphi,\eta)\{P\} \qquad (\text{ResJoin}) \quad (r,\varphi,\eta)\{\boldsymbol{0}\} \xrightarrow{\overline{(r,\varphi,\eta)}} \boldsymbol{0}$$

$$(\text{Comm}'_{\text{R}}) \quad \frac{P_1 \xrightarrow{(r,\varphi,\eta)} P_1' \quad P_2 \xrightarrow{\overline{(r,\varphi,\eta)}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

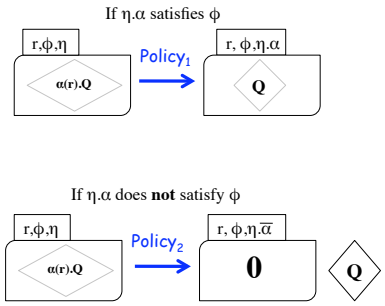


Figure 5: *Rules* $(Policy_1)$ *and* $(Policy_2)$

Example 3.4 *Back to our running example (Example 3.2), where all the processes used here have been defined, the following dynamic computation illustrates how the system works and a possible trace that violates the policy. We recall that each access* $\alpha(r_i)$ *to a resource* r_i *has an integer cost* c_α *and the policy* φ_i *of each resource is obeyed if the resource threshold value* v_{r_i} *is not passed.*

At the beginning, the process Users instantiates a new user (a resource request point) when receiving a resource identifier, e.g. r_1*:*

$$\begin{array}{l} System \\ \equiv Res \parallel Users' \parallel x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\} \parallel \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\boldsymbol{0} \\ \xrightarrow{\tau} Res \parallel Users' \parallel Plan' \parallel req(r_1)\{\alpha(r_1).rel(r_1)\}, \end{array}$$

where $Users' ::= x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\} \parallel x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel y(t).req(t)\{\beta(t).rel(t)\}$ *and* $Plan' ::= \bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\boldsymbol{0}$.
The new user can acquire the resource and other resource identifiers are also available (on the channels x_2, x_3, y*). In the following, for the sake of simplicity, we only show the sub-processes involved in the computation. Assume that the new user takes* r_1*. In this case, we have the following transitions:*

$$\begin{array}{l} req(r_1)\{\alpha(r_1).rel(r_1)\} \parallel (r_1,\varphi_1,\varepsilon)\{\boldsymbol{0}\} \\ \xrightarrow{\tau} (r_1,\varphi_1,\varepsilon)\{\alpha(r_1).rel(r_1)\} \\ \xrightarrow{\alpha(r_1)} (r_1,\varphi_1,\alpha)\{rel(r_1)\} \\ \xrightarrow{rel(r_1)} (r_1,\varphi_1,\alpha.rel)\{\boldsymbol{0}\} \end{array}$$

Now, the remaining three further users are similarly instantiated, triggered by the $Plan'$ *outputs* $\bar{y}\langle r_2\rangle$, $\bar{x_2}\langle r_2\rangle$, *and* $\bar{x_3}\langle r_1\rangle$.

$$\begin{array}{l} Users'|Plans' \\ \xrightarrow{\tau} x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\} \parallel x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel req(r_2)\{\beta(r_2).rel(r_2)\} \parallel \bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle \\ \xrightarrow{\tau} x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel req(r_2)\{\beta(r_2).rel(r_2)\} \parallel req(r_2)\{\alpha(r_2).rel(r_2)\} \parallel \bar{x_3}\langle r_1\rangle \\ \xrightarrow{\tau} req(r_2)\{\beta(r_2).rel(r_2)\} \parallel req(r_2)\{\alpha(r_2).rel(r_2)\} \parallel req(r_1)\{\alpha(r_1).rel(r_1)\} \end{array}$$

In the current setting, the new three users make one request on the remaining resource r_1 and two requests on r_2. Since we have only one copy of r_2, requests should be done one at a time. Suppose to first satisfy the requests of r_1, as in the following transitions:

$$\begin{array}{l}(r_1,\varphi_1,\alpha.rel)\{\alpha(r_1).rel(r_1)\}\\ \xrightarrow{\alpha(r_1)}(r_1,\varphi_1,\alpha.rel.\alpha)\{rel(r_1)\}\\ \xrightarrow{rel(r_1)}(r_1,\varphi_1,\alpha.rel.\alpha.rel)\{\boldsymbol{0}\}\end{array}$$

As a result, the first resource is available again. Similarly, the second request proceeds as follows:

$$\begin{array}{l}req(r_2)\{\beta(r_2).rel(r_2)\}\parallel(r_2,\varphi_2,\varepsilon)\{\boldsymbol{0}\}\\ \xrightarrow{\tau}(r_2,\varphi_2,\varepsilon)\{\beta(r_2).rel(r_2)\}\\ \xrightarrow{\beta(r_2)}(r_2,\varphi_2,\beta)\{rel(r_2)\}\\ \xrightarrow{rel(r_2)}(r_2,\varphi_2,\beta.rel)\{\boldsymbol{0}\}\end{array}$$

Note that, at this point, if the third request proceeded, then a forced release could occur. This happens because the user would attempt to perform an α action on r_2, on which a β action was previously performed: since the trace $\beta.rel.\alpha$ belong to the bad prefixes identified by the policy φ_2, this inconsistent trace would represent a usage that leads to a policy violation.

$$\begin{array}{l}req(r_2)\{\alpha(r_2).rel(r_2)\}\parallel(r_2,\varphi_2,\beta.rel)\{\boldsymbol{0}\}\\ \xrightarrow{\tau}(r_2,\varphi_2,\beta.rel)\{\alpha(r_2).rel(r_2)\}\\ \xrightarrow{\overline{\alpha(r_2)}}(r_2,\varphi_2,\beta.rel.\overline{\alpha})\{\boldsymbol{0}\}\parallel\ rel(r_2)\end{array}$$

4 Control Flow Analysis

In this section, we present a control flow analysis for our calculus, extending the one for π-calculus [19]. This CFA is simpler than the one in [21], because it is not contextual, i.e. the analysis is insensitive to the context. This facilitates its implementation. The CFA computes a safe over-approximation of all the possible communications of resource identifiers and names on channels. Furthermore, it provides an over-approximation of all the possible usage traces on the given resources, thus predicting possible cases of bad usage.

Performed under the perspective of processes, the analysis tries to answer to questions like the following: "Are the resources initially granted sufficient to guarantee a correct usage?". We assume that a certain fixed amount of resources is given and we do not consider any dynamic reconfiguration. The reconfiguration is up to the resource manager and is not addressed by our CFA.

For simplicity, we incrementally present our analysis. We first develop (in the next Sub-section 4.1) the analysis for the subset of G-Local π-calculus in which the processes enclosed in resource boundaries are *sequential processes*. The extension to general parallel processes, which requires some more complex technical machinery, will be shown later, in Sub-section 4.2. Finally, in Sub-section 4.3, we will address the static treatment of finite-state iterative processes.

As anticipated in the previous section, to simplify the definition of our Control Flow Analysis, as usually done (see e.g. [19, 18]), we *discipline* the choice of fresh names, and therefore alpha-conversion.

Indeed, the result of analysing a process P must still hold for all its derivative processes Q, including all the processes obtained from Q by alpha-renaming. In particular, in the CFA, the names and the variables that occur in P must keep a connection with the names possibly modified during the dynamic evolution. To statically maintain the identity of values and variables, we partition all the names used by a process into finitely many equivalence classes and we use the names of the equivalence classes instead of the actual names. Names in the same equivalence class are assigned to a common canonical name. Not to further overload our notation, we simply write n for the canonical name of the name n. As a consequence, there are only finitely many canonical names in any execution of a given process. The resulting *disciplined alpha-renaming* requires that two names can be alpha-renamed only when they have the same canonical name. To have only finitely many canonical names in any execution of a given process, we assign the same canonical name to all the names that can be generated by a restriction under a replication [39].

4.1 CFA: Step 1

We first provide a CFA for finite processes in which the bodies of resource boundaries are sequential processes, ranged over by Q, Q'. We let S, S' to range over both processes and sequential processes. Intuitively, a sequential process Q represents a single thread of execution in which one or more resources can be used. This implies that only one single point for releasing each resource occurs in each non deterministic branch of a process. We assume that, by construction, in every sequential branch located in the scope of the resource r, there always exists a release action $rel(r)$ coming after the last access to the resource. The assumed constraint amounts to guaranteeing that resources are released after their use. Note that the only parallel branching configuration we keep, $(r, \varphi, \tilde{\eta})\{\mathbf{0}\}^{\jmath}||Q$, is needed for handling release actions.

We further assume here, and in the extensions of the CFA, that the analysed processes are deadlock-free. The deadlock detection problem is, in general, very difficult in loosely coupled distributed systems, where information about the state of process and resources is not stored in a single central point. This makes it challenging to gather global properties like deadlock freedom. This is the case of our programming model. Deadlock freedom can be addressed either by dynamic or static techniques. *Gossip-based algorithms* (also known under the name of epidemic algorithms), see e.g. [54, 55], have emerged as an effective dynamic mechanism to gather global knowledge in a distributed peer-to-peer networks (see e.g. [32, 48]). In these algorithms each component of the network is in charge of forwarding state information to the other components of the network. Furthermore, there are static techniques and tools (based on type systems) that have been proposed for deadlock detection in mobile calculi (see [41, 60, 49, 72, 44], to cite only a few, and their discussion in Section 5). These techniques provide the basis for developing deadlock detection in our setting. Since our focus here is on resource usage, we assume to exploit either static or dynamic techniques to detect and prevent deadlocks, before performing our CFA.

Here, we only analyse processes that have finite behaviour, i.e. processes without replication. Finally, to facilitate our analysis, we consider a labelled version of the chosen subset of our calculus, described by the syntax in Figure 6. Labels $\jmath, \chi \in \mathscr{L}$ are associated with resource boundaries as follows: $(r, \varphi, \tilde{\eta})\{\mathbf{0}\}^{\jmath}$, $(r, \varphi, \tilde{\eta})\{Q\}^{\chi}$ and $req(r)\{Q\}^{\chi}$. Labels χ are supposed to be unique. As usual in static analysis, this is merely a convenient way of indicating "program points" (here the sub-processes in resource scopes), useful when developing our CFA. Note that these labels can be mechanically attached and do not affect the dynamic semantics. Traces are accordingly extended, by collecting pairs (α, χ) or (rel, χ), where resource actions are associated to the labels of the boundaries in which they are performed. Formally,

extended traces $\tilde{\eta}$ are included in $\widehat{\mathscr{A}^*}$, where $\widehat{\mathscr{A}} = (\mathscr{A} \cup \{rel\} \times \mathscr{L})$.

In Table 2, we only present the semantic rules in which labels play a role. Original semantics can be obtained by simply removing labels by actions and resource boundaries.

P, P'	::=	*processes* $\in$ *Proc*	
		$\mathbf{0}$	empty process
		$\pi.P$	prefix action
		$(\nu x)\ P$	restriction
		$P + P'$	choice
		$P \parallel P'$	parallel composition
		$(r, \varphi, \tilde{\eta})\{Q\}^{\chi}$	resource joint point
		$req(z)\{Q\}^{\chi}$	resource request point
Q, Q'	::=	*sequential processes* $\in$ *SeqProc*	
		$\mathbf{0}$	
		$\pi.Q$	
		$(r, \varphi, \tilde{\eta})\{Q\}^{\chi}$	
		$(r, \varphi, \tilde{\eta})\{\mathbf{0}\}^{\jmath} \| Q$	
		$req(z)\{Q\}^{\chi}$	

Figure 6: Labelled syntax.

$$\text{(Acquire)}\quad req(r)\{P\}^{\chi} \parallel (r, \varphi, \tilde{\eta})\{\mathbf{0}\}^{\jmath} \xrightarrow{\tau} (r, \varphi, \tilde{\eta})\{P\}^{\chi} \qquad \text{(Release)}\quad \frac{P \xrightarrow{rel?r} P'}{(r, \varphi, \tilde{\eta})\{P\}^{\chi} \xrightarrow{rel(r)} (r, \varphi, \tilde{\eta}.(rel, \chi))\{\mathbf{0}\}^{\jmath} \parallel P'}$$

$$\text{(Policy}_1\text{)}\quad \frac{P \xrightarrow{\alpha?r} P' \quad \tilde{\eta}.(\alpha, \chi) \models \varphi}{(r, \varphi, \tilde{\eta})\{P\}^{\chi} \xrightarrow{\alpha(r)} (r, \varphi, \tilde{\eta}.(\alpha, \chi))\{P'\}^{\chi}} \qquad \text{(Policy}_2\text{)}\quad \frac{P \xrightarrow{\alpha?r} P' \quad \tilde{\eta}.(\alpha, \chi) \not\models \varphi}{(r, \varphi, \tilde{\eta})\{P\}^{\chi} \xrightarrow{\overline{\alpha(r)}} (r, \varphi, \tilde{\eta}.(\overline{\alpha}, \chi))\{\mathbf{0}\}^{\jmath} \parallel P'}$$

$$\text{(Local)}\quad \frac{P \xrightarrow{\mu} P'}{(r, \varphi, \tilde{\eta})\{P\}^{\chi} \xrightarrow{\mu} (r, \varphi, \tilde{\eta})\{P'\}^{\chi}}\ \mu \notin (\mathscr{A} \cup \{rel\})(r)$$

Table 2: Instrumented operational semantics.

We are now ready to introduce our CFA. The result of analysing a process P is a triple (ρ, κ, Γ), called *estimate* of P, that provides a safe approximation of the process behavior, by capturing information on the possible processes P may evolve into. As the CFA for the π-calculus [19], ours focus on the use of channels and of resources. More precisely, ρ and κ offer an over-approximation of all the possible values that the variables in the system may be bound to, and of the values that may flow on channels. Furthermore, to statically check resource usage against the required policies, we have the further component Γ, which provides a set of the possible traces of actions on each resource.

The analysis correctly captures the behaviour of P, i.e. the estimate (ρ, κ, Γ) is valid for all the derivatives P' of P. In particular, the analysis keeps track of the following approximations (our abstract domains), each collecting a different kind of information (the first two components are treated as in [19]):

- An approximation $\rho : \mathscr{N} \cup (\mathscr{R} \cup \mathscr{S}) \to \wp(\mathscr{N} \cup \mathscr{R})$ of names bindings. If $a \in \rho(x)$ then the channel variable x can assume the channel value a. Similarly, if $r \in \rho(s)$ then the resource variable s can assume the resource identifier r. Note that $\rho(r) = \{r\}$ for each resource identifier r.
- An approximation $\kappa : \mathscr{N} \to \wp(\mathscr{N} \cup \mathscr{R})$ of the values that can be sent on each channel. If $b \in \kappa(a)$, then the channel value b can be sent on the channel a, while if $r \in \kappa(a)$, then the resource identifier r can be sent on the channel a.
- An approximation $\Gamma : \mathscr{R} \to \wp(\{(\varphi, \tilde{\eta}) |\ \varphi \in \Phi, \tilde{\eta} \in \widehat{\mathscr{A}^*}\})$ of resource behavior. If $(\varphi, \tilde{\eta}) \in \Gamma(r)$ with $\tilde{\eta} = (\alpha, \chi_\alpha).(rel, \chi_\alpha).(\beta, \chi_\beta).(rel, \chi_\beta)$ then $\tilde{\eta}$ and *all its prefixes* are possible admissible traces over the resource r (with policy φ) performed by the sequence of sub-processes labelled by χ_α and χ_β.

Our analysis relies on the following auxiliary function H that, given a resource r and a boundary labelled by χ, extracts the sequence or extended trace of all the access and release actions on r. In a sense, H returns the declared usage of r of the resource boundary, to be checked against the policy of the required resource, once obtained it.

Definition 4.1 *We inductively define the function* $H : \mathscr{R} \times SeqProc \times \mathscr{L} \to \widehat{\mathscr{A}^*}$ *as follows*

$$\begin{array}{rcl}
H(r, \mathbf{0}, \chi) & = & \varepsilon \\
H(r, \pi.Q, \chi) & = & H(r, Q, \chi) \text{ if } \pi \text{ is an input/output prefix} \\
H(r, \alpha(r).Q, \chi) & = & (\alpha, \chi).H(r, Q, \chi) \\
H(r, \alpha(z).Q, \chi) & = & H(r, Q, \chi) \text{ if } z \neq r \\
H(r, rel(r).Q, \chi) & = & (rel, \chi) \\
H(r, rel(z).Q, \chi) & = & H(r, Q, \chi) \text{ if } z \neq r \\
H(r, (r', \varphi', \tilde{\eta}')\{Q'\}^{\chi'}, \chi) & = & H(r, Q', \chi) \\
H(r, req(z)\{Q'\}^{\chi'}, \chi) & = & H(r, Q', \chi) \text{ if } z \neq r \\
H(r, ((r', \varphi', \tilde{\eta}')\{\mathbf{0}\}^{\flat} \parallel Q), \chi) & = & H(r, Q, \chi)
\end{array}$$

Example 4.2 *In our running example, we can apply H to the boundary process obtained after the first transition, where we add the label* χ_α^1 *(in blue in the pdf) to the boundary:*

$$req(r_1)\{\alpha(r_1).rel(r_1)\}^{\chi_\alpha^1}$$

we have that

$$H(r_1, req(r_1)\{\alpha(r_1).rel(r_1)\}^{\chi_\alpha^1}, \chi_\alpha^1) = H(r_1, \alpha(r_1).rel(r_1), \chi_\alpha^1) = (\alpha, \chi_\alpha^1).(rel, \chi_\alpha^1)$$

At run time, the actual firing of the actions extracted by the function H depends on the history of accesses on the resource, and on the required policy. If the required resource is obtained, all the actions included in the extracted sequence are gradually appended to the current history of the resource, as long as they do not violate the policy. In case of violation, only the actions that precede the violation and the violation action itself are appended.

To statically mimic this incremental updating process of histories, the CFA relies on the further auxiliary function *Adm*, whose application in the analysis will be explained in the next paragraph. Given two traces $\tilde{\eta}, \tilde{\eta}'$ and a policy φ, *Adm* returns the concatenation of $\tilde{\eta}$ with the *maximum* prefix of $\tilde{\eta}'$ that can be appended to $\tilde{\eta}$, without violating φ, followed by the first violation action, if any. We can have two cases: (i) either the appended prefix coincides with the whole second trace $\tilde{\eta}'$, (ii) or the appended prefix coincides only with the part of the second trace that precedes the violation and terminates with the violation action.

Definition 4.3 *We inductively define the function* $Adm : \widehat{\mathscr{A}^*} \times \widehat{\mathscr{A}^*} \to \widehat{\mathscr{A}^*}$ *as follows*

$$Adm(\varphi, \tilde{\eta}, \tilde{\eta}') = \begin{cases} \tilde{\eta}.\tilde{\eta}' & \text{if } \tilde{\eta}.\tilde{\eta}' \models \varphi \\ \tilde{\eta}.\tilde{\eta}''.(\overline{\alpha}, \chi) & \text{if } \tilde{\eta}.\tilde{\eta}'' \models \varphi \wedge \tilde{\eta}.\tilde{\eta}''.(\alpha,\chi) \not\models \varphi \text{ where } \tilde{\eta}' = \tilde{\eta}''.(\alpha,\chi).\tilde{\eta}''' \end{cases}$$

Validation To *validate* the correctness of a proposed estimate (ρ, κ, Γ), we introduce a set of clauses that operate upon judgments in the form $(\rho, \kappa, \Gamma) \models P$, specified in terms of Flow Logic [56]. The judgement $(\rho, \kappa, \Gamma) \models P$ expresses that the components ρ, κ, and Γ provide a valid analysis result for the behaviour of P. Note that the validation process does not say how to compute the analysis result itself. The judgments of the CFA, given in Tables 3 and 4, are defined inductively in the structure of processes, by presenting a clause for each syntactic construct. Table 3 illustrates the clauses for the π-calculus constructs, while in Table 4, we focus on the constructs that handle resources.

$[\mathbf{Nil}_\alpha]$	$(\rho, \kappa, \Gamma) \models \mathbf{0}$	iff true
$[\mathbf{Out}_\alpha]$	$(\rho, \kappa, \Gamma) \models \bar{x}y.S$	iff $\forall a \in \rho(x) : \rho(y) \subseteq \kappa(a) \ \wedge \ (\rho, \kappa, \Gamma) \models S$
$[\mathbf{In}_\alpha]$	$(\rho, \kappa, \Gamma) \models x(y).S$	iff $\forall a \in \rho(x) : \kappa(a) \cap \mathscr{N} \subseteq \rho(y) \ \wedge \ (\rho, \kappa, \Gamma) \models S$
$[\mathbf{Sum}_\alpha]$	$(\rho, \kappa, \Gamma) \models P_1 + P_2$	iff $(\rho, \kappa, \Gamma) \models P_1 \wedge (\rho, \kappa, \Gamma) \models P_2$
$[\mathbf{Par}_\alpha]$	$(\rho, \kappa, \Gamma) \models P_1 \parallel P_2$	iff $(\rho, \kappa, \Gamma) \models P_1 \wedge (\rho, \kappa, \Gamma) \models P_2$
$[\mathbf{Res}_\alpha]$	$(\rho, \kappa, \Gamma) \models (\nu x)P$	iff $(\rho, \kappa, \Gamma) \models P \wedge x \in \rho(x)$

Table 3: CFA rules for the π-calculus part

We first comment the clauses in Table 3. All the clauses dealing with a compound process check that the analysis also holds for its immediate sub-processes. In particular, the analysis of $(\nu x)P$ is equal to the one of P, in rule $[\mathbf{Res}_\alpha]$, while the analysis for $P_1 + P_2$ and $P_1 \parallel P_2$, in rules $[\mathbf{Sum}_\alpha]$ and $[\mathbf{Par}_\alpha]$ are equal to the analysis of P_1 and of P_2. We comment on the main rules. Besides the validation of the continuation process S (S can be a process or a sequential process), the rule for output $[\mathbf{Out}_\alpha]$ requires that the set of names, which can be communicated along each element of $\rho(x)$, includes the names to which y can evaluate. Symmetrically, the rule for input $[\mathbf{In}_\alpha]$ demands that the sets of names that can pass along the name variable x are included in the sets of names to which y can evaluate, provided that the passed names belong to $\mathscr{N}$.

Intuitively, the estimate components take into account the possible dynamics of the process under consideration. The clauses' checks mimic the semantic evolution, by modelling the semantic preconditions and the consequences of the possible synchronisations. In the rule for input $[\mathbf{In}_\alpha]$, e.g., CFA checks whether the precondition of a synchronisation is satisfied, i.e. whether there is a corresponding output possibly sending a value that can be received by the analysed input. To give a valid prediction of the analysed synchronisation action, the conclusion imposes that the variable y can be bound to that value.

Example 4.4 *Consider the following process, where no resources are handled:*

$$P = (\bar{a}d + \bar{a}b) \parallel a(w).\bar{c}w$$

$[\mathbf{OutR}_\alpha]$	$(\rho,\kappa,\Gamma) \models \bar{x}z.S$	iff $\forall a \in \rho(x) : \rho(z) \subseteq \kappa(a) \ \wedge \ (\rho,\kappa,\Gamma) \models S$
$[\mathbf{InR}_\alpha]$	$(\rho,\kappa,\Gamma) \models x(s).S$	iff $\forall a \in \rho(x) : (\kappa(a) \cap \mathscr{R}) \setminus R \subseteq \rho(s)$ $\wedge \ \forall r \in \rho(s) \ s.t. \ r \notin \mathsf{fr}(P) : (\rho,\kappa,\Gamma) \models S\{r/s\}_R$
$[\mathbf{Joint}^{\mathbf{1}}_\alpha]$	$(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^\flat$	iff $(\rho,\kappa,\Gamma) \models \mathbf{0} \ \wedge \ (\varphi,\tilde{\eta}) \in \Gamma(r)$
$[\mathbf{Joint}^{\mathbf{2}}_\alpha]$	$(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^\flat \parallel Q$	iff $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^\flat \wedge (\rho,\kappa,\Gamma) \models Q$
$[\mathbf{Joint}^{\mathbf{3}}_\alpha]$	$(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{Q\}^\chi$	iff $(\rho,\kappa,\Gamma) \models Q \ \wedge \ (\varphi, Adm(\varphi,\tilde{\eta},\tilde{\eta}')) \in \Gamma(r)$ where $\tilde{\eta}' = H(r,Q,\chi)$
$[\mathbf{Req}_\alpha]$	$(\rho,\kappa,\Gamma) \models req(r)\{Q\}^\chi$	iff $(\rho,\kappa,\Gamma) \models Q \wedge$ $\forall(\varphi,\tilde{\eta}) \in \Gamma(r).\chi \not\leq \tilde{\eta}. \ (\varphi, Adm(\varphi,\tilde{\eta},\tilde{\eta}')) \in \Gamma(r))$ where $\tilde{\eta}' = H(r,Q,\chi)$
$[\mathbf{Alpha}_\alpha]$	$(\rho,\kappa,\Gamma) \models \alpha(r).Q$	iff $(\rho,\kappa,\Gamma) \models Q$
$[\mathbf{Rel}_\alpha]$	$(\rho,\kappa,\Gamma) \models rel(r).Q$	iff $(\rho,\kappa,\Gamma) \models Q$

Table 4: CFA rules for the resource treatment

A possible estimate is given by the following entries (Γ is empty, because there are no resources):

- $\rho(w) \supseteq \{b,d\}$,
- $\kappa(a) \supseteq \{b,d\}$, *and* $\kappa(c) \supseteq \{b,d\}$.

A simple check shows that the proposed estimate is valid. More in detail, by rules $[\mathbf{Par}_\alpha]$ *and* $[\mathbf{Sum}_\alpha]$, $(\rho,\kappa,\Gamma) \models P$ *if and only if we have that* $(\rho,\kappa,\Gamma) \models \bar{a}d$, $(\rho,\kappa,\Gamma) \models \bar{a}b$, *and* $(\rho,\kappa,\Gamma) \models a(w).\bar{c}w$. *The checks in the rule* $[\mathbf{Out}_\alpha]$ *imply that b and d belong to* $\kappa(a)$, *while the checks in rule* $[\mathbf{In}_\alpha]$ *imply that* $\kappa(a) \subseteq \rho(w)$ *and therefore that b and d belong to* $\rho(w)$. *Finally, the checks in the rule* $[\mathbf{Out}_\alpha]$ *for the continuation* $\bar{c}w$ *imply that* $\rho(w) \subseteq \kappa(c)$, *and therefore that b and d belong to* $\kappa(c)$.

We can now focus on the rules for the management of resources of Table 4. The rules for input and output of resource identifiers, $[\mathbf{OutR}_\alpha]$ and $[\mathbf{InR}_\alpha]$ are similar to the corresponding rules for names in Table 3. Of course, in the input rule, the names that can be bound the resource variable must belong to $\mathscr{R}$ and, to mimic the side condition of rule $(Comm_R)$, r does not belong to the free resource identifiers $\mathsf{fr}(P)$ of the continuation P. Furthermore, the rule requires performing, for every actual value r that can be bound to s, a distinct check the corresponding possible continuations $P\{r/s\}_R$, thus allowing us to gain precision. As a consequence, in all the other rules, there are no open resource variables s.

The rule for the empty *resource joint point* $[\mathbf{Joint}^{\mathbf{1}}_\alpha]$ amounts to checking that the inclusion of the pair $(\varphi,\tilde{\eta})$ in the component Γ for r, while the rule $[\mathbf{Joint}^{\mathbf{2}}_\alpha]$ for the composition between *resource joint point* and another process Q simply amounts to the composition of their analyses.

The rule $[\mathbf{Joint}^3_\alpha]$ for non-empty *resource joint point* $(r,\varphi,\tilde{\eta})\{Q\}^\chi$ (where $Q \neq \emptyset$ and $\chi \neq \varepsilon$) deserves more comments. The rule checks whether $\Gamma(r)$ includes the right traces of usage of the resource r, due to the process Q. More precisely,

- given the trace $\tilde{\eta}'$ syntactically computed by $H(r,Q,\chi)$, i.e. the trace of accesses that Q is willing to perform on r,
- by exploiting the result of the function *Adm*, the rule checks whether the admissible part of the trace $\tilde{\eta}'$ is recorded in $\Gamma(r)$, appended to the current trace $\tilde{\eta}$. More in detail, if the trace $\tilde{\eta}'$ never violates the policy φ, the whole trace must be appended to $\tilde{\eta}$, i.e. $(\varphi,\tilde{\eta}.\tilde{\eta}') \in \Gamma(r)$. Otherwise $(\varphi,\tilde{\eta}.\tilde{\eta}''.\overline{\alpha}) \in \Gamma(r)$, where $\tilde{\eta}''$ is the maximum prefix of $\tilde{\eta}'$, such that appended to $\tilde{\eta}''$ respects the policy φ, and $\overline{\alpha(r)}$ is the first possible violation action of the policy.
- Finally, the rule includes the validation of the enclosed process Q.

Similarly, the rule for *resource request point* $[\mathbf{Req}_\alpha]$, includes the validation of the enclosed process Q and checks, for each trace in $\Gamma(r)$ of previous accesses on r, its composition with the trace extracted from $H(r,Q,\chi)$ against φ and checks whether the right part of traces are included in the component Γ for r. The traces $\tilde{\eta}$ in $\Gamma(r)$ that we compose must not include χ (in symbols $\chi \not\in \tilde{\eta}$), i.e. we only consider accesses made by processes different from the process labelled by χ.

The analysis of resource prefix actions $[\mathbf{Alpha}_\alpha]$ and $[\mathbf{Rel}_\alpha]$ amounts to the simple validation of the continuation process Q.

Example 4.5 *We briefly apply the analysis to our running example, where the processes are enriched with labels (in blue in the pdf). First, we associate labels with the resource boundaries as follows:*

$$\begin{array}{rl}
Res ::= & (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \\
Users ::= & x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\}^{\chi^1_\alpha} | x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\}^{\chi^2_\alpha} \parallel \\
& x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\}^{\chi^3_\alpha} \parallel y(t).req(t)\{\beta(t).rel(t)\}^{\chi_\beta} \\
Plan ::= & \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0} \\
System ::= & Res \parallel Users \parallel Plan
\end{array}$$

To illustrate how the labels work, we recall one of the possible system transitions.

$$\begin{array}{l}
System \\
\equiv Res \parallel Users' \parallel x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\}^{\chi^1_\alpha} \parallel \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0} \\
\xrightarrow{\tau} (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel Users' \parallel Plan' \parallel req(r_1)\{\alpha(r_1).rel(r_1)\}^{\chi^1_\alpha}
\end{array}$$

Further possible transitions leads to

$$\begin{array}{rl}
\xrightarrow{\tau} & (r_1,\varphi_1,\varepsilon)\{\alpha(r_1).rel(r_1)\}^{\chi^1_\alpha} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel Users' \parallel Plan' \\
\xrightarrow{\tau} & (r_1,\varphi_1,(\alpha,\chi^1_\alpha))\{rel(r_1)\}^{\chi^1_\alpha} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel Users' \parallel Plan' \\
\xrightarrow{\tau} & (r_1,\varphi_1,(\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha))\{\mathbf{0}\}^{\chi^1_\alpha} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel Users' \parallel Plan' \\
\equiv & (r_1,\varphi_1,(\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha)))\{\mathbf{0}\}^{\flat} \parallel (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel (r_2,\varphi_2,\varepsilon)\{\mathbf{0}\}^{\flat} \parallel Users' \parallel Plan'
\end{array}$$

The CFA entries include:

- $\rho(s_1) \supseteq \{r_1\}$, $\rho(s_2) \supseteq \{r_2\}$, $\rho(s_3) \supseteq \{r_1\}$, $\rho(t) \supseteq \{r_2\}$; *correspondingly:*
- $\kappa(x_1) \supseteq \{r_1\}$, $\rho(x_2) \supseteq \{r_2\}$, $\rho(x_3) \supseteq \{r_1\}$, $\rho(y) \supseteq \{r_2\}$;

- $\Gamma(r_1) \supseteq \{(\varphi_1,\varepsilon),(\varphi_1,(\alpha,\chi^i_\alpha).(rel,\chi^i_\alpha)),(\varphi_1,(\alpha,\chi^j_\alpha).(rel,\chi^j_\alpha).(\alpha,\chi^i_\alpha).(rel,\chi^i_\alpha)),$
  $(\varphi_1,(\alpha,\chi^i_\alpha).(rel,\chi^i_\alpha).(\alpha,\chi^j_\alpha).(rel,\chi^j_\alpha))\}$ *(with* $i,j \in \{1,3\}$ *and* i,j *distinct);*
- $\Gamma(r_2) \supseteq \{(\varphi_2,\varepsilon),(\varphi_2,(\beta,\chi_\beta).(rel,\chi_\beta)),(\varphi_2,(\beta,\chi_\beta).(rel,\chi_\beta).(\overline{\alpha},\chi^2_\alpha)),$
  $(\varphi_2,(\alpha,\chi^2_\alpha).(rel,\chi^2_\alpha)),(\varphi_2,(\alpha,\chi^2_\alpha).(rel,\chi_\alpha).(\overline{\beta},\chi_\beta))\}$*;*

To illustrate our analysis, we show some successful CFA checks performed on the above results, where $Users' ::= x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\}^{\chi^2_\alpha} \| x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\}^{\chi^3_\alpha} \| y(t).req(t)\{\beta(t).rel(t)\}^{\chi_\beta}$ *and* $Plan' ::= \bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0}$*. Since, by rule* $[\mathbf{Par}_\alpha]$*,* $(\rho,\kappa,\Gamma) \models System$ *if and only if we have that* $(\rho,\kappa,\Gamma) \models Res$*,* $(\rho,\kappa,\Gamma) \models Users$*, and* $(\rho,\kappa,\Gamma) \models Plan$*, we have to perform the checks in the rules*

- $[\mathbf{Joint}^{\mathbf{1}}_\alpha]$ *for the empty resource joint point, where checking* $(\rho,\kappa,\Gamma) \models (r_1,\varphi_1,\varepsilon)\{\mathbf{0}\}^\flat$ *implies that* $(\rho,\kappa,\Gamma) \models \{\mathbf{0}\}^\flat$ *and* $(\varphi_1,\varepsilon) \in \Gamma(r_1)$*.*
- $[\mathbf{InR}_\alpha]$ *for input actions, where* $(\rho,\kappa,\Gamma) \models x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\}^{\chi^1_\alpha} \| Users'$ *implies that* $r_1 \in \rho(s_1)$*, and*
- $[\mathbf{Req}_\alpha]$ *for resource requests, where* $(\rho,\kappa,\Gamma) \models req(r_1)\{\alpha(r_1).rel(r_1)\}^{\chi^1_\alpha}$ *implies that* $\Gamma(r_1)$ *includes* $(\varphi_1,(\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha))$*. Given indeed* $H(r_1,\alpha(r_1).rel(r_1),\chi^1_\alpha) = (\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha)$*, since* $\varphi_1 \models (\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha)$*, we have that* $Adm(\varphi_1,\varepsilon,(\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha)) = (\alpha,\chi^1_\alpha).(rel,\chi^1_\alpha)$*.*
- $[\mathbf{Out}_\alpha]$ *for output actions, as in* $(\rho,\kappa,\Gamma) \models \bar{x_1}\langle r_1\rangle.Plan'$ *that implies that* $r_1 \in \kappa(x_1)$ *and that* $(\rho,\kappa,\Gamma) \models Plan'$*.*

Existence We presented a procedure for validating whether or not a proposed estimate (ρ,κ,Γ) is acceptable. We now show that there always exists a least choice of (ρ,κ,Γ) that is acceptable for CFA rules, and therefore there always exists a least estimate. With "least" we intend with respect to the partial order defined on the estimates, by set inclusion, componentwise, as follows.

Definition 4.6 *A set of proposed estimates is ordered by setting* $(\rho,\kappa,\Gamma) \sqsubseteq (\rho',\kappa',\Gamma')$ *iff* $\forall w \in \mathscr{N} \cup \mathscr{R}$: $\rho(w) \subseteq \rho'(w)$*,* $\forall a \in \mathscr{N} : \kappa(a) \subseteq \kappa'(a)$*,* $\forall r \in \mathscr{R} : \Gamma(r) \subseteq \Gamma'(r)$*.*

To prove the existence of a least estimate we rely on the fact that the set of proposed estimates is a *Moore family*, i.e. it is a subset of a complete lattice closed under greatest lower bounds [57]. The formal definition is the following.

Definition 4.7 *A subset* Y *of a complete lattice* $L = (L,\sqsubseteq)$ *is a* Moore family *if and only if it contains* $(\sqcap Y')$ *for all* $Y' \subseteq Y$*.*

Theorem 4.8 (Existence of estimates) *For all the processes* P*, the set* $\{(\rho,\kappa,\Gamma) | (\rho,\kappa,\Gamma) \models P\}$ *is a Moore family.*

The above theorem guarantees that there always exists a least estimate to the specification in Tables 3 and 4, since the set $\{(\rho,\kappa,\Gamma) | (\rho,\kappa,\Gamma) \models P\}$ is a subset of the Moore family and $\sqcap\{(\rho,\kappa,\Gamma) | (\rho,\kappa,\Gamma) \models P\}$ is still an estimate.

Our analysis can be implemented along the lines of the CFA for the π-calculus [19]. The only difference is due to the rule for checking inputs of resource identifiers. As a matter of fact, checking for every actual value r that can be bound to the variable s allows us to gain precision, since there is a distinct check for each possible continuation $P\{r/s\}_R$. These checks introduce an exponential factor in the complexity of the algorithm, which can be reasonably kept low, by limiting the number of possible resources.

Correctness The analysis provides us with an approximation of the overall behaviour of the analysed process. Moreover, the analysis respects the operational semantics of G-Local π-calculus. To prove the semantic correctness, we need the following auxiliary results. The first ones state that estimates are resistant to substitution of closed terms for variables.

Fact 4.9 *Given an estimate* (ρ,κ,Γ) *and* $v \in \rho(x)$*, we have that* $\forall y \in \mathcal{N} : \rho(y(\{v/x\})) \subseteq \rho(y)$

Lemma 4.10 ***(Substitution)*** *If* $(\rho,\kappa,\Gamma) \models P$ *then* $(\rho,\kappa,\Gamma) \models P\{v/x\}$*, provided that* $v \in \rho(x)$*.*

The following lemma says that an estimate for a process P is also a valid estimate for every process congruent to P.

Lemma 4.11 ***(Congruence)*** *If* $(\rho,\kappa,\Gamma) \models P$ *and* $P \equiv Q$*, then* $(\rho,\kappa,\Gamma) \models Q$*.*

We are now ready to state the subject reduction result that proves the semantics correctness of the given analysis. First, we prove its correctness for its immediate derivatives, where a derivative is a single-step evolution of the process P.

Theorem 4.12 ***(Subject reduction)*** *If* $P \xrightarrow{\mu} P'$*, and* $(\rho,\kappa,\Gamma) \models P$*, then* $(\rho,\kappa,\Gamma) \models P'$*.*

The correctness for all the derivatives of P immediately follows from the above theorem.

Corollary 4.13 $(\rho,\kappa,\Gamma) \models P$ *and* $P \xrightarrow{\mu}{}^{*} P'$*, then* $(\rho,\kappa,\Gamma) \models P'$*.*

Policy compliance Our analysis computes information on the resource usage and correctly predicts possible cases of bad usage. In particular, this information can be statically exploited to check whether a process complies with a given policy.

We first give a dynamic characterisation of what we mean with a process P to comply with a given policy φ over the resource r. A process enjoys this property if neither it nor any of its derivatives can perform an invalid access according to φ. Formally, we have the following definition, where $\xrightarrow{\mu}{}^{*}$ denotes the reflexive and transitive closure of $\xrightarrow{\mu}$.

Definition 4.14 *The process P, where r is declared with policy* φ*,* complies with φ *for r, if and only if* $P \xrightarrow{\mu}{}^{*} P'$ *implies that there is no* P'' *such that* $P' \xrightarrow{\overline{\alpha(r)}} P''$*.*

The information included in the component Γ allows us to define a static notion that corresponds to the above dynamic property. We first observe that the component Γ of our CFA is in charge of recording all the possible usage traces on each resource r. Actually, given r, $\Gamma(r)$ is composed of pairs $(\varphi,\tilde{\eta})$, where the trace $\tilde{\eta}$ records every action on r. Traces can also include pairs $(\overline{\alpha},\chi)$ that refer to actions α that correspond to possible "violations". More precisely, the presence of a pair $(\overline{\alpha},\chi)$ indicates that the sub-process labelled by χ may be forced to release the resource r, when trying to perform the non-allowed action α on r. We call *faulty* traces the traces that include violation actions.

Definition 4.15 *A static trace* $\tilde{\eta} \in \widehat{\mathscr{A}^*}$ *is* faulty *if it includes* $(\overline{\alpha},\chi)$ *for some* $\alpha \in \mathscr{A}$ *and for some* $\chi \in \mathscr{L}$*.*

On the static side a process respects a policy for a resource r, when we cannot find any faulty trace in the traces included in the component Γ.

Definition 4.16 *A process P, where r is declared with policy* φ*, is said to* respect φ *for r, if and only if*

$$\exists(\rho,\kappa,\Gamma).(\rho,\kappa,\Gamma) \models P \text{ and } \forall(\varphi,\tilde{\eta}) \in \Gamma(r).\tilde{\eta} \text{ is not faulty.}$$

By suitably handling the safe over-approximation the CFA introduces, we can predict at static time whether a process complies with its policy. Because of the over-approximation, the analysis can give false positives (what the analysis includes corresponds to something that *can happen*), but it can never give false negatives (what the analysis does *not* include cannot happen). As far as policy compliance is concerned, this means that if all the traces are not faulty, then we can prove that policy violations cannot occur at run time, and therefore that the process correctly uses its resources. As a consequence, the absence of faulty traces in the analysis results is a sufficient condition for the checked process to comply with its policy.

Theorem 4.17 *If P* respects *the policy φ for r then P* complies with *φ.*

Instead, if the analysis contains faulty traces, then there is the *possibility* of policy violations, as discussed in the following example.

Example 4.18 *Back to our running example, and to its analysis result, it is easy to see that there are at least two possible policy violations, which are captured by our CFA in the component $\Gamma(r_2)$. The first faulty trace, given by the entry:*

$$(\varphi_2, (\beta, \chi_\beta).(rel, \chi_\beta).(\overline{\alpha}, \chi_\alpha))$$

corresponds to the dynamic computation, developed in the previous section, where the resource r_2 is forcedly released, when one of the user attempts to perform an α action on r_2, after an action β. The corresponding trace $(\beta, \chi_\beta).(rel, \chi_\beta).(\overline{\alpha}, \chi_\alpha)$ records indeed the tentative violation of the policy φ_2 on the resource r_2. Analogously, the second faulty trace, given by the entry

$$(\varphi_2, (\alpha, \chi_\alpha^2).(rel, \chi_\alpha).(\overline{\beta}, \chi_\beta))$$

represents a similar violation of the policy φ_2 on the resource r_2. In these particular cases, both traces reflect the dynamic behaviour, where the accesses to the resource r_2 of both sub-processes lead to sure run time violations. If the whole system offered a further copy of the resource r_2, as in

$$Res' = (r_1, \varphi_1, \varepsilon)\{\mathbf{0}\}^\flat \parallel (r_1, \varphi_1, \varepsilon)\{\mathbf{0}\}^\flat \parallel (r_2, \varphi_2, \varepsilon)\{\mathbf{0}\}^\flat \parallel (r_2, \varphi_2, \varepsilon)\{\mathbf{0}\}^\flat$$

then we would have at least some computations without violation but, still, the whole system would not comply with the policy φ_2. Consider instead the slightly modified version of the system in the running example.

$$\begin{aligned}
Res_1 ::= \;& (r_1, \varphi_1, \varepsilon)\{\mathbf{0}\}^\flat \parallel (r_1, \varphi_1, \varepsilon)\{\mathbf{0}\}^\flat \parallel (r_2, \varphi_2, \varepsilon)\{\mathbf{0}\}^\flat \\
Users_1 ::= \;& x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\}^{\chi_\alpha^1} | (\bar{x_2}\langle r_2\rangle.x_2(s_2)).req(s_2)\{\alpha(s_2).rel(s_2)\}^{\chi_\alpha^2} \parallel \\
& x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\}^{\chi_\alpha^3} \parallel y(t).req(t)\{\beta(t).rel(t)\}^{\chi_\beta} \\
Plan_1 ::= \;& \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0} \\
System_1 ::= \;& Res \parallel Users_1 \parallel Plan_1
\end{aligned}$$

where the output prefix $\bar{x_2}\langle r_2\rangle$ is now put before the corresponding input $x_2(s_2)$. Since the two prefixes are placed in sequence they cannot synchronise. Nevertheless, because of the over-approximation, the analysis still counts r_2 among the possible values in $\rho(s_2)$. As a result, we still have faulty traces, but $System_1$ dynamically complies with φ_2 for r_2.

$$
\begin{array}{lcll}
P,P' & ::= & \textit{processes} \in \textit{Proc} & \\
 & | & (r,\varphi,\check{\eta})\{F\}^{\chi} & \text{resource joint point} \\
 & | & req(z)\{F\}^{\chi} & \text{resource request point} \\
 & & & \\
F,F' & ::= & \textit{flat parallel processes} \in \textit{FlProc} & \\
 & & \mathbf{0} & \\
 & | & \alpha^{\lambda}(z).Q & \\
 & | & rel^{\lambda}(z).Q & \\
 & | & \pi.Q & \\
 & | & \Pi_{i=0}^{n} F_i & \\
 & | & (r,\varphi,\check{\eta})\{F\}^{\chi} & \\
 & | & (r,\varphi,\check{\eta})\{\mathbf{0}\}^{\flat}||F & \\
 & | & req(z)\{F\}^{\chi} &
\end{array}
$$

Figure 7: Labelled syntax for parallel threads.

4.2 CFA: step 2

We now introduce the necessary extensions to apply our Control Flow Analysis to the case of parallel threads. As done above, to make the control flow analysis tractable we assume that our static machinery applies to *deadlock-free* systems.

Furthermore, for the sake of simplicity, we suppose to have a flat structure of parallel processes, as established by the syntax in Figure 7, where we use the indexed product as n-ary parallel constructor. The general case can be similarly handled, at the cost of some slightly more involved notational extensions. For an example, see next section, where we introduce suitable forms of unbounded iterators over resources.

Differently from the previous case, a group of sequential processes can now concurrently access and use the same resource r. As a consequence the usage traces of r are obtained by interleaving the single contributions of each process.

To statically obtain the corresponding new usage traces, we need to extract (from the resource boundaries) the single sequences of the access and release actions on r and then merging or shuffling the obtained traces, without altering the order of access events inside the single starting traces.

Before introducing the shuffling function *Shfl* for merging traces, we need some auxiliary definitions. We first need to distinguish actions with the same name, but performed in different moments by different processes. To this aim, we associate a further label $\lambda \in \Lambda$, different in each point, with every action α, i.e. actions are in the form $\alpha^{\lambda}(r)$ and $rel^{\lambda}(r)$. The traces are extended accordingly and are composed by triples like (α,χ,λ). The new traces, called $\check{\eta}$, are included indeed in $\check{\mathscr{A}}^*$, where $\check{\mathscr{A}} = (\mathscr{A} \cup \{rel\} \times \mathscr{L} \times \Lambda)$. Again these labels can be easily forgotten and removed to obtain the original semantics.

Then, we need a way to preserve the order of access events inside the single starting traces in the resulting shuffled traces. This requires establishing an order between the components of the same trace, based on the order in which they occur. Trace components are obtained by applying an auxiliary function $\mathbf{L}$ that, given an extended trace $\check{\eta}$, returns the union of all of its components (α,χ,λ).

Definition 4.19 *Given a sequence $\check{\eta} \in (\mathscr{A} \cup \{rel\}) \times \mathscr{L} \times \Lambda)^*$, we define $\mathbf{L}(\check{\eta})$ as follows:*

$$
\begin{array}{lcl}
\mathbf{L}(\varepsilon) & = & \emptyset \\
\mathbf{L}((\alpha,\chi,\lambda).\check{\eta}) & = & \{(\alpha,\chi,\lambda)\} \cup \mathbf{L}(\check{\eta})
\end{array}
$$

Given a sequence $\check{\eta} \in (\mathscr{A} \cup \{rel\}) \times \mathscr{L} \times \Lambda)^*$ *and two components* (α,χ,λ), (β,χ',μ) *in* $\check{\eta}$, *we say that* (α,χ,λ) occurs before (β,χ',μ) *in* $\check{\eta}$, *written as*

$$(\alpha,\chi,\lambda) <_{\check{\eta}} (\beta,\chi',\mu)$$

if and only if $\check{\eta} = \check{\eta}'.(\alpha,\chi,\lambda).\check{\eta}''.(\beta,\chi',\mu).\check{\eta}'''$, *where* $\check{\eta}'.\check{\eta}'',\check{\eta}''' \in ((\mathscr{A} \cup \{rel\}) \times \mathscr{L} \times \Lambda))^*$.
Given sequences $\check{\eta}_0,\check{\eta}_1,...\check{\eta}_k \in ((\mathscr{A} \cup \{rel\}) \times \mathscr{L} \times \Lambda))^*$, *we define*

$$Shfl(\check{\eta}_0,...,\check{\eta}_k) = \{\check{\eta} \in \mathbf{L}(\check{\eta}_0) \cup ... \cup \mathbf{L}(\check{\eta}_k) | \forall i \in [0,k], \forall (\alpha,\chi,\lambda),(\beta,\chi',\mu) \in \check{\eta}_i :$$
$$(\alpha,\chi,\lambda) <_{\check{\eta}_i} (\beta,\chi',\mu) \Rightarrow (\alpha,\chi,\lambda) <_{\check{\eta}} (\beta,\chi',\mu)\}$$

At this point, we can introduce the function SH that, given a flat process F, a resource boundary labelled by χ and a resource r, computes the shuffling of all the traces due to every sequential subprocesses of F, computed by exploiting an auxiliary function H' that extends H to the new traces.

Definition 4.20 *We inductively define the function* $SH : \mathscr{F} \times FlProc \times \mathscr{L} \to \check{\mathscr{A}}^*$ *as follows*

$$\begin{array}{lcl} SH(r,F_0 \parallel ... \parallel F_n,\chi) & = & Shfl(H'(r,F_0,\chi),...,H'(r,F_n,\chi)) \\ SH(r,\mathbf{0},\chi) & = & \emptyset \\ SH(r,F,\chi) & = & \{H'(r,F,\chi)\} \end{array}$$

where the function $H' : \mathscr{R} \times FlProc \times \mathscr{L} \to \check{\mathscr{A}}^*$ *is defined as follows*

$$\begin{array}{rcl} H'(r,\mathbf{0},\chi) & = & \varepsilon \\ H'(r,\pi.F,\chi) & = & H'(r,F,\chi) \text{ if } \pi \text{ is an input/output prefix} \\ H'(r,\alpha^{\lambda}(r).F,\chi) & = & (\alpha,\chi,\lambda).H'(r,F,\chi) \\ H'(r,\alpha^{\lambda}(z).F,\chi) & = & H'(r,F,\chi) \text{ if } z \neq r \\ H(r,rel^{\lambda}(r).F,\chi) & = & (rel,\chi,\lambda) \\ H(r,rel^{\lambda}(z).F,\chi) & = & H'(r,F,\chi) \text{ if } z \neq r \\ H'(r,(r',\varphi',\check{\eta}')\{F'\}^{\chi'},\chi) & = & H'(r,F',\chi) \\ H'(r,req(z)\{F'\}^{\chi'},\chi) & = & H'(r,F',\chi) \text{ if } z \neq r \\ H'(r,((r',\varphi',\check{\eta}')\{\mathbf{0}\}^{\flat} \parallel F),\chi) & = & H'(r,F,\chi) \end{array}$$

Example 4.21 *If we apply the function SH to the following process F, where we put in blue (in the pdf) the second flat parallel process and its contributions*

$$F = req(r)\{\alpha^{\lambda_1}(r).\beta(r)^{\lambda_1'}.rel^{\lambda_1''}(r) \parallel \gamma^{\lambda_2}(r).\alpha^{\lambda_2'}(r).rel^{\lambda_2''}(r)\}^{\chi}$$

we obtain

$$SH(r,F,\chi) = Shfl(H'(r,\alpha^{\lambda_1}(r).\beta(r)^{\lambda_1'}.rel^{\lambda_1''}(r),\chi),(H'(r,\gamma^{\lambda_2}(r).\alpha^{\lambda_2'}(r).rel^{\lambda_2''}(r),\chi))$$

Now, since

$$H'(r,\alpha^{\lambda_1}(r).\beta(r)^{\lambda_1'}.rel^{\lambda_1''}(r),\chi) = (\alpha,\chi,\lambda_1).(\beta,\chi,\lambda_1').(rel,\chi,\lambda_1'')$$
$$H'(r,\gamma^{\lambda_2}(r).\alpha^{\lambda_2'}(r).rel^{\lambda_2''}(r),\chi) = (\gamma,\chi,\lambda_2).(\alpha,\chi,\lambda_2').(rel,\chi,\lambda_2'')$$

we have that $SH(F)$ *includes all the possible shuffles, e.g. the following ones:*

$$(\alpha,\chi,\lambda_1).(\beta,\chi,\lambda_1').(rel,\chi,\lambda_1'').(\gamma,\chi,\lambda_2').(\alpha,\chi,\lambda_2').(rel,\chi,\lambda_2'')$$
$$(\alpha,\chi,\lambda_1).(\gamma,\chi,\lambda_2').(\beta,\chi,\lambda_1').(\gamma,\chi,\lambda_2').(rel,\chi,\lambda_1'').(rel,\chi,\lambda_2'')$$
$$(\alpha,\chi,\lambda_1).(\gamma,\chi,\lambda_2').(\gamma,\chi,\lambda_2').(\beta,\chi,\lambda_1').(rel,\chi,\lambda_1'').(rel,\chi,\lambda_2'')$$
$$(\gamma,\chi,\lambda_2').(\alpha,\chi,\lambda_2').(rel,\chi,\lambda_2'').(\alpha,\chi,\lambda_1).(\beta,\chi,\lambda_1').(rel,\chi,\lambda_1'')$$

$[\mathbf{Joint2}^{\mathbf{3}}_{\alpha}]$	$(\rho,\kappa,\Gamma) \models (r,\varphi,\check{\eta})\{F\}^{\chi}$	iff $(\rho,\kappa,\Gamma) \models F \ \wedge\ \forall \check{\eta}' \in SH(r,F,\chi).(\varphi, Adm(\varphi,\check{\eta},\check{\eta}') \in \Gamma(r))$
$[\mathbf{Req2}_{\alpha}]$	$(\rho,\kappa,\Gamma) \models req(r)\{F\}^{\chi}$	iff $(\rho,\kappa,\Gamma) \models F \ \wedge\ \forall(\varphi,\check{\eta}) \in \Gamma(r).\chi \notin \check{\eta} \ \wedge\ \forall \check{\eta}' \in SH(r,F,\chi).$ $(\varphi, Adm(\varphi,\check{\eta},\check{\eta}')) \in \Gamma(r)$

Table 5: CFA rules for parallel threads: the most interesting cases

Similarly, we extend the function Adm to treat the new traces, by introducing the function Adm'.

Definition 4.22 *We inductively define the function* $Adm' : \check{\mathscr{A}}^* \times \check{\mathscr{A}}^* \to \check{\mathscr{A}}^*$ *as follows*

$$Adm'(\varphi,\check{\eta},\check{\eta}') = \begin{cases} \check{\eta}.\check{\eta}' & \text{if } \check{\eta}.\check{\eta}' \models \varphi \\ \check{\eta}.\check{\eta}''.(\overline{\alpha},\chi,\lambda) & \text{if } \check{\eta}.\check{\eta}'' \models \varphi \wedge \check{\eta}.\check{\eta}''.(\alpha,\chi,\lambda) \not\models \varphi \text{ where } \check{\eta}' = \check{\eta}''.(\alpha,\chi,\lambda).\check{\eta}''' \end{cases}$$

The extended CFA changes only in the clauses for resource scopes $[\mathbf{Joint2}^{\mathbf{3}}_{\alpha}]$ and $[\mathbf{Req}^{\mathbf{3}}_{\alpha}]$, described in Table 5, because now the parts of the traces appended to the starting one come from the set of all the possible shuffles of the traces of the sequential sub-processes in F. The rule $[\mathbf{Joint2}^{\mathbf{3}}_{\alpha}]$ checks whether $\Gamma(r)$ includes the right traces of usage of the resource r, due to all the sequential sub-processes in F. Formally the clauses check whether $(\varphi, Adm(\varphi,\check{\eta},\check{\eta}')) \in \Gamma(r)$ for each $\check{\eta}'$ in the shuffles returned by $SH(r,F,\chi)$. Similarly, the rule $[\mathbf{Req2}_{\alpha}]$, includes the validation of the enclosed process F and, for each trace in $\Gamma(r)$ of previous accesses on r, checks its composition with the trace $\check{\eta}'$ extracted from $SH(r,F,\chi)$ against the policy φ, by verifying if the compliant part of the resulting trace is included in $\Gamma(r)$, together with the first violating action, if any. Also here, the traces $\check{\eta}$ in $\Gamma(r)$ that we compose must not include χ (in symbols $\chi \notin \check{\eta}$). The other CFA clauses are similar to the ones in Tables 3 and 4. The only difference consists in the use of extended traces $\check{\eta}$ in place of the standard traces η.

Also for this extended analysis, there is a subject reduction theorem that states the correctness of the analysis, as well as an existence result as that in Theorem 4.8. Furthermore, we have a similar version of Theorem 4.17 to state that the absence of faulty traces in the analysis results is sufficient for the checked process to comply with its policy. For the sake of brevity, we only state the correctness result.

Theorem 4.23 ***(Subject reduction)*** *If* $P \xrightarrow{\mu} P'$, *and* $(\rho,\kappa,\Gamma) \models P$, *then* $(\rho,\kappa,\Gamma) \models P'$.

4.3 CFA: Step 3

In the previous sub-sections, to keep the analyses simple, we restricted our attention only to processes that have finite behaviour. The analysis can be extended and enriched to deal with possibly infinite behaviour, as well. For the sake of simplicity, we focus on the behaviour of finite state processes, obtained by including suitable forms of *unbounded iterators over resources* in our calculus.

Our first step consists in extending the syntax of sequential processes, as presented in Figure 8. We reintroduce the bang replication construct $!req(r)\{R\}^{\chi_!}$ intended only for the repeated use and request of the same resource r, provided it is available. We assume that inside the boundary there are just simple sequential processes R, composed by only sequences of prefixes. To deal with the new construct, called *unrestricted resource request point*, we embed the structural congruence rule of replication inside the rule for resource acquisition, as follows.

$$
\begin{array}{lcll}
P,P' & ::= & \textit{processes} \in \textit{Proc} & \\
\ldots & & & \\
 & | & (r,\varphi,\hat{\eta})\{Q\}^{\chi} & \text{resource joint point} \\
 & | & req(z)\{Q\}^{\chi} & \text{resource request point} \\
 & | & !req(z)\{R\}^{\chi_!} & \textit{unrestricted} \text{ resource request point} \\
 & | & (r,\varphi,\hat{\eta})\{R\}^{\chi_!} & \text{single-iteration resource joint point} \\
 & | & (r,\varphi,\hat{\eta})\{\mathbf{0}\}^{\flat}||R & \\
 & & & \\
Q,Q' & ::= & \textit{sequential processes} \in \textit{SeqProc} & \\
 & | & \mathbf{0} & \\
 & | & \pi.Q & \\
 & | & [\hat{\pi}]^*.R & \text{regexp prefix action} \\
 & | & (r,\varphi,\hat{\eta})\{Q\}^{\chi} & \\
 & | & (r,\varphi,\hat{\eta})\{\mathbf{0}\}^{\flat}||Q & \\
 & | & req(z)\{Q\}^{\chi} & \textit{restricted} \text{ resource request point} \\
 & & & \\
R,R' & ::= & \textit{simple sequential processes} \in \textit{SeqProc} & \\
 & | & \mathbf{0} & \\
 & | & \pi.R & \\
 & & & \\
\hat{\pi},\hat{\pi}' & ::= & \textit{action sequences} & \\
 & | & \pi & \text{standard action} \\
 & | & \pi.\hat{\pi} & \text{regular expression sequence of actions}
\end{array}
$$

Figure 8: Labelled syntax.

$$\text{(Repeated Acquire)}\quad !req(r)\{R\}^{\chi_!} \parallel (r,\varphi,\hat{\eta})\{\mathbf{0}\}^{\flat} \xrightarrow{\tau} !req(r)\{R\}^{\chi_!} \parallel (r,\varphi,\hat{\eta})\{R\}^{\chi_!}$$

The primitive $!req(r)\{R\}^{\chi_!}$ enables sequential process R to access the resource r an unlimited amount of times, by decoupling the request $!req(r)\{R\}^{\chi_!}$ from the actual usage $(r,\varphi,\eta)\{R\}^{\chi_!}$, where the label $\chi_!$ recalls that this usage derives from the unfolding of a replication. Note that repeated labels are generated by structural congruence in the replication case.

Example 4.24 *To better understand this new construct, consider a process $!req(r)\{R\}^{\chi_!}$, where $R = \alpha(r).\beta(r).rel(r)$, that repeatedly requires an available resource r:*

$$!req(r)\{\alpha(r).\beta(r).rel(r)\}^{\chi_!}||(r,\varphi,\hat{\eta})\{\mathbf{0}\}^{\flat}$$

Since the contribution of R to the usage of resource r is in the form $[(\alpha,\chi_!).(\beta,\chi_!).(rel,\chi_!)]$, the whole intended usage of the resource r is given by an unlimited number of repetitions of $[(\alpha,\chi_!).(\beta,\chi_!).(rel,\chi_!)]$.

To concisely represent these kinds of traces, we can resort to the operators used for regular languages. As a consequence, the intended usage of the resource can be represented as the repeated trace $[(\alpha,\chi_!).(\beta,\chi_!).(rel\chi_!)]^*$, where the Kleene star $*$ reflects the potentially infinite iteration of that portion of trace. This trace has to be checked against the policy φ. To perform this check, we can resort to standard techniques of model checking, like the ones based on automata (see e.g. [6]).

As a further extension, we introduce the possibility of describing unrestricted repeated sequences of actions, in the form of regular expressions $[\hat{\pi}]^*$, to be used inside a resource boundary, whose semantic treatment, reminiscent of some suitable form of the sequential bang, can be provided by the following structural congruence rule

$$[\hat{\pi}]^*.R \equiv \hat{\pi}.[\hat{\pi}]^*.R + R$$

where, for the sake of simplicity, we assume that after $[\hat{\pi}]^*$ in $[\hat{\pi}]^*.R$ there are no further repeated sequences of actions. As an example, let us consider a process that requires a resource r that repeatedly performs some access operations.

$$req(r)\{\alpha(r).[\beta(r).\gamma(r)]^*.rel(r)\}^{\chi} || (r, \varphi, \tilde{\eta})\{\mathbf{0}\}^{\vartheta}$$

The trace that represents this intended usage of r should be in the form

$$(\alpha, \chi).[(\beta, \chi).(\gamma, \chi)]^*.(rel, \chi)$$

To treat these sequences, we just need the following new case in the function H'', natural extension of H to the extended traces $\hat{\eta}$ (all the other cases are similar to the ones in the definition H).

$$H''(r, [\hat{\pi}]^*, \chi_!) = [H(r, \hat{\pi}, \chi_!)]^*$$

To deal with the new constructs, also traces and their handling have to be extended accordingly. First of all, we need labels in the form $\chi_!$, to distinguish unrestricted iteration usage from standard resource usage. The set of traces is therefore $\hat{\mathscr{A}}^*$, where $\hat{\mathscr{A}} = (\mathscr{A} \cup \{rel\} \times (\mathscr{L} \cup \mathscr{L}_!))$, $\mathscr{L}_!$ is ranged over by $\chi_!$ and $(\mathscr{L} \cup \mathscr{L}_!)$ is ranged over by χ_ω.

To capture replicated behaviour, such as the one presented in the previous example, we need to have standard regular expressions $\hat{\eta}$ over the alphabet $\hat{\mathscr{A}}$. Again these labels can be easily forgotten and removed to obtain the original semantics.

Similarly, we accordingly extend the function Adm to treat the new traces, by introducing the function Adm''.

Definition 4.25 *We inductively define the function* $Adm'' : \hat{\mathscr{A}}^* \times \hat{\mathscr{A}}^* \to \hat{\mathscr{A}}^*$ *as follows*

$$Adm''(\varphi, \hat{\eta}, \hat{\eta}') = \begin{cases} \hat{\eta}.\hat{\eta}' & \text{if } \hat{\eta}.\hat{\eta}' \models \varphi \\ \hat{\eta}.\hat{\eta}''.(\overline{\alpha}, \chi_\omega) & \text{if } \hat{\eta}.\hat{\eta}'' \models \varphi \wedge \hat{\eta}.\hat{\eta}''.(\alpha, \chi_\omega) \not\models \varphi \text{ where } \hat{\eta}' = \hat{\eta}''.(\alpha, \chi_\omega).\hat{\eta}''' \end{cases}$$

To complete the formal development of the new analysis, we have to show the way the contributions of every iterative context can be combined among them and with the contributions of non-iterative contexts. Since these kinds of combinations are non trivial, we resort to some illustrative examples, presented in an incremental fashion.

Example 4.26 *Consider two processes that repeatedly require the resource r in parallel with the resource r.*

$$!(req(r)\{\alpha(r).\beta(r).rel(r)\}^{\chi_!}) || !(req(r)\{\gamma(r).\alpha(r).rel(r)\}^{\chi'_!}) || (r, \varphi, \hat{\eta})\{\mathbf{0}\}^{\vartheta}$$

- *The contribution of the first process to the usage of resource r is in the form* $[(\alpha, \chi_!).(\beta, \chi_!).(rel, \chi_!)]$ *at each iteration, while the contribution of the second one is* $[(\gamma, \chi'_!).(\alpha, \chi'_!)).(rel, \chi'_!)]$.
- *At run time, the two processes alternate their actions on the resource. Consequently, the two contributions can be interleaved or shuffled in every possible way.*

$[\mathbf{Joint3}_\alpha]$	$(\rho,\kappa,\Gamma,\Delta_\omega) \models (r,\varphi,\hat{\eta})\{Q\}^\chi$	iff $(\rho,\kappa,\Gamma,\Delta_\omega) \models Q \wedge [H(r,Q,\chi)] \in \Delta_\omega(r) \wedge$ $\forall \hat{\eta}' \in Mix(\Delta_\omega(r))(\varphi, Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$
$[\mathbf{Joint4}_\alpha]$	$(\rho,\kappa,\Gamma,\Delta_\omega) \models (r,\varphi,\hat{\eta})\{R\}^{\chi_!}$	iff $(\rho,\kappa,\Gamma,\Delta_\omega) \models R \wedge [H(r,R,\chi_!)] \in \Delta_\omega(r) \wedge$ $\forall \hat{\eta}' \in Mix(\Delta_\omega(r))(\varphi, Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$
$[\mathbf{Req}_\alpha]$	$(\rho,\kappa,\Gamma,\Delta_\omega) \models req(r)\{Q\}^\chi$	iff $(\rho,\kappa,\Gamma,\Delta_\omega) \models Q \wedge [H(r,Q,\chi)] \in \Delta_\omega(r) \wedge$ $\forall (\varphi,\hat{\eta}) \in \Gamma(r).\chi \not\in \hat{\eta}.\forall \hat{\eta}' \in Mix(\Delta_\omega(r))$ $(\varphi, Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$
$[!\mathbf{Req}_\alpha]$	$(\rho,\kappa,\Gamma,\Delta_\omega) \models !req(r)\{R\}^{\chi_!}$	iff $(\rho,\kappa,\Gamma,\Delta_\omega) \models R \wedge [H(r,R,\chi_!)] \in \Delta_\omega(r) \wedge$ $\forall (\varphi,\hat{\eta}) \in \Gamma(r).\forall \hat{\eta}' \in Mix(\Delta_\omega(r))$ $(\varphi, Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$

Table 6: CFA rules for unbounded iterators: the most interesting cases

- *Shuffle can be rendered with the operator $+$ that represent the standard union of regular languages. The required traces are then given by the regular expression $\hat{\eta}.[(\alpha,\chi_!).(\beta,\chi_!).(rel,\chi_!) + (\gamma,\chi_!).(\alpha,\chi_!).(rel,\chi_!)]^*$. Intuitively, each time the resource is available, one of the two requests in the two iterative contexts can obtain the resource, use and release it.*

Accordingly, the extension to the case of requests inside many iterative contexts leads to the summation of several contributes.

As before, we can also have non-iterative processes requiring a resource. This implies yet another step in the combination of contributions, as shown in the following example.

Example 4.27 *Consider now two processes that repeatedly require the resource r, and one non repetitive process that require r just once, in parallel with the resource r.*

$$!(req(r)\{\alpha(r).\beta(r).rel(r)\}^{\chi_!})||!(req(r)\{\gamma(r).\alpha(r).rel(r)\}^{\chi'_!})||$$
$$req(r)\{\alpha(r).\alpha(r).rel(r).X\}^{\chi''}||(r,\varphi,\hat{\eta})\{\mathbf{0}\}^\flat$$

- *Again, the contribution of the first process to the usage of resource r is $[(\alpha,\chi_!).(\beta,\chi_!).(rel,\chi_!)]$ at each replication, the contribution of the second one is in the form $[(\gamma,\chi'_!).(\alpha,\chi'_!).(rel,,\chi'_!)]$, while the contribution of the third one is $[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')]$.*
- *The required traces are like the ones in the previous example, except that they contain* just one *occurrence of $[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')]$, i.e.*

$$\hat{\eta}.\hat{\eta}_!.[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')].\hat{\eta}_!$$

with $\hat{\eta}_! = [(\alpha,\chi_!).(\beta,\chi_!).(rel,\chi_!) + (\gamma,\chi_!).(\alpha,\chi_!).(rel,\chi_!)]^*$.

Even more complicated, the case in which there are more finite contributions to place just once, as in:

$$!(req(r)\{\alpha(r).\beta(r).rel(r)\}^{\chi_!})||!(req(r)\{\gamma(r).\alpha(r).rel(r)\}^{\chi'_!})||$$
$$req(r)\{\alpha(r).\alpha(r).rel(r).X\}^{\chi''}||req(r)\{\beta(r).\alpha(r).rel(r).X\}^{\chi'''}(r,\varphi,\hat{\eta})\{\mathbf{0}\}^\flat$$

where we would obtain

$$\hat{\eta}.\hat{\eta}_!.[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')].\hat{\eta}_!.[(\beta,\chi''').(\alpha,\chi''').(rel,\chi''')].\hat{\eta}_!+$$
$$\hat{\eta}.\hat{\eta}_!.[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')]\hat{\eta}_!.[(\alpha,\chi'').(\alpha,\chi'').(rel,\chi'')].\hat{\eta}_!$$

Therefore, we can have all the traces obtained by interleaving the contributions of non-iterative contexts (in every possible order) with Kleene closure of the summation of all the contributions of every iterative context. Note that, again, we remain inside regular languages.

To formally deal with these kinds of traces, we introduce a further analysis component Δ_ω that, given a resource r, collects the individual intended contributions on r of all replicated and non-replicated contexts, respectively. It is convenient, in Δ_ω, to distinguish between the parts:

- $\Delta_!(r)$, which collects the repetitive contributions, i.e. the ones in the form

$$\hat{\eta}_! \in (\mathscr{A} \cup \{rel\} \times \mathscr{L}_!)^*$$

- and $\Delta(r)$, which collects the non-repetitive contributions, i.e. the ones in the form

$$\hat{\eta} \in (\mathscr{A} \cup \{rel\} \times \mathscr{L})^*$$

The combinations of all the contributions on r can be obtained starting from Δ_ω in the following way. Let *Mix* be the function able to compute all the needed combinations, by suitably interleaving the contributions in Δ_ω. Finite contributions can appear in every point, but just once. Its definition follows.

$$Mix(\Delta_\omega(r)) = \{\hat{\eta}' = [\sum_{\hat{\eta}_!^i \in \Delta_!} \hat{\eta}_!^i]^* \, \hat{\eta}^1 [\sum_{\hat{\eta}_!^i \in \Delta_!} \hat{\eta}_!^i]^* ... \hat{\eta}^k [\sum_{\hat{\eta}_!^i \in \Delta_!} \hat{\eta}_!^i]^* \mid \text{ for all permutations } \hat{\eta}^1 ... \hat{\eta}^k \text{ with } \hat{\eta}^i \in \Delta(r)\}$$

where a permutation is intended as a one-to-one correspondence of a set with itself, which carries each element of the set into the one that occupies the same position after the permutation is applied.

Also for this extension, the CFA changes only in the clauses for resource scopes described in Table 6. More in detail, the rules check whether $\Gamma(r)$ includes the right traces of usage of the resource r, due to all the contributions $\hat{\eta}'$ in $Mix(\Delta_\omega(r))$, checked against φ. Formally they check whether $(\varphi, Adm(\varphi, \hat{\eta}, \hat{\eta}')) \in \Gamma(r)$ for each $\hat{\eta}'$ in $Mix(\Delta_\omega(r))$. The remaining CFA clauses are similar to the ones in Tables 3 and 4. The only difference consists in the use of extended traces $\hat{\eta}$ in place of standard traces η.

Also in this case, there is a subject reduction theorem that states the correctness of the analysis, as well as an existence result as that in Theorem 4.8. Furthermore, we have a similar version of Theorem 4.17 to state that the absence of faulty traces in the analysis results is sufficient for the checked process to comply with its policy. For the sake of brevity, we only state the correctness result.

Theorem 4.28 ***(Subject reduction)*** *If $P \xrightarrow{\mu} P'$, and $(\rho, \kappa, \Gamma, \Delta_\omega) \models P$, then $(\rho, \kappa, \Gamma, \Delta_\omega) \models P'$.*

5 Concluding remarks and discussions

We have presented a model for the management of resources in distributed applications. Our model takes the form of an extension of π-calculus with suitable primitives to manage resources. The novel feature of our proposal relies on the adoption of the publish-subscribe paradigm to handle resource acquisition. We argued that our calculus supports the design of modular applications that are loosely coupled with

respect to resources. Furthermore, the definition of coordination policies on processes that act on the same resources is naturally supported. Finally, we developed a Control Flow Analysis that computes a static approximation of resource usages of applications.

Understanding the foundations of the distributed management of resources is a challenging research line that may extend state-of-the-art advances of programming language constructs, algorithms and reasoning techniques for resource-aware programming of distributed applications. In the last few years, the problem of providing the mathematical basis for the mechanisms that support resource usage has received a major attention. A number of models has been proposed (see e.g. [11, 25, 51, 31, 46], to cite only a few) for resource management. These are characterised by different motivations, design choices and technical solutions.

The G-Local π-calculus design. We started the design of our calculus by adopting the so-called history-based approach, which has been studied in [7, 36, 64, 6]. The history-dependent framework overcomes the weaknesses of stack-based approaches [37] that record only a fragment of the trace instead of the whole trace (called *history*). In [11], the authors propose an extension of the λ-calculus to statically verify resource usage. The work combines *local checks* of program points, where critical resources can be accessed, with *global policies*, which enforce a global invariant to hold at any program point. Our work is inspired and extends in several places these works, by borrowing ideas from service oriented computing (SOC) [61]. A key theme of SOC is the design of a general theory of services, often based on mobile calculi, for formalising service-oriented applications [70, 23] and for developing suitable verification techniques [27, 42, 52]. Although our model is based on mobile calculi, we specifically focus on the orthogonal issue of the correctness of resource usage.

The novel feature of our proposal relies on the *interaction patterns* between resources and computational processes. These interactions are established through a sort of *publish-subscribe* paradigm. Notice that these features have been exploited in service oriented computing. The emphasis on design of service orientation so far has been on the description of interactions and on the development of related concepts, like context-awareness or service sessions. Invocation of "services" establishes a session, whose interactions follow a certain protocol through the standard mechanism of communication, provided by mobile calculi. Under this regard, services could be considered as resources at a high level point of view. Resources in our approach are represented by structures with states and usage polices. Usage policies related to individual resources provide indeed a flexible way to define fine-grain access control on resources. This emphasises that the focus of our approach is on the *correct usage of resources* rather than on the *discipline of interactions* like in SOC.

Resources in the G-Local π-calculus have scopes that can be thought as resource *administrative domains*, similar to the scope of locations in Mobile Ambients (MA) [29]. Closer to ours is the work in [35], where an ambient is considered as a unit for monitoring and coordinating activities. More precisely, each ambient is equipped with a guardian, which monitors the activities of sub-components (i.e. processes and sub-ambients). Unlike MA, the scopes of resources are more restricted since the scopes cannot be opened. Placing restriction on the scope of resources is a design decision that makes the control of resource management easier. While the scopes of locations in MA are managed by explicit actions of processes, configurations of resources in our approach are not under the control of processes and resources have no control over other resources. This assumption is justified in terms of loosely coupling design, which is typical of modular distributed applications.

Models of resources. The simplest model of resources in process calculi is given by the notion of *names*. In name-passing process calculi, names can be communicated and exchanged, while in ambient-like calculi, names assume the role of locations. Thus, it is natural to treat names as resources in many process-calculus approaches to resource management [71, 50, 67, 68]. In [50], resource usage is simply bound to the number of communications in channels. The work in [68] focuses on the ownership and publicity of names. The proposals in [71, 67] are centred on allocation/deallocation of resources. In the first, reconfiguration steps are internalised inside processes via the operations for allocating and de-allocating channels. A similar idea is found in [67], where the authors introduce closer explicit transition rules for eliminating *dead* processes through different *garbage-collectable* relations on processes. The drawback of these works is that properties of resources are quite limited. In [45], more advanced properties of names are introduced and studied with a logic for express the relations between processes and resources. In our approach resources are structured to allow reasoning on the actual states of resources.

The π-calculus dialect of [51] provides a framework for checking resource usage in distributed systems. The treatment of resources in this approach is closer to ours in terms of properties of resources. Indeed, private names are extended to resources, i.e. names with a set of traces to define control over resources. Also resource request and resource release are simulated through communication of private names. This provides a shared semantics of resources, i.e. several processes can have a concurrent access to resources (by communicating private names). Resources in our approach may be considered as names with additional structures, however they cannot be *private*. Unlike private names with the dynamic boundaries through the scope extrusion, they have a *fixed* boundary. Also, our semantics of resources differs from the one in [51]: when a process obtains a resource, it has an exclusive access to it.

The works in [31, 25] consider an explicit notion of resources, different from ours. In particular, in [31], the authors propose a process calculus with an explicit representation of resources, in which the evolution of processes and resources follows a *synchronous* SCCS style. More precisely, resources form a monoid, i.e. a set of elements with a binary operator. Thus, a resource can be non-deterministically split into *smaller* pieces (by the binary operator defined in the monoid) to be distributed among processes. In this way, the notion of resources is closely related to the sharing interpretation of the **BI**-family logics (see [58] for details). Also, a modal logic, based on the **BI**-family logics, is developed to specify resource properties. The drawback of this approach is the co-evolution of processes and resources. It requires a pre-defined model of resources, which is sometimes difficult to define. In our approach, resources are independent stateful entities, thus subject to be requested, and are equipped with their own global interaction usage policy, defined as a set of traces. Therefore, LTL formulas or equivalent formalisms can be used to specify temporal properties of resources. It is worthwhile noting that this approach promotes a tightly coupled paradigm where resources and processes run with high degrees of dependency, while our approach fosters a loosely-coupled view of the resource-consumer interactions.

The work presented in [25] mainly focuses on specifying SLA by describing resources as suitable *soft* constraints. In this proposal, c-semirings [14] act as models of resources. The shared store of constraints on resources represents SLA contracts established through allocation and deallocation of resources. C-semirings allow for expressing soft constraints, i.e. constraints that give informative values instead of just true or false. Similar to ours, available resources are obtained through suitable requests. It is easy to see that we might exploit constraints to express global resource usage as well.

Static Approach to resource usage. We equip our calculus with a static machinery for managing resource usage, based on Control Flow Analysis. The CFA computes a safe over-approximation of communication-based and finite resource-based behaviour. First, resource-based behaviour is described

with their possible traces and configurations (i.e. the resource contexts). Our CFA extends the one for the π-calculus in [19], with some insights coming from [15] and from [16]. The extension is needed to handle resources and to have the suitable information in the analysis results, to statically check some reachability properties of resources. The novelty of our CFA consists in applying flow logic [56] to manage the asynchrony of the publish-subscribe paradigm. A first contribution in this direction can be found in [22].

Other approaches have addressed resource usage issues by exploiting types. A resource-aware type system for the π-calculus has been introduced in [50]. Resources are channels and *linear* types are employed to derive static information about process interactions over channels: e.g. two processes communicating over a linear channel cannot interfere with other communicating processes. The linear type systems for the π_l-calculus in [72, 44] control and limit channel reuse in order to statically avoid the occurrence of *deadlocks*. Other static techniques, based on type systems, have been proposed for deadlock detection in mobile calculi, e.g. [41, 60], also mentioned in Subsection 4.1. In [41], the authors present a new deadlock detection technique for the value-passing CCS (and for the π-calculus) that enables the analysis of networks with arbitrary numbers of nodes. Padovani, in [60], studies two refinements of the linear π-calculus that ensure both deadlock freedom and lock freedom. There are also tools for deadlock detection, such as TyPiCal [49], a quite powerful analyser for π-calculus, developed by Kobayashi. Teller [67] introduces a variant of the π-calculus where specialised processes, called *finalisers*, run to de-allocate channel resources and where a type system provides bounds on resources usage. In [38] π-calculus channels are viewed as resources that must explicitly be allocated before their use and can be deallocated when no longer needed. A suitable type system supports the development of proof techniques for comparing resource usage. A behavioural type system for statically checking spatial properties, using a fragment of spatial logics [28], has been developed in [2]. Our proposal differs from these static approaches in the notion of resource. We assume that the resource-consumer relationship is governed by suitable usage policies. Moreover, our focus is on the global coordination that manages the way resources, aggregated from multiple sources, are used. A more detailed discussion on the relationships with type systems can be found in [33], together with a type system for our calculus, based on behavioural types. There, types abstract two kinds of behavioural information: communication-based and resource-based. Essentially, while CFA allows us to address reachability and safety property, behavioural types allow us instead to address different kinds of properties, such as linear temporal properties, including liveness properties.

Programming abstractions for resource awareness. In programming languages, abstraction mechanisms, like objects, classes or abstract data types, aim at helping developers to focus on the software design. Indeed, focussing on business logic of software systems allows for freeing developers from implementation details. For example, the try-with-resources statement has recently been introduced in JDK7 [59] to support developers in ensuring proper termination over resources. The separation of concerns between business logic and operation logic is indeed the main focus of our work. The term *operation logic*, coined in [47], refers to resource management in distributed settings, where applications are capable of accessing a variety of resources. Our work is a first step on the way of providing a foundational basis for resource-aware programming abstractions. We illustrate, in Section 2, an example of resource-aware programming abstractions, suggested by our approach.

Closest to our idea of resource-aware programming are the proposals presented in [3, 66]. The first one introduces a *typesafe-oriented programming language* by extending the object paradigm with object states. Objects are modelled in terms of *changing states*, rather than *classes*. The development of the

resource-aware programming *Plaid*, introduced in [66], follows the idea outlined above. *Typesafe programming*, originally introduced in [65], expresses that each state of an object has its own representation and methods (only these methods are available for the object in this state) may lead the object into a new state. The dynamics of states allows developers to control program behaviour. Under this regard, we could consider objects as linked to a security policy, thus restricting the transitions of objects from one state to another. In a broader view of resources, objects can be considered as resources, and therefore can be assimilated to our notion of resources. The essential difference between the typesafe approach and ours is that for us resources are independent entities and that we rely on loose coupling design to separate fragments of code that use resources, from resources themselves, rather than scattering fixed operations through a set of states of objects. More precisely, access control over resources in the programming language in [66] is more fine-grained than ours: the adopted access control includes shared, immutable and exclusive accesses. Our approach, instead, supports only exclusive resource accesses.

Future work. Several extensions are possible. Here, we outline some of them devoted to future work.

- **Model of resources.** Our model relaxes model *privacy* of resources, i.e. all resources are public entities, hence every process knowing resource names may potentially access them. Instead of taking private names as resources, we could extend resources with privacy, i.e. considering r as a private name, as in the following example:

  $$P ::= (\nu r)((r,\varphi,\eta)\{\mathbf{0}\} \parallel (r,\varphi,\eta)\{\mathbf{0}\} \parallel Q)$$

  where two identical resources are available under a private name r for being used only by the processes that know r (Q in this case). The nature of private names with notion of group is not new, however, the novel treatment of resource privacy could give a closer view of resource usage, which can be potentially further divided in:

  i) an internal view, which only focuses on the states of resources;
  ii) an external view, which focuses on external information by processes that request resources.

- **Dynamic reconfiguration.** Resource entities could be dynamically reconfigured via resource movements. Besides the structural rules, we could have included the following transition rules $(Appear)$ and $(Disappear)$:

  $$\text{(Appear)} \quad P \xrightarrow{\tau} P \parallel (r,\varphi,\eta)\{\mathbf{0}\} \quad \text{(Disappear)} \quad (r,\varphi,\eta)\{P\} \xrightarrow{\tau} \mathbf{0}$$

  which describe the abstract behaviour of the resource manager, by performing asynchronous resource reconfigurations. Asynchronous dynamic reconfiguration would offer a higher degree of loose coupling among processes and resources. Furthermore, it would be a feature not present in other proposals, such as the ones in [51, 50, 67]. A preliminary discussion on this extension can be found in [33].

- **Elasticity of resources vs replication.** Note that $(Appear)$ and $(Disappear)$ can offer a way to address the so-called *elasticity* of cloud resources. In the same line, we could propose the following structural rules for replication that work, provided that resource instances exactly match their syntax. This is only true for a resource, whose instances have the same state. In general, it could not be used for resource instances with different states.

  $$\begin{array}{ll} (\text{Rep}_1) & !(r,\phi,\eta)\{\mathbf{0}\} \xrightarrow{\tau} (r,\phi,\eta)\{\mathbf{0}\} \parallel (r,\varphi,\eta)\{\mathbf{0}\} \\ (\text{Rep}_2) & (r,\phi,\eta)\{\mathbf{0}\} \parallel (r,\varphi,\eta)\{\mathbf{0}\} \xrightarrow{\tau} !(r,\phi,\eta)\{\mathbf{0}\} \end{array}$$

- **Polyadic requests.** Since obtaining a bunch of resources is often required in a cloud-computing scenario, having polyadic requests would be desirable. Unlike polyadic input/outputs, where synchronisation involves only two parties, polyadic resource requests make transition rules more complicated as they involve multiple parties. This requires having multi-party interaction primitives in our calculus, as the ones in [17].
- **Resource movement.** The present structural rule for resource management is unconditional. Nevertheless, we could instead specify some conditions to restrict the movement of resources as in [20], by using a structural rule like the following:

$$(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel (r_1,\varphi_1,\eta_1)\{P\} \equiv (r_1,\varphi_1,\eta_1)\{(r_2,\varphi_2,\eta_2)\{\mathbf{0}\} \parallel P\} \text{ if } cond(\eta_1,\eta_2)$$

  where $cond(\eta_1,\eta_2)$ is a condition function on the current states of the two resources.
- **Semantic-based resource requests** The present formulation of the calculus ensures that any operation on a granted resource always respects the policy required over the resource. However, one would like to acquire a specific resource under certain additional parameters that specify further requirements on the resource. To describe this kind of semantic matching of the resource requests, we could extend our calculus, by allowing resource requests in the form $req(R,\psi')$, where R represents a set of resources that provide the same service, under different conditions, while $\psi' \in \Psi$ specifies the required conditions or parameters. The resource joint point can be modified accordingly as $(r,\psi,\varphi,\tilde{\eta})\{P\}$, where $\psi \in \Psi$ represents the set of chosen parameters. We would also need a new rule for resource acquisition, where a suitable function $comp$ is used to assert the compatibility of parameters required for the semantic matching:

$$req(R,\psi')\{P\} \parallel (r,\psi,\varphi,\tilde{\eta})\{\mathbf{0}\} \xrightarrow{\tau} (r,\psi,\varphi,\tilde{\eta})\{P\{r/R\}\} \text{ provided that } r \in R \wedge comp(\psi,\psi')$$

  The operations on resources, in the form $\alpha(R)$ and $release(R)$, could become $\alpha(r)$ and $release(r)$, once established the binding on r. Clearly, we would modify the CFA accordingly.

  As an illustrative scenario, suppose that a European user needs computational power for business reasons, at the lowest price. Also suppose that there are resources available over the Internet, both in Europe and in the USA, with different fees depending on the chosen time slots. Since, usually costs are lower during the night, one could have the following specification

$$\begin{aligned} Users &::= U_{Day} \parallel U_{Night} \\ U_{Day} &::= req(C,USA)\{P_d\} \\ U_{Night} &::= req(C,EU)\{P_n\} \end{aligned}$$

  where synchronisation with provided services could be in the form $(c,location,\varphi,\tilde{\eta})\{\mathbf{0}\}$, where $c \in C$ and the function $comp$ checks whether the locations match.

References

[1] G. Abowd & E.D. Mynatt (2000): *Charting past, present, and future research in ubiquitous computing*. *ACM Transactions on Computer-Human Interaction* 7, pp. 29–58.

[2] L. Acciai & M. Boreale (2010): *Spatial and behavioral types in the pi-calculus*. *Inf. Comput.* 208(10), pp. 1118–1153.

[3] J. Aldrich, J. Sunshine, D. Saini & Z. Sparks (2009): *Typestate-oriented programming*. In: *Proc. of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009 (Companion)*, ACM, pp. 1015–1022.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica & M. Zaharia (2009): *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[5] L. Atzori, A. Iera & G. Morabito (2010): *The internet of things: a survey*. *Computer Networks* 54(15), pp. 2787–2805.

[6] C. Baier & J.P. Katoen (2008): *Principles of model checking*. MIT Press.

[7] A. Banerjee & D.A. Naumann (2005): *History-Based Access Control and Secure Information Flow*. In: *Proc. of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS'04)*, *Lecture Notes in Computer Science* 3362, Springer, pp. 27–48.

[8] M. Bartoletti (2009): *Usage Automata*. In: *Proc. of Foundations and Applications of Security Analysis, Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, *Lecture Notes in Computer Science* 5511, Springer, pp. 52–69.

[9] M. Bartoletti, P. Degano & G. Ferrari (2005): *History-Based Access Control with Local Policies*. In: *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS'05)*, *Lecture Notes in Computer Science* 3441, Springer, pp. 316–332.

[10] M. Bartoletti, P. Degano, G. Ferrari & R. Zunino (2015): *Model checking usage policies*. *Mathematical Structures in Computer Science* 25(3), pp. 710–763.

[11] M. Bartoletti, P. Degano, G.L. Ferrari & R. Zunino (2009): *Local Policies for Resource Usage Analysis*. *ACM Transactions on Programming Languages and Systems* 31(6).

[12] M. Bartoletti & R. Zunino (2008): *LocUsT: a tool for checking usage policies*. Technical Report TR-08-07, Dip. Informatica, Univ. Pisa.

[13] M. Bartoletti & R. Zunino (2010): *A Calculus of Contracting Processes*. In: *Proc. of Logic in Computer Science (LICS'10)*, IEEE Computer Society, pp. 332–341.

[14] S. Bistarelli, U. Montanari & F. Rossi (1997): *Semiring-based Constraint Satisfaction and Optimization*. *Journal of the ACM* 44, pp. 201–236.

[15] C. Bodei (2009): *A Control Flow Analysis for Beta-binders with and without static compartments*. *Theoretical Computer Science* 410(33-34), pp. 3110–3127.

[16] C. Bodei, L. Brodo & R. Bruni (2009): *Static Detection of Logic Flaws in Service-Oriented Applications*. In: *Proc. of Foundations and Applications of Security Analysis, Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS 2009)*, *Lecture Notes in Computer Science* 5511, Springer, pp. 70–87.

[17] C. Bodei, L. Brodo & R. Bruni (2012): *Open Multiparty Interaction*. In: *Recent Trends in Algebraic Development Techniques, 21st International Workshop, (WADT 2012)*, *Lecture Notes in Computer Science* 7841, Springer, pp. 1–23.

[18] C. Bodei, M. Buchholtz, P. Degano, F. Nielson & H. Riis Nielson (2005): *Static validation of security protocols*. *Journal of Computer Security* 13(3), pp. 347–390.

[19] C. Bodei, P. Degano, F. Nielson & H. Nielson (2001): *Static Analysis for the Pi-Calculus with Applications to Security*. *Information and Computation* 168(1), pp. 68–92.

[20] C. Bodei, V. D. Dinh & G. L. Ferrari (2011): *A G-Local π-calculus*. In: *Proc. of Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'11)*. Available at `http://places11.di.fc.ul.pt/proceedings.pdf/view`.

[21] C. Bodei, V. D. Dinh & G. L. Ferrari (2011): *Predicting global usages of resources endowed with local policies*. In: *Proc. of the Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'11)*, *Electronic Proceedings in Theoretical Computer Science* 58, pp. 49–64.

[22] C. Bodei & G. L. Ferrari (2009): *Choreography Rehearsal*. In: *Proc. of Web Services and Formal Methods, 6th International Workshop, (WS-FM 2009)*, *Lecture Notes in Computer Science* 6194, pp. 29–45.

[23] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Thudichum Vasconcelos & G. Zavattaro (2006): *SCC: A Service Centered Calculus*. In: *Proc. of Web Services and Formal Methods, Third International Workshop (WS-FM'06)*, *Lecture Notes in Computer Science* 4184, Springer, pp. 38–57.

[24] D.F.C. Brewer & M.J. Nash (1989): *The chinese wall security policy*. In: *Proc. of IEEE Symposium on Security and Privacy*, pp. 206–214.

[25] M. G. Buscemi & U. Montanari (2007): *CC-pi: A Constraint-based Language for Specifying Service Level Agreements*. In: *Proc. of European Symposium on Programming (ESOP'07)*, *Lecture Notes in Computer Science* 4421, Springer, pp. 18–32.

[26] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg & I. Brandic (2009): *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. *Future Generation Computer Systems* 25, pp. 599–616.

[27] L. Caires (2007): *Spatial-Behavioral Types, Distributed Services, and Resources*. In: *Proc. of Trustworthy Global Computing, Second Symposium (TGC 2006)*, *Lecture Notes in Computer Science* 4661, Springer, pp. 98–115.

[28] L. Caires & L. Cardelli (2003): *A spatial logic for concurrency (part I)*. *Inf. Comput.* 186(2), pp. 194–235.

[29] L. Cardelli & A. D. Gordon (2000): *Mobile ambients*. *Theoretical Computer Science* 240(1), pp. 177–213.

[30] G. Castagna, N. Gesbert & L. Padovani (2009): *A theory of Contracts for Web services*. *ACM Transactions on Programming Languages and Systems* 31(5).

[31] M. Collinson & D.J. Pym (2010): *Algebra and Logic for Access Control*. *Formal Aspects of Computing* 22(3-4), pp. 483–484.

[32] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart & D. B. Terry (1987): *Epidemic Algorithms for Replicated Database Maintenance*. In: *Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing 1987*, pp. 1–12.

[33] V. D. Dinh (2011): *A Language-based Approach to Distributed Resources*. Ph.D. thesis, PhD in Computer Science, University of Pisa.

[34] U. Erlingsson & F. B. Schneider (1999): *SASI enforcement of security policies: a retrospective*. In: *Proc. of the 1999 Workshop on New Security Paradigms*, pp. 87–95.

[35] G. L. Ferrari, E. Moggi & R. Pugliese (2002): *Guardians for Ambient-based Monitoring*. *Electronic Notes Theoretical Computer Science* 66(3), pp. 52–75.

[36] P. W. L. Fong (2004): *Access Control By Tracking Shallow Execution History*. In: *Proc. of IEEE Symposium on Security and Privacy*, pp. 43–55.

[37] C. Fournet & A. Gordon (2003): *Stack inspection: Theory and variants*. *ACM Transactions on Programming Languages and Systems* 25, pp. 360–399.

[38] A. Francalanza, E. de Vries & M. Hennessy (2012): *Compositional Reasoning for Channel-Based Concurrent Resource Management*. Technical Report CS2012-02, Department of Computer Science, Malta University.

[39] H. Gao, C. Bodei, P. Degano & H. Riis Nielson (2007): *A Formal Analysis for Capturing Replay Attacks in Cryptographic Protocols*. In: *Proc. of Advances in Computer Science (ASIAN'07)*, *Lecture Notes in Computer Science* 4846, Springer, pp. 150–165.

[40] D. Gelernter (1985): *Generative communication in Linda*. *ACM Transactions on Programming Languages and Systems* 7(1), pp. 80–112.

[41] E. Giachino, N. Kobayashi & C. Laneve (2014): *Deadlock Analysis of Unbounded Process Networks*. In: *Proc. of Concurrency Theory (CONCUR 2014)*, *Lecture Notes in Computer Science* 8704, Springer, pp. 63–77.

[42] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi & G. Zavattaro (2006): *SOCK: A Calculus for Service Oriented Computing*. In: *Proc. of Service-Oriented Computing (ICSOC'06)*, *Lecture Notes in Computer Science* 4294, Springer, pp. 327–338.

[43] K. W. Hamlen, J. G. Morrisett & F. B. Schneider (2006): *Computability classes for enforcement mechanisms*. *ACM Trans. on Programming Languages and Systems* 28(1), pp. 175–205.

[44] K. Honda (2004): *From process logic to program logic*. In: *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, ACM, pp. 163–174.

[45] A. Igarashi & N. Kobayashi (2001): *A generic type system for the Pi-calculus*. *SIGPLAN Notices* 36, pp. 128–141.

[46] A. Igarashi & N. Kobayashi (2005): *Resource usage analysis*. *ACM Transactions on Programming Languages and Systems* 27, pp. 264–313.

[47] A.D.Gordon K. Bhargavan (2008): *Getting operations logic right: Types, serviceorientation, and static analysis*. In: *Proc. of the Workshop on "The Rise and Rise of the Declarative Datacentre"*, Microsoft Research.

[48] D. Kempe, A. Dobra & J. Gehrke (2003): *Gossip-Based Computation of Aggregate Information*. In: *Proc. of the 44th Symposium on Foundations of Computer Science (FOCS 2003)*, pp. 482–491.

[49] N. Kobayashi (2007): *TyPiCal*. Available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`.

[50] N. Kobayashi & D.N. Turner B.C. Pierce (1999): *Linearity and the pi-calculus*. *ACM Transactions on Programming Languages and Systems* 21, pp. 914–947.

[51] N. Kobayashi, K. Suenaga & L. Wischik (2006): *Resource Usage Analysis for the Pi-Calculus*. *Logical Methods in Computer Science* 2(3), pp. 1–42.

[52] A. Lapadula, R. Pugliese & F. Tiezzi (2007): *A Calculus for Orchestration of Web Services*. In: *Proc. of European Symposium on Programming (ESOP'07)*, *Lecture Notes in Computer Science* 4421, Springer, pp. 33–47.

[53] J. Ligatti, L. Bauer & D. Walker (2005): *Edit automata: Enforcement mechanisms for run-time security policies*. *International Journal of Information Security* 4, pp. 2–16.

[54] J. Lim, T. Suh & H. Yu (2013): *A Deadlock Detection Algorithm Using Gossip in Cloud Computing Environments*. In: *Ubiquitous Information Technologies and Applications*, *Lecture Notes in Electrical Engineering* 214, Springer Netherlands, pp. 781–789.

[55] J. Lim, T. Suh & H. Yu (2014): *Unstructured deadlock detection technique with scalability and complexity-efficiency in clouds*. *International Journal of Communication Systems* 27(6).

[56] H. Riis Nielson & F. Nielson (2002): *Flow logic: a multi-paradigmatic approach to static analysis*. In: *The essence of computation*, *Lecture Notes in Computer Science* 2566, Springer-Verlag, pp. 223–244.

[57] H. Riis Nielson, F. Nielson & H. Pilegaard (2012): *Flow Logic for Process Calculi*. *ACM Comput. Surv.* 44(1), p. 3.

[58] P. W. O'Hearn & D. J. Pym (1999): *The Logic of Bunched Implications*. *BULLETIN OF SYMBOLIC LOGIC* 5(2), pp. 215–244.

[59] Oracle-Corporation (2011): *The try-with-resources statement*. Available at `https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`.

[60] L. Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*, ACM, pp. 72:1–72:10.

[61] M. P. Papazoglou, P. Traverso, S.m Dustdar & F. Leymann (2007): *Service-Oriented Computing: State of the Art and Research Challenges*. *IEEE Computer* 40(11), pp. 38–45.

[62] D. Sangiorgi & D. Walker (2001): *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[63] F.B. Schneider (2000): *Enforceable security policies*. *ACM Trans. Inf. Syst. Secur.* 3(1), pp. 30–50.

[64] C. Skalka & S. F. Smith (2004): *History Effects and Verification*. In: *Proc. of ASIAN Symposium on Programming Languages and Systems (APLAS'04)*, *Lecture Notes in Computer Science* 3302, Springer, pp. 107–128.

[65] R. E. Strom & S. Yemini (1986): *Typestate: A Programming Language Concept for Enhancing Software Reliability*. *IEEE Transactions on Software Engineering* 12(1), pp. 157–171.

[66] J. Sunshine, K. Naden, S. Stork, J. Aldrich & É. Tanter (2011): *First-class state change in plaid*. In: *Proc. of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, ACM, pp. 713–732.

[67] D. Teller (2004): *Recovering resources in the pi-calculus*. In: *Proc. of Exploring New Frontiers of Theoretical Informatics (IFIP TCS'04)*, Kluwer, pp. 605–618.

[68] A. Turon & M. Wand (2011): *A Resource Analysis of the π-calculus*. *Electronic Notes in Theoretical Computer Science* 276, pp. 313–334.

[69] M. Y. Vardi & P. Wolper (1986): *An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)*. In: *Proc. of Logic in Computer Science (LICS'86)*, pp. 332–344.

[70] H. T. Vieira, L. Caires & J. Costa Seco (2008): *The Conversation Calculus: A Model of Service-Oriented Computation*. In: *Proc. of European Symposium on Programming (ESOP'08)*, *Lecture Notes in Computer Science* 4960, Springer, pp. 269–283.

[71] E. D. Vries, A. Francalanza & M. Hennessy (2009): *Uniqueness Typing for Resource Management in Message-Passing Concurrency*. In: *Proc. of LINEARITY'09*, *Electronic Proceedings in Theoretical Computer Science* 22, pp. 26–37.

[72] N. Yoshida, K. Honda & M. Berger (2007): *Linearity and bisimulation*. *J. Log. Algebr. Program.* 72(2), pp. 207–238.

[73] L. Youseff, M. Butrico & D. Da Silva (2008): *Toward a Unified Ontology of Cloud Computing*. In: *Proc. of Grid Computing Environments Workshop (GCE '08)*, pp. 1–10.

6 Appendix

Proofs of the main results of CFA, step 1

We recall Theorem 4.8 (the original appears on p. 22).

Theorem 4.8 (Existence of estimates) *For all the processes P, the set $\{(\rho,\kappa,\Gamma)|(\rho,\kappa,\Gamma) \models P\}$ is a Moore family.*

PROOF. We proceed by structural induction on P.

$$\mathscr{J} \subseteq \{(\rho,\kappa,\Gamma)|(\rho,\kappa,\Gamma) \models P\}$$

and let J and $(\rho_j,\kappa_j,\Gamma_j)$ such that $\mathscr{J} = \{(\rho_j,\kappa_j,\Gamma_j)|j \in J\}$ Next define

$$(\rho',\kappa',\Gamma') = \sqcap J = \sqcap\{(\rho_j,\kappa_j,\Gamma_j)|j \in J\}$$

and recall that the greatest lower bound is defined pointwise. The proof of the theorem amounts to checking $(\rho',\kappa',\Gamma') \models P$. For this we proceed by cases on P making use of the induction hypothesis. Most cases are straightforward and here we only consider some the more interesting ones.
The case $x(y).P$. Since $\forall j \in J : (\rho_j,\kappa_j,\Gamma_j) \models x(y).P$, we have

$$\forall j \in J : (\rho_j,\kappa_j,\Gamma_j) \models P \wedge \forall a \in \rho_j(x) : \kappa_j(a) \cap \mathscr{N} \subseteq \rho_j(y)$$

Using the induction hypothesis and the fact that $(\rho', \kappa', \Gamma')$ is obtained in a pointwise manner, we then obtain

$$(\rho', \kappa', \Gamma') \models P \wedge \forall a \in \rho'(x) : \kappa'(a) \cap \mathscr{N} \subseteq \rho'(y)$$

thus establishing the desired $(\rho', \kappa', \Gamma') \models x(y).P$.
The case $(r, \varphi, \eta)\{Q\}^{\chi}$. Since $\forall j \in J : (\rho_j, \kappa_j, \Gamma_j) \models (r, \varphi, \eta)\{Q\}^{\chi}$ we have

- $(\rho_j, \kappa_j, \Gamma_j) \models Q$ and
- $(cond\ 1)$, where $(cond\ 1)$ is $(\varphi, Adm(\varphi, \tilde{\eta}, \tilde{\eta}')) \in \Gamma(r)$ where $\tilde{\eta}' = H(r, Q, \chi)$.

Using the induction hypothesis and the fact that $(\rho', \kappa', \Gamma')$ is obtained in a pointwise manner, we then obtain

- $(\rho', \kappa', \Gamma') \models Q$ and
- $(cond'\ 1)$, where $(cond'\ 1)$ is $(\varphi, Adm(\varphi, \tilde{\eta}, \tilde{\eta}')) \in \Gamma'(r)$ where $\tilde{\eta}' = H(r, Q, \chi)$.

□

We recall Fact 4.9 (the original appears on p. 23).

Fact 4.9 *Given an estimate* (ρ, κ, Γ) *and* $v \in \rho(x)$*, we have that* $\forall y \in \mathscr{N} : \rho(y(\{v/x\})) \subseteq \rho(y)$

PROOF. If $y \neq x$ then $\rho(y(\{v/x\})) = \rho(y)$. Instead, if $y = x$ then $\rho(y(\{v/x\})) = \rho(v)$. Since $\rho(v) = \{v\}$, $v \in \rho(x)$ and $x = y$, it follows that $v \in \rho(y)$. Therefore $\rho(v) \subseteq \rho(y)$. □

We recall Lemma 4.10 (the original appears on p. 23).

Lemma 4.10 ***(Substitution)*** *If* $(\rho, \kappa, \Gamma) \models P$ *then* $(\rho, \kappa, \Gamma) \models P\{v/x\}$*, provided that* $v \in \rho(x)$.

PROOF. The proof of the thesis proceeds by structural induction on P. Most cases are straightforward, using Fact 4.9. We consider here only one of the most interesting ones.

The Case of $P \equiv a(y).P'$, where we may assume w.l.o.g. that $y \neq v, x$.

- **sub-case** $a \neq x$: $(\rho, \kappa, \Gamma) \models P$ amounts to checking that
  - $(\rho, \kappa, \Gamma) \models P'$ and
  - $\forall b \in \rho(a) : \kappa(b) \cap \mathscr{N} \subseteq \rho(y)$ (1).

  By induction hypothesis and the fact stated above, we have that $(\rho, \kappa, \Gamma) \models P'\{v/x\}$. Since $a \neq x$, then from $\rho(a(\{v/x\})) = \rho(a)$, (1) holds and then $(\rho, \kappa, \Gamma) \models P\{v/x\}$.
- **sub-case** $a = x$: again $(\rho, \kappa, \Gamma) \models P$ amounts to checking that
  - $(\rho, \kappa, \Gamma) \models P'$ and
  - $\forall b \in \rho(a) : \kappa(b) \cap \mathscr{N} \subseteq \rho(y)$ (1).

  By induction hypothesis and by the fact stated above, we have that $(\rho, \kappa, \Gamma) \models P'\{v/x\}$. Since $a = x$, then, from the inclusion $\rho(a(\{v/x\})) \subseteq \rho(a)$, (1) implies that $\forall b \in \rho(a(\{v/x\})) : \kappa(b) \cap \mathscr{N} \subseteq \rho(y)$ and this is equivalent to have that $(\rho, \kappa, \Gamma) \models P\{v/x\}$.

□

We recall Lemma 4.11 (the original appears on p. 23).

Lemma 4.11 ***(Congruence)*** *If* $(\rho, \kappa, \Gamma) \models P$ *and* $P \equiv Q$*, then* $(\rho, \kappa, \Gamma) \models Q$.

PROOF. The proof is by induction on the construction of $P \equiv Q$. Here we consider the most interesting cases.
The Case of P is alpha-conversion of Q: since we exploit canonical names to maintain the identity of bound names, changes of bound names do not affect the results of CFA analysis. We have that $\rho(a) = \rho(a') = \rho(\lfloor a \rfloor)$ and $\kappa(a) = \kappa(a') = \kappa(\lfloor a \rfloor)$, where a, a' are names in the equivalent class $\lfloor a \rfloor$.
The Case of $(r_1, \varphi_1, \tilde{\eta}_1)\{(r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \parallel Q\}^{\chi_1} \equiv (r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \parallel (r_1, \varphi_1, \tilde{\eta}_1)\{Q\}^{\chi_1}$: Note that χ_1 and $\jmath$ are the labels of the sub-processes that use resource identifiers r_1 and r_2, respectively. We have that

$$\begin{gathered}(\rho, \kappa, \Gamma) \models (r_1, \varphi_1, \tilde{\eta}_1)\{(r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \parallel Q\}^{\chi_1} \\ \text{iff} \\ (\rho, \kappa, \Gamma) \models ((r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \parallel Q) \wedge (1) \\ \text{iff} \\ (\rho, \kappa, \Gamma) \models (r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \wedge\ ((\rho, \kappa, \Gamma) \models Q \wedge (1)) \\ \text{iff} \\ (\rho, \kappa, \Gamma) \models (r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \wedge\ (\rho, \kappa, \Gamma) \models (r_1, \varphi_1, \tilde{\eta}_1)\{Q\}^{\chi_1} \\ \text{iff} \\ (\rho, \kappa, \Gamma) \models (r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\jmath} \parallel (r_1, \varphi_1, \tilde{\eta}_1)\{Q\}^{\chi_1}\end{gathered}$$

where the condition (1) amounts to $Adm(\varphi, \tilde{\eta}_1, \tilde{\eta}_1') \in \Gamma(r_1)$ where $\tilde{\eta}_1' = H(r_1, ((r_2, \varphi_2, \tilde{\eta}_2)\{\mathbf{0}\}^{\varepsilon} \parallel Q), \chi_1) = H(r_1, Q, \chi_1)$. □

We recall Theorem 4.12 (the original appears on p. 23).

Theorem 4.12 ***(Subject reduction)*** *If* $P \xrightarrow{\mu} P'$, *and* $(\rho, \kappa, \Gamma) \models P$, *then* $(\rho, \kappa, \Gamma) \models P'$.

PROOF. The proof is by induction on the way $P \xrightarrow{\mu} P'$ is obtained using the axioms and the rules in Table 1.
The case $(\mu = \tau)$. In the case of (Act), with label τ it is immediate. Clearly the other cases of the axioms (Act), and the rules $(Open)$, $(Policy_1)$, $(Policy_2)$, and $(Release)$ do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules $(Choice)$, (Par), (Res), $(Cong)$, $(Local)$. The remaining cases are $(Comm)$, $(Comm_R)$, $(Close)$, and $(Acquire)$.

- $(Comm)$: We may assume that $P \equiv P_1 \parallel P_2$ for suitable P_1 and P_2 such that $P_1 \xrightarrow{\bar{a}b} P_1'$ and $P_2 \xrightarrow{a(y)} P_2'$, with $b \in \mathscr{N}$. By Congruence Lemma 4.11, we have that $(\rho, \kappa, \Gamma) \models P$ if only if $(\rho, \kappa, \Gamma) \models P_1 || P_2$, and therefore $(\rho, \kappa, \Gamma) \models P_1$ and $(\rho, \kappa, \Gamma) \models P_2$. The induction hypothesis and the clauses in Table 3, ensure that

$$\begin{gathered}b \in \kappa(a) \ \wedge\ (\rho, \kappa, \Gamma) \models P_1' \\ \kappa(a) \cap \mathscr{N} \subseteq \rho(y) \ \wedge\ (\rho, \kappa, \Gamma) \models P_2'\end{gathered}$$

  By Substitution Lemma 4.10, since $b \in \mathscr{N}$ and $b \in \rho(y)$, we have $(\rho, \kappa, \Gamma) \models P_2'\{b/y\}$ and therefore $(\rho, \kappa, \Gamma) \models P_1' \parallel P_2'\{b/y\}$, which is equivalent to $(\rho, \kappa, \Gamma) \models P'$.

- $(Comm_R)$: by a similar argument. We may assume that $P \equiv P_1 \parallel P_2$ such that $P_1 \xrightarrow{\bar{a}r} P_1'$ and $P_2 \xrightarrow{a(s)} P_2'$, with $r \in \mathscr{R}$. By Congruence Lemma 4.11, we have that $(\rho, \kappa, \Gamma) \models P$ if only if $(\rho, \kappa, \Gamma) \models P_1 || P_2$, and therefore $(\rho, \kappa, \Gamma) \models P_1$ and $(\rho, \kappa, \Gamma) \models P_2$. The induction hypothesis and the clauses in Table 4, ensure that

$$\begin{gathered}r \in \kappa(a) \ \wedge\ (\rho, \kappa, \Gamma) \models P_1' \\ \kappa(a) \cap \mathscr{R} \subseteq \rho(s) \\ \forall r \in \rho(s)\ s.t.\ r \notin \mathsf{fr}(P) : (\rho, \kappa, \Gamma) \models S\{r/s\}(\rho, \kappa, \Gamma) \models P_2'\{r/s\}_R\end{gathered}$$

  This directly establishes $(\rho, \kappa, \Gamma) \models P_1' \parallel P_2'\{r/s\}_R$, which is equivalent to $(\rho, \kappa, \Gamma) \models P'$.

- $(Close)$: by a similar argument. We may assume that $P \equiv P_1 \parallel P_2$ for suitable P_1 and P_2 such that $P_1 \xrightarrow{\bar{a}(b)} P_1'$ and $P_2 \xrightarrow{a(y)} P_2'$, with $b \in \mathscr{N}$. By Congruence Lemma 4.11, we have that $(\rho, \kappa, \Gamma) \models P$ if only if

$(\rho,\kappa,\Gamma) \models P_1||P_2$, and therefore $(\rho,\kappa,\Gamma) \models P_1$ and $(\rho,\kappa,\Gamma) \models P_2$. The induction hypothesis and the clauses in Table 3, ensure that

$$b \in \kappa(a) \ \wedge \ (\rho,\kappa,\Gamma) \models P_1'$$
$$\kappa(a) \cap \mathscr{N} \subseteq \rho(y) \ \wedge \ (\rho,\kappa,\Gamma) \models P_2'$$

By Substitution Lemma 4.10, since $b \in \rho(y)$, we have $(\rho,\kappa,\Gamma) \models P_2'\{b/y\}$ and thus we also have that $(\rho,\kappa,\Gamma) \models (\nu b)(P_1' \parallel P_2'\{b/y\})$, which is equivalent to $(\rho,\kappa,\Gamma) \models P'$.

- (*Acquire*): P is in the form $(r,\varphi,\tilde{\eta})\{\mathbf{0}\}^{\jmath} \parallel req(r)\{Q\}^{\chi}$. We know that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^{\jmath} \parallel req(r)\{Q\}^{\chi}$, and therefore that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^{\jmath}$ and $(\rho,\kappa,\Gamma) \models req(r)\{Q\}^{\chi}$. The clauses in Table 4, ensure that

$$(\rho,\kappa,\Gamma) \models \mathbf{0} \ \wedge \ (\varphi,\tilde{\eta}) \in \Gamma(r)$$
$$(\rho,\kappa,\Gamma) \models Q \ \wedge \ (cond\ 1)$$

where $(cond\ 1)$ is

$\forall(\varphi,\tilde{\eta}) \in \Gamma(r).\chi \not\leq \tilde{\eta}.(\varphi, Adm(\varphi,\tilde{\eta},\tilde{\eta}')) \in \Gamma(r)$ where $\tilde{\eta}' = H(r,Q,\chi)$.

What we need to prove is that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{Q\}^{\chi}$. Since $(\varphi,\tilde{\eta}) \in \Gamma(r)$, the required results is then established. More precisely, we have that

$$(\rho,\kappa,\Gamma) \models Q \ \wedge \ (cond'\ 1)$$

where $(cond'\ 1)$ is $(\varphi, Adm(\varphi,\tilde{\eta},\tilde{\eta}')) \in \Gamma(r)$ where $\tilde{\eta}' = H(r,Q,\chi)$.

and therefore, we can conclude that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{Q\}^{\chi}$.

The case $(\mu = \bar{a}b)$. In the output case of (Act), it is immediate. We have that $P = \mu.P'$. The clauses in Table 3 directly ensure that $(\rho,\kappa,\Gamma) \models P'$ The other axioms do not apply, as well as the rules $(Open)$, $(Close)$, $(Comm)$, and $(Comm_R)$, while the property is preserved by the remaining rules, thanks to the Congruence Lemma 4.11 and the induction hypothesis.

The case $(\mu = \bar{a}r)$. In the output case of (Act_R), it is immediate. We have that $P = \mu.P'$. The clauses in Table 4 directly ensure that $(\rho,\kappa,\Gamma) \models P'$ The other axioms do not apply, as well as the rules $(Open)$, $(Close)$, $(Comm)$, and $(Comm_R)$, while the property is preserved by the remaining rules, thanks to the Congruence Lemma 4.11 and the induction hypothesis.

The case $(\mu = \bar{a}(y))$. In the case of rule $(Open)$, it follows from the clauses in Table 3 that $y \in \kappa(a)$ and that $(\rho,\kappa,\Gamma) \models P'$. The axioms and the rules do not apply, as well as the rules $(Close)$, $(Comm)$, and $(Comm_R)$, while the property is preserved by the remaining rules, thanks to the Congruence Lemma 4.11 and the induction hypothesis.

The case $(\mu = a(y))$. In the input case of rule (Act), it is immediate. We have that $P = \mu.P'$. The clauses in Table 3 directly ensure that $(\rho,\kappa,\Gamma) \models P'$ The other axioms do not apply, as well as the rules $(Open)$, $(Close)$, $(Comm)$, and $(Comm_R)$, while the property is preserved by the remaining rules, thanks to the Congruence Lemma 4.11 and the induction hypothesis.

The case $(\mu = a(s))$. In the input case of rule (Act_R) it follows from the clauses in Table 4 that $(\rho,\kappa,\Gamma) \models P'$ and that for each $b \in \kappa(a) \cap \mathscr{R}$, we have that $r \in \rho(s)$ and $\forall r \in \rho(s).(\rho,\kappa,\Gamma) \models P'\{r/s\}_R$ The other axioms do not apply, as well as the rules $(Open)$, $(Close)$, $(Comm)$, and $(Comm_R)$, while the property is preserved by the remaining rules, thanks to the Congruence Lemma 4.11 and the induction hypothesis.

The cases $(\mu = \alpha?r)$ and $(\mu = rel?r)$. In the case of (Act_R), with label $\mu = \alpha?r$ or $\mu = rel?r$ it is immediate. We have that $P = \mu.P'$. The clauses in Table 4 directly ensure that $(\rho,\kappa,\Gamma) \models P'$ The other axioms and the rules $(Open)$, $(Policy_1)$, $(Policy_2)$, and $(Release)$ do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules $(Choice)$, (Par), (Res), $(Cong)$, $(Local)$.

The case $(\mu = \alpha(r))$. The axioms and the rules $(Open)$, $(Close)$, $(Comm)$, $(Comm_R)$, $(Acquire)$, $(Release)$ and $(Policy_2)$ do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules $(Choice)$, (Par), (Res), $(Cong)$, $(Local)$. The only case to be considered is that of $(Policy_1)$. We may assume that $P \equiv (r,\varphi,\tilde{\eta})\{\alpha(r).Q\}^{\chi}$ for suitable Q and $\tilde{\eta}.(\alpha,\chi) \models \varphi$. From the clauses in Table 4, we have that $(\rho,\kappa,\Gamma) \models \alpha(r).Q$ (and in turn that $(\rho,\kappa,\Gamma) \models Q$), and that $\tilde{\eta}.(\alpha,\chi).\tilde{\eta}' \in \Gamma(r)$, where $H(r,\alpha(r).Q,\chi) = (\alpha,\chi).\tilde{\eta}'.\tilde{\eta}''$ is included in $H(r,P,\chi)$. We have that $\tilde{\eta}.(\alpha,\chi).\tilde{\eta}' \models \varphi$ (where $\tilde{\eta}''$ is possibly empty). Since

$(\alpha,\chi).H(r,Q,\chi) = H(r,\alpha(r).Q,\chi)$, we have that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta}.(\alpha,\chi))\{Q\}^{\chi}$. We can therefore conclude that $(\rho,\kappa,\Gamma) \models \overline{P'}$.
The case ($\mu = \overline{\alpha(r)}$). The axioms and the rules (*Open*), (*Close*), (*Comm*), ($Comm_R$), (*Acquire*), (*Release*) and ($Policy_1$) do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules (*Choice*), (*Par*), (*Res*), (*Cong*), (*Local*). The only case to be considered is that of ($Policy_1$). We may assume that $P \equiv (r,\varphi,\tilde{\eta})\{\alpha(r).Q\}^{\chi}$ for suitable Q and that $\tilde{\eta}.(\alpha,\chi) \not\models \varphi$. From the clauses in Table 4, we have that $(\rho,\kappa,\Gamma) \models \alpha(r).Q$ (and in turn that $(\rho,\kappa,\Gamma) \models Q$), and $\tilde{\eta}.(\overline{\alpha},\chi) \in \Gamma(r)$, where $H(r,\alpha(r).Q,\chi) = (\alpha,\chi).\tilde{\eta}'$ for some $\tilde{\eta}'$. These conditions suffice to obtain that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta}.(\overline{\alpha},\chi))\{\mathbf{0}\}^{\jmath}||Q$, from which we can conclude that $(\rho,\kappa,\Gamma) \models P'$.
The case ($\mu = rel(r)$). The axioms and the rules (*Open*), (*Close*), (*Comm*), ($Comm_R$), (*Acquire*), ($Policy_1$) and ($Policy_2$) do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules (*Choice*), (*Par*), (*Res*), (*Cong*), (*Local*). The only case to be considered is that of (*Release*). We may assume that $P \equiv (r,\varphi,\tilde{\eta})\{rel(r).Q\}^{\chi}$, for suitable Q, such that Q does not include any further access operation on r. From the clauses in Table 4, we have that $(\rho,\kappa,\Gamma) \models rel(r).Q$ (and in turn that $(\rho,\kappa,\Gamma) \models Q$), and $\tilde{\eta}.(rel,\chi) \in \Gamma(r)$. These conditions suffice to obtain that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta}.(rel,\chi))\{\mathbf{0}\}^{\jmath}||Q$, from which we can conclude that $(\rho,\kappa,\Gamma) \models P'$.

□

We recall Theorem 4.17 (the original appears on p. 24).

Theorem 4.17 *If P* respects *the policy φ for r then P* complies with *φ.*

PROOF. By contraposition, we suppose that P does not comply with φ, e.g. that there exist P',P'' such that $P \xrightarrow{\mu}{}^{*} P' \xrightarrow{\overline{\alpha(r)}} P''$, where $\alpha(r)$ is the first violation action occurred in the sequence of transitions. The proof of the theorem amounts to checking that P does not respect φ for r.
For this we proceed by cases on P' making use of the induction hypothesis. By the Corollary 4.13 of the Subject Reduction Theorem, we know that $(\rho,\kappa,\Gamma) \models P$ implies $(\rho,\kappa,\Gamma) \models P''$.

The axioms and the rules (*Open*), (*Close*), (*Comm*), ($Comm_R$), (*Acquire*), (*Release*) and ($Policy_1$) do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules (*Choice*), (*Par*), (*Res*), (*Cong*), (*Local*). The only case to be considered is that of ($Policy_2$). We may assume that $P \equiv (r,\varphi,\tilde{\eta})\{\alpha(r).Q\}^{\chi}$ for suitable Q and that $\tilde{\eta}.(\alpha,\chi) \not\models \varphi$. (α,χ) is the first pair obtained by computing $H(r,Q,\chi)$, i.e. $H(r,Q,\chi) = (\alpha,\chi).\tilde{\eta}'$, for some $\tilde{\eta}'$. From the clauses in Table 4, we therefore obtain the inclusion of the faulty trace $(\varphi,\tilde{\eta}.(\overline{\alpha},\chi))$ in $\Gamma(r)$, which suffices to prove that P does not respect φ on r.

□

Proofs of the main results of CFA, step 2 and 3

We recall Theorem 4.23 (the original appears on p. 27).

Theorem 4.23 ***(Subject reduction)*** *If $P \xrightarrow{\mu} P'$, and $(\rho,\kappa,\Gamma) \models P$, then $(\rho,\kappa,\Gamma) \models P'$.*

PROOF. The proof is by induction on the way $P \xrightarrow{\mu} P'$ is obtained using the axioms and the rules in Table 1. We only consider the most interesting cases. The others are similar to the ones in the proof of Theorem 4.12.
The case ($\mu = \tau$) with the rule (*Acquire*), where P is in the form $(r,\varphi,\check{\eta})\{\mathbf{0}\}^{\jmath} \parallel req(r)\{F\}^{\chi}$. We know that $(\rho,\kappa,\Gamma) \models (r,\varphi,\check{\eta})\{\mathbf{0}\}^{\jmath} \parallel req(r)\{F\}^{\chi}$, and therefore that $(\rho,\kappa,\Gamma) \models (r,\varphi,\check{\eta})\{\mathbf{0}\}^{\jmath}$ and $(\rho,\kappa,\Gamma) \models req(r)\{F\}^{\chi}$. The clauses in Table 4, suitably modified with the ones in Table 5, ensure that

$$(\rho,\kappa,\Gamma) \models \mathbf{0} \ \wedge \ (\varphi,\check{\eta}) \in \Gamma(r)$$
$$(\rho,\kappa,\Gamma) \models F \ \wedge \ (cond\ 1)$$

where $(cond\ 1)$ is

$$\forall(\varphi,\check{\eta}) \in \Gamma(r).\chi \not\pounds \tilde{\eta} \ \wedge\ \forall\check{\eta}' \in SH(r,F,\chi)$$
$$(\varphi,Adm(\varphi,\check{\eta},\check{\eta}') \in \Gamma(r)$$

What we need to prove is that $(\rho,\kappa,\Gamma) \models (r,\varphi,\check{\eta})\{F\}^{\chi}$. Since $(\varphi,\check{\eta}) \in \Gamma(r)$, the required results is then established. More precisely, we have that

$$(\rho,\kappa,\Gamma) \models F \ \wedge\ (cond'\ 1)$$

where $(cond'\ 1)$ is

$$\forall\check{\eta}' \in SH(r,F,\chi)$$
$$(\varphi,Adm(\varphi,\check{\eta},\check{\eta}') \in \Gamma(r)$$

and therefore, we can conclude that $(\rho,\kappa,\Gamma) \models (r,\varphi,\tilde{\eta})\{F\}^{\chi}$.

The case $(\mu = \alpha^{\lambda}(r))$. The axioms and the rules $(Open)$, $(Close)$, $(Comm)$, $(Comm_R)$, $(Acquire)$, $(Release)$ and $(Policy_2)$ do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules $(Choice)$, (Par), (Res), $(Cong)$, $(Local)$. The only case to be considered is that of $(Policy_1)$. We may assume that $P \equiv (r,\varphi,\check{\eta})\{F\}^{\chi}$ for suitable F, such that the prefix $\alpha(r)$ is at top-level in F, i.e. $F = \Pi_{i=0}^{n} F_i$ and there is an index i such that $F_i = \alpha^{\lambda}(r).F'$. We also assume that $\check{\eta}.(\alpha,\chi,\lambda) \models \varphi$. From the clauses in Table 4, suitably modified with the ones in Table 5, we have that $(\rho,\kappa,\Gamma) \models \alpha^{\lambda}(r).R$ (and in turn that $(\rho,\kappa,\Gamma) \models R$), and that $H'(r,\alpha^{\lambda}(r).F_i,\chi) = (\alpha,\chi,\lambda).\check{\eta}'.\check{\eta}''$ belongs to $SH(r,F,\chi)$, and therefore to $SH(r,P,\chi)$. Since $\check{\eta}.(\alpha,\chi,\lambda).\check{\eta}' \models \varphi$ (with $\check{\eta}''$ possibly empty), we have that $\check{\eta}.(\alpha,\chi,\lambda).\check{\eta}' \in \Gamma(r)$. Since $(\alpha,\chi,\lambda).H'(r,F_i,\chi) = H'(r,\alpha^{\lambda}(r).F_i,\chi)$ still belongs to $SH(r,F,\chi)$, and therefore to $SH(r,P,\chi)$, we have that $(\rho,\kappa,\Gamma) \models (r,\varphi,\check{\eta}.(\alpha,\chi,\lambda))\{F\}^{\chi}$. We can therefore conclude that $(\rho,\kappa,\Gamma) \models P'$.

□

We recall Theorem 4.28 (the original appears on p. 31).

Theorem 4.28 ***(Subject reduction)*** *If* $P \xrightarrow{\mu} P'$, *and* $(\rho,\kappa,\Gamma,\Delta_\omega) \models P$, *then* $(\rho,\kappa,\Gamma,\Delta_\omega) \models P'$.

PROOF. The proof is by induction on the way $P \xrightarrow{\mu} P'$ is obtained using the axioms and the rules in Table 1 plus the new rule $(Repeated\ Acquire)$. We only consider the most interesting cases. The others are similar to the ones in the proof of Theorem 4.12.

- $(Repeated\ Acquire)$: P is in the form $!req(r)\{R\}^{\chi_!} \parallel (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^{\flat}$. We know that $(\rho,\kappa,\Gamma,\Delta_\omega) \models P$, and, as a consequence, that $(\rho,\kappa,\Gamma,\Delta_\omega) \models (r,\varphi,\tilde{\eta})\{\mathbf{0}\}^{\flat}$ and $(\rho,\kappa,\Gamma,\Delta_\omega) \models !req(r)\{R\}^{\chi_!}$. The clauses in Table 6, ensure that

  $$(\rho,\kappa,\Gamma,\Delta_\omega) \models \mathbf{0} \ \wedge\ (\varphi,\tilde{\eta}) \in \Gamma(r)$$
  $$(\rho,\kappa,\Gamma,\Delta_\omega) \models R \ \wedge\ (cond\ 1)$$

  where $(cond\ 1)$ is

  $\forall(\varphi,\hat{\eta}) \in \Gamma(r).\forall\hat{\eta}' \in Mix(\Delta_\omega(r))(\varphi,Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$, where $[H(r,R,\chi_!)] \in \Delta_\omega(r)$.

  We need to prove that $(\rho,\kappa,\Gamma,\Delta_\omega) \models P'$, where $P' = !req(r)\{R\}^{\chi_!} \parallel (r,\varphi,\tilde{\eta})\{R\}^{\chi_!}$. Consequently, we need to prove that $(\rho,\kappa,\Gamma,\Delta_\omega) \models (r,\varphi,\tilde{\eta})\{R\}^{\chi_!}$ and $(\rho,\kappa,\Gamma,\Delta_\omega) \models !req(r)\{R\}^{\chi_!}$.

  Since $(\varphi,\hat{\eta}) \in \Gamma(r)$, the required results is then established. More precisely, we have that

  $$(\rho,\kappa,\Gamma,\Delta_\omega) \models R \ \wedge\ (cond'\ 1)$$

  where $(cond'\ 1)$ is $\forall\hat{\eta}' \in Mix(\Delta_\omega(r))(\varphi,Adm(\varphi,\hat{\eta},\hat{\eta}')) \in \Gamma(r)$, where $[H(r,R,\chi_!)] \in \Delta(r)$.

  We can therefore conclude that $(\rho,\kappa,\Gamma,\Delta_\omega) \models (r,\varphi,\hat{\eta})\{R\}^{\chi_!}$.

- **The case** $(\mu = \alpha(r))$. The axioms and the rules $(Open)$, $(Close)$, $(Comm)$, $(Comm_R)$, $(Acquire)$, $(Release)$ and $(Policy_2)$ do not apply. By Congruence Lemma 4.11 and the induction hypothesis, the property is preserved by the rules $(Choice)$, (Par), (Res), $(Cong)$, $(Local)$. The only case to be considered is that

of $(Policy_1)$. We may assume that $P \equiv (r, \varphi, \hat{\eta})\{R\}^{\chi_!}$ for suitable S, such that the prefix $\alpha(r)$ is at top-level in R, i.e. $R = \alpha(r).R'$. We also assume that $\hat{\eta}.(\alpha, \chi_\omega) \models \varphi$. From the clauses in Table 4, suitably modified with the ones in Table 6, we have that $(\rho, \kappa, \Gamma, \Delta_\omega) \models \alpha(r).R$ (and in turn that $(\rho, \kappa, \Gamma, \Delta_\omega) \models R$), $H''(r, \alpha(r).R, \chi_\omega) = (\alpha, \chi_\omega).\hat{\eta}'.\hat{\eta}''$ belongs to $\Delta_\omega(r)$, and that $\exists \hat{\eta}'.\hat{\eta}.(\alpha, \chi_\omega).\hat{\eta}'.\hat{\eta}'' \in Mix(\Delta(r))$. Since $\varphi \models \hat{\eta}.(\alpha, \chi_!).\hat{\eta}'$ (with $\hat{\eta}''$ possibly empty), we have that $\hat{\eta}.(\alpha, \chi_!).\hat{\eta}' \in \Gamma(r)$. Since $H''(r, \alpha(r).R, \chi) = (\alpha, \chi_!).H''(r, R, \chi)$ still belongs to $\Delta_\omega(r)$, we have $(\rho, \kappa, \Gamma, \Delta_\omega) \models (r, \varphi, \hat{\eta}.(\alpha, \chi_\omega))\{R\}^{\chi}$. We can therefore conclude that $(\rho, \kappa, \Gamma, \Delta_\omega) \models P'$.

□