A Calculus for Modeling Floating Authorizations

Jovanka Pantović, Ivan Prokić, Hugo Torres Vieira

A Calculus for Modeling Floating Authorizations

Jovanka Pantović[1 [0000−0002−3974−5064]], Ivan Prokić[1 [0000−0001−5420−1527]], and Hugo Torres Vieira[2 [0000−0001−7461−6156]]

[1] Faculty of Technical Sciences, University of Novi Sad, Serbia,
[2] IMT School for Advanced Studies Lucca, Italy

Abstract. Controlling resource usage in distributed systems is a challenging task given the dynamics involved in access granting. Consider, e.g., the setting of floating licenses where access can be granted if the request originates in a licensed domain and if the number of active users is within the license limits. Access granting in such scenarios is given in terms of floating authorizations, addressed in this paper as first class entities of a process calculus model, encompassing the notions of domain, accounting and delegation. We present the operational semantics of the model in two equivalent alternative ways, each informing on the specific nature of authorizations. We also introduce a typing discipline to single out systems that never get stuck due to lacking authorizations, addressing configurations where authorization assignment is not statically prescribed in the system specification.

1 Introduction

Despite the continuous increase of computational resources, their usage will always nevertheless be subject to accessibility and availability. Regardless whether such resources are of hardware or software in nature, they might have finite or virtually infinite capabilities. Physical examples, that can be mapped to finite capabilities directly, include devices such as printers or cell phones, and components of a computing system such as memory or processors. Virtual examples, such as a shared memory cell or a web service, can be more easily seen as having infinite potential but often their availability is also finitely constrained. In general, ensuring proper resource usage is a crucial yet non trivial task, given the highly dynamic nature of access requests and the flexibility necessary to handle such requests, while ensuring secure and efficient system operation.

For security purposes it is crucial to control that access is granted only to authorized users, but also to enforce that access granting is subject to availability. Concrete examples include a wireless access point that has a given access policy and a limited capacity on the number of connected devices, and a software application that is licensed to be used by an institution in a bounded way. Both examples address limited capabilities that are accessible in a shared way to authorized users. We thus identify a notion of *floating* authorization, adopting the terminology from the licensing realm, where access to resources can be granted to any authorized user in a given *domain* up to the point the capacity is reached. We also identify a notion of implicit granting since users may be granted access directly when using the resource (e.g., running the licensed software).

Considering the licensing setting, we find further examples that inform on the dynamic nature of authorizations. For example, a user deploys a software application on

the cloud and provides the license himself (cf. *Bring Your Own License* [6]), potentially losing access to run the application locally given the capacity constraints. We identify here a notion of authorization *delegation*, where the intention to yield or obtain an authorization is explicit. We therefore distill the following dimensions in a floating authorizations model: domain so as to specify where access may be (implicitly) granted; accounting so as to capture capacity; and delegation so as to model (explicit) authorization granting.

In this paper, we present a model that encompasses these dimensions, developed focusing on a process calculus tailored for communication centered systems. In our model the only resources considered are communication channels, so our exploration on authorization control is carried out considering authorizations refer to channels and their usage (for communication) is controlled. Our development builds on the π-calculus [14], which provides the basis to model communicating systems including name passing, and on a model for authorizations [8], from where we adopt the language constructs for authorization domain scope and delegation. We adapt here the interpretation for authorization domains so as to encompass accounting, which thus allows us to model floating authorizations. To the best of our knowledge, ours is the first process calculus model that addresses floating resources (in our case authorizations) as first class entities. Naturally, this does not mean our language cannot be represented in more canonical models, only that we are unaware of existing approaches that offer abstractions that support reasoning on floating resources in a dedicated way.

After presenting the model, we show a typing discipline that ensures systems never incur in authorization errors, i.e., systems that are blocked due to lacking authorizations. Our type analysis addresses systems for which the authorization assignment is not statically given in the system specification, in particular when authorizations for (some) received names are already held by the receiving parties, a notion we call contextual authorizations. We first discuss some examples that informally introduce our model.

1.1 Model Preview

We present our language by resorting to the licensing setting for the sake of a more intuitive reading. First of all, to model authorization domains we consider a scoping construct, and we write $(\mathit{license})\mathit{University}$ to represent that the $\mathit{University}$ domain holds one $\mathit{license}$. This means that one use of $\mathit{license}$ within $\mathit{University}$ is authorized. In particular, if $\mathit{University}$ comprises two students that are simultaneously active, which we dub Alice and Bob, we may write $(\mathit{license})(\mathit{Alice} \mid \mathit{Bob})$ in which case either Alice or Bob can be granted the $\mathit{license}$ that is floating, but not both of them.

The idea is to support a notion of accounting, so when one of the students uses the license the scope is confined accordingly. For example, if Bob uses the license and evolves to $\mathit{LicensedBob}$ then the system evolves to $\mathit{Alice} \mid (\mathit{license})\mathit{LincensedBob}$ where the change in the scope denotes that $\mathit{license}$ is not available to Alice anymore. The evolution of the system is such that the authorization is directly confined to Bob, who is using it, so as to model implicit granting. At this point Alice cannot implicitly grab the authorization and gets stuck if she tries to use $\mathit{license}$. Hence, name $\mathit{license}$ may be shared between Alice and Bob, so the (change of) scoping does not mean the name is privately held by $\mathit{LicensedBob}$, just the authorization.

Now consider that Bob and $Carol$ want to explicitly exchange an authorization, which can be carried out by a delegation mechanism supported by two communication primitives. We write $auth\langle license\rangle.UnlicensedBob$ to represent the explicit delegation of one authorization for $license$ via communication channel $auth$, after which activating configuration $UnlicensedBob$. Instead, by $auth(license).LicensedCarol$ we represent the dual primitive that allows to receive one authorization for $license$ via channel $auth$ after which activating configuration $LicensedCarol$. So by

$$(license)(auth)auth\langle license\rangle.UnlicensedBob \mid (auth)auth(license).LicensedCarol$$

we represent a system where the authorization for $license$ can be transferred leading to

$$(auth)UnlicensedBob \mid (auth)(license)LicensedCarol$$

where the scope of the authorization for $license$ changes accordingly. The underlying communication is carried out by a synchronization on channel $auth$, for which we remark the respective authorizations $(auth)$ are present. In fact, in our model channels are the only resources and their usage is subject to the authorization granting mechanism.

Our model supports a form of fairness and does not allow a "greedy" usage of resources. For example, in the configuration $(license)(Alice \mid (license)LincensedBob)$ the user $LincensedBob$ can only be granted the closest $(license)$ first, not interfering with $Alice$, but can be also granted the top-level $license$ if he needs both licenses.

We remark that no name passing is involved in the authorization delegation mechanism and that name $license$ is known to both ends in the first place. Instead, name passing is supported by dedicated primitives, namely

$$(comm)comm!license.Alice \mid (comm)comm?x.Dylan$$

represents a system where the name $license$ can be passed via a synchronization on channel $comm$, leading to the activation of $Alice$ and $Dylan$ where, in the latter, the placeholder x is instantiated by $license$. Notice that the synchronization can take place since the authorizations to use channel $comm$ are given, one for each endpoint.

Name passing allows to model systems where access to channels changes dynamically (since the communicated names refer to channels) but, as hinted in the previous examples, knowing a name does not mean being authorized to use it. So, for instance, $(comm)comm?x.x!reply.0$ specifies a reception in (authorized) $comm$ where the received name is then used to output $reply$, leading to an inactive state (denoted by 0). Receiving $license$ in channel $comm$ leads to $(comm)license!reply.0$ where the authorization for $comm$ is still present but no authorization for $license$ is acquired as a result of the communication. Hence the output specified using $license$ is not authorized and cannot take place. We remark that an authorization for $reply$ is not required, hence communicating a name does not entail usage for the purpose of authorizations. By separating name passing and authorization delegation we are then able to model systems where unauthorized intermediaries (e.g., brokers) may be involved in forwarding names between authorized parties without ever being authorized to use such names. For example $(comm)comm?x.(forward)forward!x.0.$ which requires no further authorizations.

Configuration $(comm)comm?x.(auth)auth(x).LicensedDylan$ shows a possible pattern for authorizing received names where, after the reception on $comm$, an authorization reception for the received name (using placeholder x) via $auth$ is specified, upon which the authorization to use the received name is acquired. Another possibility for enabling authorizations for received names is to use the authorization scoping, e.g., $(comm)comm?x.(x)LicensedDylan$ where the authorization (x) is instantiated by the received name. This example hints on the fact that the authorization scoping is a powerful mechanism that may therefore be reserved in some circumstances to the Trusted Computing Base, resorting to authorization delegation elsewhere.

To introduce the last constructs of our language, consider the system

$$!(license)license?x.(x)license\langle x\rangle.0 \mid (\nu fresh)(license)license!fresh.license(fresh).0$$

where a licensing server is specified on the left hand side, used in the specification given on the right hand side. By $(\nu fresh)Domain$ we represent the creation of a new name which is private to $Domain$, so the specification on the right hand side can be read as first create a name, then send it via channel $license$, after which receive the authorization to use the *fresh* name via channel $license$ and then terminate (the received authorization is not actually used in this simple example). On the left hand side, we find a replicated (i.e., repeatably available) reception on channel $license$, after which an authorization scope for the received name is specified that may then be delegated away.

We now present our type analysis returning to the university scenario. Consider $(exam)(minitest)(alice)alice?x.x!task.0$ where there are available authorizations for $exam$ and for $minitest$, and where $alice$ is waiting to receive the channel on which she will send $task$. Assuming that she can only receive $exam$ or $minitest$ the authorizations specified are sufficient. Which authorization will actually be used depends on the received name, so the authorization is implicitly taken when the received channel is used. However, if a name $viva$ is sent then the provided authorizations do not suffice.

In order to capture the fact that the above configuration is authorization safe, provided it is inserted in a context that matches the assumptions described previously, our type analysis records the names that can be safely communicated. For instance, we may say that only names $exam$ and $minitest$ can be communicated in channel $alice$. Also, consider that $exam$ and $minitest$ can only receive values that are not subject to authorization control. We then denote by $\{alice\}(\{exam, minitest\}(\emptyset))$ the type of channel $alice$ in such circumstances, i.e., that it is not subject to replacement (we will return to this point), and that it can be used to communicate $exam$ and $minitest$ that in turn cannot be used for communication (typed with $\emptyset$), reading from left to right. Using this information, we can ensure that the specification given for $alice$ above is safe, since all names that will possibly be used in communications are contextually authorized.

To analyze the use of the input variable x we then take into account that it can be instantiated by either $exam$ or $minitest$ (which cannot be used for channel communication) so the type of x is $\{exam, minitest\}(\emptyset)$. Hence the need to talk about possible replacements of a name, allowing us to uniformly address names that are bound in inputs. Our types, denoted $\gamma(T)$, are then built out of two parts, one addressing possible replacements of the name identity itself (γ), and the other informing on the (type of the) names that may be exchanged in the channel (T). The typing assumption $alice : \{alice\}(\{exam, minitest\}(\emptyset))$ informs on the possible contexts where the

$P ::= 0$	(Inaction)	$a!b.P$	(Output)	$a\langle b\rangle.P$	(Send authorization)
$P \mid P$	(Parallel)	$a?x.P$	(Input)	$a(b).P$	(Receive authorization)
$(\nu a)P$	(Restriction)	$(a)P$	(Authorization)	$!(a)a?x.P$	(Replicated input)

Table 1. Syntax of processes.

system above can be safely used. For instance, it is safe to compose with the system $(alice)alice!minitest$ where $minitest$ is sent to $alice$, since the name to be sent belongs to the names expected on $alice$. Instead, consider configuration

$$(exam)(minitest)((alice)alice?x.x!task.0 \mid (bob)bob?x.x!task.0)$$

which is also safe and addressed by our typing analysis considering typing assumptions $alice : \{alice\}(\{exam\}(\emptyset))$ and $bob : \{bob\}(\{minitest\}(\emptyset))$. Notice that which authorization is needed by each student is not statically specified in the system, which is safe when both $exam$ and $minitest$ are sent given the authorization scopes can be confined accordingly. Clearly, the typing specification already yields a specific association.

The typing analysis shown in Section 3 addresses such configurations where authorizations for received names may be provided by the context. In Section 2 we present the operational semantics of our language considering two equivalent alternatives that inform on the specific nature of authorizations in our model. We refer to the supporting document [13] for additional material, namely proofs of the results reported here.

2 A Model of Floating Authorizations

In this section, we present our process model, an extension of the π-calculus [14] with specialized constructs regarding authorizations adopted from a model for authorizations [8]. The syntax of the language is given in Table 1, assuming a countable set of *names* $\mathcal{N}$, ranged over by $a, b, c, \ldots, x, y, z, \ldots$ We briefly present the constructs adopted from the π-calculus. An inactive process is represented by 0. By $P \mid P$ we represent two processes simultaneously active, that may interact via synchronization in channels. The name restriction construct $(\nu a)P$ specifies the creation of a channel name a that is known only to the process P. The output prefixed process $a!b.P$ sends the name b on channel a and proceeds as P, and the input prefixed process $a?x.P$ receives a name on channel a and replaces name x in P with the received name.

The remaining constructs involve authorization specifications. Term $(a)P$ is the authorization scoping, representing that process P has one authorization to use channel a. In contrast with name restriction, name a is not private to P. Term $a\langle b\rangle.P$ represents the process that delegates one authorization for name b along name a and proceeds as P. Term $a(b).P$ represents the dual, i.e., a process which receives one authorization for name b along name a and proceeds as $(b)P$. Term $!(a)a?x.P$ allows to specify infinite behavior: the process receives the name along (authorized) name a and substitutes x in P with the received name, which is activated in parallel with the original process. We adopt a simple form of infinite processes, but we remark that a general replication

construct can be encoded following standard lines using replicated input, as discussed later.

In $(\nu x)P$, $a?x.P$ and $!(a)a?x.P$ the name x is *binding* with scope P. As usual, we use $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ to denote the sets of free and bound names in P, respectively. In $(a)P$ occurrence of the name a is free and occurrences of names a and b in processes $a\langle b\rangle.P$ and $a(b).P$ are also free. We remark that in our model authorization scope extrusion is not applicable since a free name is specified, unlike name restriction, and constructs to send and receive authorizations only affect the scope of authorizations. For the rest of presentation we use the following abbreviations: α_a stands for $a!b, a?x, a\langle b\rangle$ or $a(b)$ (including when $b = a$) and $(\nu\tilde{a})$ stands for $(\nu a_1)\ldots(\nu a_n)$.

2.1 Action Semantics

As in the π-calculus, the essence of the behavior of processes can be seen as communication. Specific to our model is that two processes ready to synchronize on a channel must be authorized to use the channel. For example, consider that $(a)a!b.P \mid (a)a?x.Q$ can evolve to $(a)P \mid (a)Q\{b/x\}$, since both sending and receiving actions are authorized, while $(a)a!b.P \mid a?x.Q$ lacks the proper authorization on the receiving end, hence the synchronization cannot occur. Another specific aspect of our language is authorization delegation. For example, in $(a)(b)a\langle b\rangle.P \mid (a)a(b).Q$ both actions along name a are authorized and the delegating process has the authorization on b, hence the delegation can take place, leading to $(a)P \mid (a)(b)Q$ where the authorization scope for b changes. If actions along name a are not authorized or the process delegating lacks the authorization for b, then the synchronization is not possible. We formally define the behavior of processes by means of a labeled transition system and afterwards by means of a reduction semantics, which are shown equivalent but inform in somewhat different ways on the specific nature of authorizations in our model.

The labeled transition system (LTS) relies on observable actions, ranged over by α, defined as follows:

$$\alpha ::= (a)^i a!b \mid (a)^i a?b \mid (a)^i(b)^j a\langle b\rangle \mid (a)^i a(b) \mid (\nu b)(a)^i a!b \mid \tau_\omega$$

where ω is of the form $(a)^{i+j}(b)^k$ and $i, j, k \in \{0, 1\}$ and it may be the case that $a = b$. We may recognize the communication action prefixes together with annotations that capture carried/lacking authorizations and bound names. Intuitively, a communication action tagged with $(a)^0$ represents the action lacks an authorization on a, while $(a)^1$ represents the action is carrying an authorization on a. In the case for authorization delegation two such annotations are present, one for each name involved. As in π-calculus, (νb) denotes that the name in the object of the output is bound. In the case of internal steps, the ω identifies the lacking authorizations, and we use τ to abbreviate $\tau_{(a)^0(b)^0}$ where no authorizations are lacking. By $\mathsf{n}(\alpha)$, $\mathsf{fn}(\alpha)$ and $\mathsf{bn}(\alpha)$ we denote the set of all, free and bound names of α.

The transition relation is the least relation included in $\mathcal{P} \times \mathcal{A} \times \mathcal{P}$, where $\mathcal{P}$ is the set of all processes and $\mathcal{A}$ is the set of all actions, that satisfies the rules in Table 2, which we now briefly describe. The rules (L-OUT), (L-IN), (L-OUT-A), (L-IN-A) capture the actions that correspond to the communication prefixes. In each rule the continuation

(L-OUT) $a!b.P \xrightarrow{a!b} (a)P$ (L-IN) $a?x.P \xrightarrow{a?b} (a)P\{b/x\}$ (L-OUT-A) $a\langle b\rangle.P \xrightarrow{a\langle b\rangle} (a)P$ (L-IN-A) $a(b).P \xrightarrow{a(b)} (a)(b)P$

(L-IN-REP) $!(a)a?x.P \xrightarrow{(a)a?b} (a)P\{b/x\} \mid !(a)a?x.P$

(L-PAR) $\dfrac{P \xrightarrow{\alpha} Q \quad \mathsf{bn}(\alpha) \cap \mathsf{fn}(R) = \emptyset}{P \mid R \xrightarrow{\alpha} Q \mid R}$

(L-RES) $\dfrac{P \xrightarrow{\alpha} Q \quad a \notin n(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)Q}$ (L-OPEN) $\dfrac{P \xrightarrow{(a)^i a!b} Q \quad a \neq b}{(\nu b)P \xrightarrow{(\nu b)(a)^i a!b} Q}$ (L-SCOPE-INT) $\dfrac{P \xrightarrow{\tau_{\omega(a)}} Q}{(a)P \xrightarrow{\tau_\omega} Q}$ (L-SCOPE-EXT) $\dfrac{P \xrightarrow{\sigma_a} Q}{(a)P \xrightarrow{(a)\sigma_a} Q}$

(L-SCOPE) $\dfrac{P \xrightarrow{\alpha} Q \quad \tau_{\omega(a)} \neq \alpha \neq \sigma_a}{(a)P \xrightarrow{\alpha} (a)Q}$ (L-COMM) $\dfrac{P \xrightarrow{(a)^i a!b} P' \quad Q \xrightarrow{(a)^j a?b} Q' \quad \omega = (a)^{2-i-j}}{P \mid Q \xrightarrow{\tau_\omega} P' \mid Q'}$

(L-CLOSE) $\dfrac{P \xrightarrow{(\nu a)(b)^i b!a} P' \quad Q \xrightarrow{(b)^j b?a} Q' \quad \omega = (b)^{2-i-j} \quad a \notin fn(Q)}{P \mid Q \xrightarrow{\tau_\omega} (\nu a)(P' \mid Q')}$

(L-AUTH) $\dfrac{P \xrightarrow{(b)^k (a)^i a\langle b\rangle} P' \quad Q \xrightarrow{(a)^j a(b)} Q' \quad \omega = (a)^{2-i-j}(b)^{1-k}}{P \mid Q \xrightarrow{\tau_\omega} P' \mid Q'}$

Table 2. LTS rules.

is activated under the scope of the proper authorizations so as to realize *confinement*. The labels are not decorated with any authorizations, representing that the actions are not carrying any authorizations, omitting $(a)^0$ annotations. In contrast, replicated input is authorized by construction, which is why in (L-IN-REP) the label is decorated with the authorization. Rule (L-PAR) lifts the actions of one of the branches (the symmetric rule is omitted) while avoiding unintended name capture; rule (L-RES) says that actions of P are also actions of $(\nu a)P$, provided that the restricted name is not specified in the action; and (L-OPEN) captures the bound output case, opening the scope of the restricted name a, thus allowing for scope extrusion, all three rules adopted from the π-calculus.

Rule (L-SCOPE-INT) shows the case of a synchronization that lacks an authorization on a, so in the conclusion the action exhibited no longer lacks the respective authorization and leads to a state where the authorization scope is no longer present. We use $\omega(a)$ to abbreviate $(a)^2(b)^k$, $(a)^1(b)^k$, and $(b)^{i+j}(a)^1$, for a given b (including the case $b = a$), in which case ω is obtained by the respective exponent decrement. We remark that in contrast to the extrusion of a restricted name via bound output, where the scope floats *up* to the point a synchronization (rule (L-CLOSE) explained below), authorization scopes actually float *down* to the level of communication prefixes (cf. rules (L-OUT), (L-IN), (L-OUT-A), (L-IN-A)), so as to capture confinement. Rule (L-SCOPE-EXT) follows

(SC-PAR-INACT) $P \mid 0 \equiv P$ (SC-PAR-COMM) $P \mid Q \equiv Q \mid P$ (SC-PAR-ASSOC) $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

(SC-RES-INACT) $(\nu a)0 \equiv 0$ (SC-RES-SWAP) $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$ (SC-REP) $!(a)a?x.P \equiv !(a)a?x.P \mid (a)a?x.P$

(SC-RES-EXTR) $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad (a \notin \mathsf{fn}(P))$ (SC-ALPHA) $P \equiv_\alpha Q \implies P \equiv Q$

(SC-AUTH-SWAP) $(a)(b)P \equiv (b)(a)P$ (SC-AUTH-INACT) $(a)0 \equiv 0$ (SC-SCOPE-AUTH) $(a)(\nu b)P \equiv (\nu b)(a)P \quad \text{if } a \neq b$

Table 3. Structural congruence.

similar lines as (L-SCOPE-INT) as it also refers to lacking authorizations, specifically for the case of an (external) action that is not carrying a necessary authorization. We use σ_a to denote both an action that specifies a as communication subject (cf. α_a) and is annotated with $(a)^0$ (including bound output), and of the form $(b)^i b\langle a\rangle$ where $i \in \{0, 1\}$ (which includes $(a)^1 a\langle a\rangle$). By $(a)\sigma_a$ we denote the respective annotation exponent increase. Rule (L-SCOPE) captures the case of an action that is not lacking an authorization on a, in which case the action crosses seamlessly the authorization scope for a.

The synchronization of parallel processes is expressed by the last three rules, omitting the symmetric cases. In rule (L-COMM) one process is able to send and other to receive a name b along name a so the synchronization may take place. If the sending and receiving actions are not carrying the appropriate authorizations, then the transition label τ_ω specifies the lacking authorizations (the needed two minus the existing ones). In rule (L-CLOSE) one process is able to send a bound name a and the other to receive it, along name b, so the synchronization may occur leading to a configuration where the restriction scope is specified (avoiding unintended name capture) so as to finalize the scope extrusion. The authorization delegation is expressed by rule (L-AUTH), where an extra annotation for ω is considered given the required authorization for the delegated authorization. Carried authorization annotations, considered here up to permutation, thus identify, in a compositional way, the requirements for a synchronization to occur.

2.2 Reduction Semantics

Reduction is defined as a binary relation between processes, denoted $\rightarrow$, so $P \rightarrow Q$ specifies that process P evolves to process Q in one computational step. In order to identify processes which differ syntactically but have the same behavior, we introduce the *structural congruence* relation $\equiv$, which is the least congruence relation between processes satisfying the rules given in Table 3. Most rules are standard considering structural congruence in the π-calculus. In addition we adopt rules introduced previously [8] that address authorization scopes, including (SC-AUTH-INACT) that allows to discard unused authorizations.

Regarding authorization scoping, we remark there is no rule which relates authorization scoping and parallel composition. Adopting a rule of the sort $(a)(P \mid Q) \equiv$

$$\mathcal{C}[\cdot] \;::=\; \cdot \mid P \mid \mathcal{C}[\cdot] \mid (a)\mathcal{C}[\cdot] \qquad \mathcal{C}[\cdot_1,\cdot_2] \;::=\; \mathcal{C}[\cdot_1] \mid \mathcal{C}[\cdot_2] \mid P \mid \mathcal{C}[\cdot_1,\cdot_2] \mid (a)\mathcal{C}[\cdot_1,\cdot_2]$$

Table 4. Contexts with one and two holes.

$$\text{(C-END)}\quad \mathit{drift}(\,\cdot\,;\emptyset;\tilde{d}) = \cdot$$

$$\text{(C-REM)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot];\tilde{a};\tilde{d},c) = \mathcal{C}'[\cdot]}{\mathit{drift}((c)\mathcal{C}[\cdot];\tilde{a},c\,;\tilde{d}) = \mathcal{C}'[\cdot]} \qquad \text{(C-SKIP)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot];\tilde{a};\tilde{d}) = \mathcal{C}'[\cdot] \quad c \notin \tilde{d}}{\mathit{drift}((c)\mathcal{C}[\cdot];\tilde{a};\tilde{d}) = (c)\mathcal{C}'[\cdot]}$$

$$\text{(C-PAR)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot];\tilde{a};\tilde{d}) = \mathcal{C}'[\cdot]}{\mathit{drift}(\mathcal{C}[\cdot] \mid R;\tilde{a};\tilde{d}) = \mathcal{C}'[\cdot] \mid R} \qquad \text{(C2-SPL)}\quad \frac{\mathit{drift}(\mathcal{C}_1[\cdot_1];\tilde{a};\tilde{d}) = \mathcal{C}'_1[\cdot] \quad \mathit{drift}(\mathcal{C}_2[\cdot_2];\tilde{b};\tilde{e}) = \mathcal{C}'_2[\cdot]}{\mathit{drift}(\mathcal{C}_1[\cdot_1] \mid \mathcal{C}_2[\cdot_2];\tilde{a};\tilde{b};\tilde{d};\tilde{e}) = \mathcal{C}'_1[\cdot_1] \mid \mathcal{C}'_2[\cdot_2]}$$

$$\text{(C2-REM-L)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\tilde{d},c;\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}((c)\mathcal{C}[\cdot_1,\cdot_2];\tilde{a},c\,;\tilde{b};\tilde{d};\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2]} \qquad \text{(C2-REM-R)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\tilde{d};\tilde{e},c) = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}((c)\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b},c\,;\tilde{d};\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2]}$$

$$\text{(C2-SKIP)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\tilde{d};\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2] \quad c \notin \tilde{d},\tilde{e}}{\mathit{drift}((c)\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\tilde{d};\tilde{e}) = (c)\mathcal{C}'[\cdot_1,\cdot_2]} \qquad \text{(C2-PAR)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\tilde{d};\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2] \mid R;\tilde{a};\tilde{b};\tilde{d};\tilde{e}) = \mathcal{C}'[\cdot_1,\cdot_2] \mid R}$$

Table 5. *drift* rules.

$(a)P \mid (a)Q$ would represent introducing/discarding one authorization, thus interfering with authorization accounting. We distinguish $(a)(P \mid Q)$ where the authorization is shared between P and Q and $(a)P \mid (a)Q$ where two authorizations are specified, one for each process. Another approach could be a rule of the sort $(a)(P \mid Q) \equiv P \mid (a)Q$, which also may affect the computational power of a process. For example, processes $a!b.0 \mid (a)0$ and $(a)(a!b.0 \mid 0)$ are clearly not to be deemed equal. Structural congruence is therefore not expressive enough to isolate two authorized processes willing to synchronize. To define the reduction relation we introduce an auxiliary notion of static contexts with one and two holes and operation *drift* that allows to single out configurations where communication can occur. Intuitively, reduction is possible if two processes have active prefixes ready for synchronization and each holds the proper authorization.

Static contexts are defined in Table 4 following standard lines. We use $\cdot_1$ and $\cdot_2$ notation to avoid ambiguity, i.e., when $\mathcal{C}[\cdot_1,\cdot_2] = \mathcal{C}'[\cdot_1] \mid \mathcal{C}''[\cdot_2]$ then $\mathcal{C}[P,Q] = \mathcal{C}'[P] \mid \mathcal{C}''[Q]$. Note that in Table 4 there is no case for name restriction construct (νa), which allows to identify specific names and avoid unintended name capture. Remaining cases specify holes can occur in parallel composition and underneath the authorization scope, the only other contexts underneath which processes are deemed active. We omit the symmetric cases for parallel composition since contexts will be used up to structural congruence.

Operator *drift* plays a double role: on the one hand it is defined only when the hole/holes is/are under the scope of the appropriate number of authorizations in the context; on the other hand, when defined, it yields a context obtained from the original

one by removing specific authorizations (so as to capture confinement). In our model, the specific authorizations that are removed for the sake of confinement are the ones nearest to the occurrence of the hole. Operator $drift$ is defined inductively on the structure of contexts by the rules shown in Table 5, both for contexts with one hole (rules prefixed with C-) and for contexts with two holes (rules prefixed with C2-). For the one hole case, $drift(\mathcal{C}[\cdot]; \tilde{a}; \tilde{d})$ takes a context and lists of names $\tilde{a}$ and $\tilde{d}$, in which the same name can appear more than once. The first list carries the names of authorizations that are to be removed and the second carries the names of authorizations that have already been removed. Similarly, $drift(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e})$ takes a context with two holes, two lists of names $\tilde{a}$ and $\tilde{b}$ representing the names of authorizations which are to be removed and two list of names $\tilde{d}$ and $\tilde{e}$ representing names of authorizations already removed. Lists $\tilde{a}$ and $\tilde{d}$ refer to the $\cdot_1$ hole while $\tilde{b}$ and $\tilde{e}$ refer to the $\cdot_2$ hole.

We briefly comment on the rules shown in Table 5, reading from conclusion to premises. The rule where the authorization is removed from the context (C-REM) specifies that the name is passed from the "to be removed" list to the "has been removed" list, where we use $\tilde{a}, c$ to refer to the list that contains $\tilde{a}$ and c and likewise for $\tilde{d}, c$. In the rule where the authorization is preserved in the context (C-SKIP), we check if the name is not in the second list, hence only authorizations that were not already removed proceeding towards the hole can be preserved. This ensures the removed authorizations are the ones nearest to the hole. The rule for parallel composition (C-PAR) is straightforward and the base rule (C-END) is defined only if the first list is empty. This implies that the operator is defined only when all authorizations from the first list are actually removed from the context up to the point the hole is reached. Note that when defining the operator for some context $\mathcal{C}[\cdot]$ and some list of names $\tilde{a}$ that are to be removed from the context, no authorizations have been removed and the respective list $\tilde{d}$ is empty. For example,

$$
\begin{array}{l}
drift((a)\cdot; a; \emptyset) = \cdot \\
drift((a)((a)\ \cdot\ \mid R); a; \emptyset) = (a)(\cdot \mid R) \\
drift(\cdot; a; \emptyset) \text{ is undefined} \\
drift((a)(b)\cdot; a, b; \emptyset) = \cdot \\
drift((a)\cdot; a, b; \emptyset) \text{ is undefined} \\
drift((a)\ \cdot\ \mid (b)0; a, b; \emptyset) \text{ is undefined.}
\end{array}
$$

Rule (C2-SPL) describes the case for two contexts with one hole each, in which case the respective operator is used to obtain the resulting context, considering the name lists $\tilde{a}$ and $\tilde{d}$ for the context on the left hand side and $\tilde{b}$ and $\tilde{e}$ for the context on the right hand side. The remaining rules follow exactly the same lines of the ones for contexts with one hole, where authorization removal addresses left and right hole in a dedicated way. For example,

$$
\begin{array}{l}
drift((b)(a)(a)(\ \cdot_1 \mid \cdot_2\ ); a, b; a; \emptyset; \emptyset) = \ \cdot_1 \mid \cdot_2 \\
drift((b)(a)(\ \cdot_1 \mid (a)\cdot_2\ ); a, b; a; \emptyset; \emptyset) = \ \cdot_1 \mid \cdot_2 \\
drift((a)(b)\ \cdot_1\ \mid (a)\cdot_2; a, b; a; \emptyset; \emptyset) = \ \cdot_1 \mid \cdot_2 \\
drift((b)(\ \cdot_1 \mid (a)(a)\cdot_2\ ); a, b; a; \emptyset; \emptyset) \text{ is undefined.}
\end{array}
$$

The derivation for the case of two holes relies on the derivations for the cases of one hole and is possible only if the axioms for empty contexts hold. Thus, the operator is undefined if the proper authorizations are lacking. As before, lists $\tilde{d}$ and $\tilde{e}$ are

$$\text{(R-COMM)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];a;a)=\mathcal{C}'[\cdot_1,\cdot_2]}{\mathcal{C}[a!b.P,a?x.Q]\to\mathcal{C}'[(a)P,(a)Q\{b/x\}]} \qquad \text{(R-AUTH)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];a,b;a)=\mathcal{C}'[\cdot_1,\cdot_2]}{\mathcal{C}[a\langle b\rangle.P,a(b).Q]\to\mathcal{C}'[(a)P,(a)(b)Q)]}$$

$$\text{(R-STRU)}\quad \frac{P\equiv P'\to Q'\equiv Q}{P\to Q} \qquad \text{(R-NEWC)}\quad \frac{P\to Q}{(\nu a)P\to(\nu a)Q}$$

Table 6. Reduction rules.

used only internally by the operator and we abbreviate $\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b};\emptyset;\emptyset)$ with $\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\tilde{a};\tilde{b})$.

We may now present the reduction rules, shown in Table 6. Rule (R-COMM) states that two processes can synchronize on name a, passing name b from emitter to receiver, only if both processes are under the scope of, at least one per each process, authorizations for name a. The yielded process considers the context where the two authorizations have been removed by the drift operator, and specifies the *confined* authorizations for a which scope only over the continuations of the communication prefixes P and Q. Analogously to (R-COMM), rule (R-AUTH) states that two process can exchange authorization (b) on a name a only if the first process is under the scope of authorization b and if both processes are authorized to perform an action on name a. As before, the yielded process considers the context where the authorizations have been removed by the drift operator. The authorization for b is removed for the delegating process and confined to the receiving process so as to model the exchange. Finally, the rule (R-STRU) closes reduction under structural congruence, and rule (R-NEWC) closes reduction under the restriction construct (νa). Note there are no rules that close reduction under parallel composition and authorization scoping, as these constructs are already addressed by the contexts in (R-COMM) and (R-AUTH). There is also no rule dedicated to replicated input since, thanks to structural congruence rule (SC-REP), a single copy of replicated process may be distinguished and take a part in a synchronization captured by (R-COMM).

As mentioned previously, we may encode a general form of replication, that we may write as $!P$, as

$$(\nu a)((a)a!a.0 \mid !(a)a?x.(P \mid a!a.0))$$

where $a \notin \mathsf{fn}(P)$, since in two steps it reduces to

$$P \mid (\nu a)((a)(P \mid a!a.0) \mid !(a)a?x.(P \mid a!a.0)).$$

where both a copy of P and the original process are active. Notice that since $a \notin \mathsf{fn}(P)$ the process $((a)(P \mid a!a.0)$ does not require any further authorizations on a.

Synchronizations in our model are tightly coupled with the notion of authorization, in the sense that in the absence of the proper authorizations the synchronizations cannot take place. We characterize such undesired configurations, referred to as *error* processes, by identifying the redexes singled-out in the reduction semantics which are stuck due to the lack of the necessary authorizations. This is the case when the premise of the reduction rules does not hold, hence when the drift operator is not defined.

Definition 1 (Error). *Process P is an error if $P \equiv (\nu\tilde{c})\mathcal{C}[\alpha_a.Q, \alpha'_a.R]$ and*

1. *$\alpha_a = a!b$, $\alpha'_a = a?x$ and $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ is undefined, or*
2. *$\alpha_a = a\langle b\rangle$, $\alpha'_a = a(b)$ and $drift(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ is undefined.*

Notice that the definition is aligned with that of reduction, where structural congruence is used to identify a configuration that directly matches one of the redexes given for reduction, but where the respective application of $drift$ is undefined.

We may show that (fully authorized) τ transitions match reductions.

Theorem 1 (Harmony). *$P \to Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.*

Both presentations of the semantics inform on the particular nature of authorizations in our model. On the one hand, the labeled transition system is more explicit in what concerns authorization manipulation; on the other hand, the reduction semantics identifies the pairs of processes that may synchronize, and is therefore useful to identify communication errors in a direct way and more amenable to use in the proofs of our typing results.

3 Type System

In this section, we present a type discipline that allows to statically identify *safe* processes, hence that do not exhibit unauthorized actions (cf. Definition 1). As mentioned before, our typing analysis addresses configurations where authorizations can be granted contextually. Before presenting the type language, we introduce auxiliary notions that cope with name generation, namely *symbol* annotations and *well-formedness*.

Since the process model includes name restrictions and types contain name identities, we require a symbolic handling of bound names when they are included in type specifications. Without loss of generality, we refine the process model for the purpose of the type analysis adding an explicit symbolic representation of name restrictions. In this way, we avoid a more involved handling of bound names in typing environments.

Formally, we introduce a countable set of symbols $\mathcal{S}$ ranged over by $\mathbf{r}, \mathbf{s}, \mathbf{t}, \ldots$, disjoint with the set of names $\mathcal{N}$, and symbol κ not in $\mathcal{N} \cup \mathcal{S}$. Also, in order to introduce a unique association of restricted names and symbols, we refine the syntax of the name creation construct $(\nu a)P$ in two possible ways, namely $(\nu a : \mathbf{r})P$ and $(\nu a : \kappa)P$, decorated with symbol from $\mathcal{S}$ or with symbol κ, respectively. We use $\mathsf{sym}(P)$ to denote a set of all symbols from $\mathcal{S}$ in process P. Names associated with symbols from $\mathcal{S}$ may be provided contextual authorizations, while names associated with symbol κ may not.

For the purpose of this section we adopt the reduction semantics, adapted here considering refined definitions of structural congruence and reduction. In particular for structural congruence, we omit axiom (SC-RES-INACT) $(\nu a)0 \equiv 0$ and we decorate name restriction accordingly in rules (SC-RES-SWAP), (SC-RES-EXTR) and (SC-SCOPE-AUTH)—e.g., $P \mid (\nu a : \mathbf{r})Q \equiv (\nu a : \mathbf{r})(P \mid Q)$ and $P \mid (\nu a : \kappa)Q \equiv (\nu a : \kappa)(P \mid Q)$ keeping the condition $a \notin \mathsf{fn}(P)$. We remark that the omission of (SC-RES-INACT) is used in process models where name restriction is decorated with typing information (cf. [1]).

The annotations with symbols from $\mathcal{S}$ allow to yield a unique identification of the respective restricted names. The *well-formed* processes we are interested in have unique

(T-STOP)
$\Delta \vdash_\rho 0$

(T-PAR)
$$\frac{\Delta \vdash_{\rho_1} P_1 \quad \Delta \vdash_{\rho_2} P_2 \quad \mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset}{\Delta \vdash_{\rho_1 \uplus \rho_2} P_1 \mid P_2}$$

(T-AUTH)
$$\frac{\Delta \vdash_{\rho \uplus \{a\}} P}{\Delta \vdash_\rho (a)P}$$

(T-NEW)
$$\frac{\Delta, a : \{a\}(T) \vdash_\rho P \quad \Delta' = \Delta\{\mathbf{r}/a\} \quad \mathbf{r} \notin \mathsf{sym}(P) \quad a \notin \rho, names(T)}{\Delta' \vdash_\rho (\nu a : \mathbf{r})P}$$

(T-NEW-REP)
$$\frac{\Delta, a : \kappa(T) \vdash_\rho P \quad a \notin \rho, names(T, \Delta)}{\Delta \vdash_\rho (\nu a : \kappa)P}$$

(T-OUT)
$$\frac{\Delta \vdash_\rho P \quad \Delta(a) = \gamma(\gamma'(T)) \quad \Delta(b) = \gamma''(T) \quad \gamma'' \subseteq \gamma' \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_\rho a!b.P}$$

(T-IN)
$$\frac{\Delta, x : T \vdash_\rho P \quad \Delta(a) = \gamma(T) \quad x \notin \rho, names(\Delta) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_\rho a?x.P}$$

(T-REP-IN)
$$\frac{\Delta, x : T \vdash_{\{a\}} P \quad \Delta(a) = \gamma(T) \quad x \notin \rho, names(\Delta) \quad \mathsf{sym}(P) = \emptyset}{\Delta \vdash_\rho !(a)a?x.P}$$

(T-DELEG)
$$\frac{\Delta \vdash_\rho P \quad \Delta(a) = \gamma(T) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_{\rho \uplus \{b\}} a\langle b\rangle.P}$$

(T-RECEP)
$$\frac{\Delta \vdash_{\rho \uplus \{b\}} P \quad \Delta(a) = \gamma(T) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_\rho a(b).P}$$

Table 7. Typing rules.

occurrences of symbols from $\mathcal{S}$ and no occurrences of symbols from $\mathcal{S}$ in the body of replicated inputs. We may show that well-formedness is preserved under (adapted) structural congruence and reduction, and that typable processes are well-formed.

We may now introduce the type language, which syntax is given by $\gamma ::= \varphi \mid \kappa$ and $T ::= \gamma(T) \mid \emptyset$. Types inform on safe instantiations of names that are subject to contextual authorizations, when γ is a set of names from $\mathcal{N}$ and of symbols from $\mathcal{S}$ (denoted φ), and when names are not subject to contextual authorizations (κ). In $\gamma(T)$ type T characterizes the names that can be communicated in the channel, and type $\emptyset$ represents names that cannot be used for communication. A typing environment Δ is a set of typing assumptions of the form $a : T$. We denote by $names(T)$ the set of names that occur in T and by $names(\Delta)$ the set of names that occur in all entries of Δ.

The type system is defined by the rules given in Table 7. A typing judgment $\Delta \vdash_\rho P$ states that P uses channels as prescribed by Δ and that P can only be placed in contexts that provide the authorizations given in ρ, which is a multiset of names (from

$\mathcal{N}$, including their multiplicities). The use of a multiset can be motivated by considering process $a!b.0 \mid a?x.0$ that can be typed as $a : \{a\}(\{b\}(\emptyset)) \vdash_\rho a!b.0 \mid a?x.0$ where necessarily ρ contains $\{a, a\}$ specifying that the process can only be placed in contexts that offer two authorizations on name a (one is required per communicating prefix).

We briefly discuss the typing rules. Rule (T-STOP) says that the inactive process is typable using any Δ and ρ. Rule (T-PAR) states that if processes P_1 and P_2 are typed under the same environment Δ, then $P_1 \mid P_2$ is typed under Δ as well. Also if P_1 and P_2 own enough authorizations when contexts provide authorizations ρ_1 and ρ_2, respectively, then $P_1 \mid P_2$ will have enough authorizations when the context provides the sum of authorizations from ρ_1 and ρ_2. By $\rho_1 \uplus \rho_2$ we represent the addition operation for multisets which sums the frequencies of the elements. The condition $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$ is necessary to ensure well-formedness. Rule (T-AUTH) says that $(a)P$ and P are typed under the same environment Δ, due to the fact that scoping is a non-binding construct. If P owns enough authorizations when the context provides $\rho \uplus \{a\}$, then $(a)P$ can be safely placed in a context that provides ρ.

Rule (T-NEW) states that if P is typable under an environment that contains an entry for a, then $(\nu a : \mathbf{r})P$ is typed under the environment removing the entry for a and where each occurrence of name a in Δ is substituted by the symbol $\mathbf{r}$. By $\Delta\{\mathbf{r}/a\}$ we denote the environment obtained by replacing occurrences of a by $\mathbf{r}$ in every assumption in Δ, hence in every type, excluding when a has an entry in Δ. Condition $\mathbf{r} \notin \mathsf{sym}(P)$ is necessary to ensure well-formedness and $a \notin \rho, names(T)$ says that the context cannot provide authorization for name a and ensures consistency of the typing assumption.

The symbolic representation of a bound name in the typing environment enables us to avoid the case when a restricted (unforgeable) name could be sent to a process that expects to provide a contextual authorization for the received name. For example consider process $(a)(d)a?x.x!c.0 \mid (\nu b : \mathbf{r})(a)a!b.0$ where a contextual authorization for d is specified, a configuration excluded by our type analysis since the assumption for the type of channel a carries a symbol (e.g., $a : \{a\}(\{\mathbf{r}\}(\emptyset))$) for which no contextual authorizations can be provided. Notice that the typing of the process in the scope of the restriction uniformly handles the name, which leaves open the possibility of considering contextual authorizations for the name within the scope of the restriction.

In rule (T-NEW-REP) the difference with respect to rule (T-NEW) is that no substitution is performed, since the environment must already refer to symbol κ in whatever pertains to the restricted name. For example to type process $(\nu b : \kappa)(a)a!b.0$ the type of a must be $\gamma(\kappa(T))$, for some γ and T, where κ identifies the names communicated in a are never subject to contextual authorizations.

Rule (T-OUT) considers that P is typed under an environment where types of names a and b are such that all possible replacements for name b (specified in γ'') are safe to be communicated along name a (which is formalized by $\gamma'' \subseteq \gamma'$, where γ' is given in the type of a), and also that T (the carried type of b) matches the specification given for a. In such case, the process $a!b.P$ is typed under the same environment. There are two possibilities to ensure name a is authorized, namely the context may provide directly the authorization for name a or it may provide authorizations for all replacements of name a, formalized as $a \notin \rho \Rightarrow \gamma \subseteq \rho$. Notice this latter option is crucial to address

contextual authorizations and that in such case γ does not contain symbols, since ρ by definition does not. Rule (T-IN) follows similar principles.

Rule (T-REP-IN) considers the continuation is typed with an assumption for the input bound name x and that the expected context provides authorizations only for name a. In such a case, the replicated input is typable considering the environment obtained by removing the entry for x, which must match the carried type of a, provided that $x \notin \rho, names(\Delta)$ since it is bound to this process. Also, the fact that process P does not contain any symbol from $\mathcal{S}$ is necessary to ensure the unique association of symbols and names when copies of the replicated process are activated (see the discussion on example (4) at the end of this Section). In that case, process $!(a)a?x.P$ can be placed in any context that conforms with Δ and provides (any) ρ.

Rules (T-DELEG) and (T-RECEP) consider the typing environment is the same in premises and conclusion. The handling of the subject of the communication (a) is similar to, e.g., rule (T-OUT) and the way in which the authorization is addressed in rule (T-RECEP) follows the lines of rule (T-AUTH). In rule (T-DELEG), the authorization for b is added to the ones expected from the context. Notice that in such way no contextual authorizations can be provided for delegation, but the generalization is direct.

For the sake of presenting the results, we say process P is well-typed if $\Delta \vdash_\emptyset P$ and Δ only contains assumptions of the form $a : \{a\}(T)$ or $a : \kappa(T)$. At top level the typing assumptions address the free names of the process, which are not subject to instantiation. Free names are either characterized by $a : \{a\}(T)$ which says that a is the (final) instantiation, or by $a : \kappa(T)$ which says that a cannot be granted contextual authorizations. For example, process $(a)a!b.0 \mid (a)(b)a?x.x!c.0$ is typable under the assumption that name b has type $\{b\}(\{c\}(\emptyset))$, while it is not typable under the assumption $\kappa(\{c\}(\emptyset))$. The fact that no authorizations are provided by the context ($\rho = \emptyset$) means that the process P is self-sufficient in terms of authorizations.

We may now present our results, starting by mentioning some fundamental properties. We may show that typing derivations enjoy standard properties (Weakening and Strengthening) and that typing is preserved under structural congruence (Subject Congruence). As usual, to prove typing is preserved under reduction we may also show an auxiliary result that addresses name substitution (cf. Lemma 3.1 [13]). Noticeably, even though subtyping is not present, the result uses an inclusion principle that already hints on substitutability. Our first main result says that typing is preserved under reduction.

Theorem 2 (Subject Reduction). *If P is well-typed, $\Delta \vdash_\emptyset P$ and $P \rightarrow Q$ then $\Delta \vdash_\emptyset Q$.*

Not surprisingly, since errors involve redexes, the proof of Theorem 2 is intertwined with the proof of the error absence property, given in our second main result which captures the soundness of our typing analysis.

Proposition 1 (Type Soundness). *If P is well-typed then P is not an error.*

As usual, the combination of Theorem 2 and Proposition 1 yields type safety, stating that any configuration reachable from a well-typed one is not an error. Hence type safety ensures that well-typed processes will never incur in a configuration where the necessary authorizations to carry out the communications are lacking.

We extend the example shown in the Introduction to illustrate the typing rules:

$$(alice)\,alice!exam.0 \mid (exam)(minitest)(alice)\,alice?x.x!task.0 \tag{1}$$

Considering assumption $alice : \{alice\}(\{exam, minitest\}(\emptyset))$, and assuming that the type of $exam$ is $\{exam\}(\emptyset)$, by rule (T-OUT) we conclude that it is safe to send name $exam$ along $alice$, since the (only) instantiation of $exam$ is contained in the carried type of $alice$. Now let us place process (1) in a context restricting $exam$:

$$(\nu exam : \mathbf{r})((alice)\,alice!exam.0 \mid (exam)(minitest)(alice)\,alice?x.x!task.0) \tag{2}$$

To type this process, the assumption for $alice$ specifies type $\{alice\}(\{\mathbf{r}, minitest\}(\emptyset))$, representing that in $alice$ a restricted name can be communicated. Hence, the process shown in (2) cannot be composed with others that rely on contextual authorizations for names exchanged in $alice$, and can only be composed with processes like $(alice)\,alice?x.(x)x!task$ or that use authorization delegation. Now consider process:

$$!(license)\,license?x.(\nu exam : \mathbf{r})((x)x!exam.0 \mid (x)(exam)x?y.y!task.0) \tag{3}$$

that models a server that receives a name and afterwards is capable of both receiving (on the lhs) and sending (a fresh name, on the rhs) along the received name. Our typing analysis excludes this process since it specifies a symbol ($\mathbf{r}$) in the body of a replicated input. In fact, the process may incur in an error, as receiving $alice$ twice leads to:

$$\begin{array}{l}(\nu exam_1 : \mathbf{r})((alice)\,alice!exam_1.0 \mid (exam_1)(alice)\,alice?y.y!task.0)\\ \mid (\nu exam_2 : \mathbf{r})((alice)\,alice!exam_2.0 \mid (exam_2)(alice)\,alice?y.y!task.0).\end{array} \tag{4}$$

where two copies of the replicated process are active in parallel, and where two different restricted names can be sent on $alice$, hence the error is reached when the contextual authorization does not match the received name (e.g., $(exam_2)(alice)\,exam_1!task.0$).

In order to address name generation within replicated input, we use distinguished symbol κ that types channels that are never subject to contextual authorizations, even within the restriction scope. Hence replacing the $\mathbf{r}$ annotation by κ in the process shown in (3) does not yield it typable, since a contextual authorization is expected for name $exam$ and may lead to an error like before. However, process:

$$!(license)\,license?x.(\nu exam : \kappa)(alice)\,alice!exam.0 \tag{5}$$

is typable, hence may be composed with contexts compliant with $alice : \{alice\}(\kappa(T))$, i.e., that do not rely on contextual authorizations for names received on $alice$.

4 Concluding remarks

In the literature, we find a plethora of techniques that address resource usage control, ranging from locks that guarantee mutual exclusion in critical code blocks to communication protocols (e.g., token ring). Several typing disciplines have been developed to ensure proper resource usage, such as [5, 10, 15], where capabilities are specified in the

type language, not as a first class entity in the model. Therefore in such approaches it is not possible to separate resource and capability like we do. We distinguish an approach that considers accounting [5], in the sense that the types specify the number of messages that may be exchanged, therefore related to the accounting presented here.

We also find proposals of models that include capabilities as first class entities, addressing usage of channels and of resources as communication objects, such as [4, 9, 12, 16]. More specifically, constructs for restricting (hiding and filtering) the behaviors allowed on channels [9, 16], usage specification in a (binding) name scope construct [12], and authorization scopes for resources based on given access policies [4]. We distinguish our approach from [9, 16] since the proposed constructs are static and are not able to capture our notion of a floating resource capability. As for [12], the usage specification directly embedded in the model resembles a type and is given in a binding scoping construct, which contrasts with our non-binding authorization scoping. Also in [4] the provided detailed usage policies are associated to the authorization scopes for resources. In both [4, 12] the models seem less adequate to capture our notion of floating authorizations as access is granted explicitly and controlled via the usage/policy specification, and for instance our notion of confinement does not seem to be directly representable.

We have presented a model of floating authorizations, a notion we believe is unexplored in the existing literature. We based our development on previous work [8] where a certain form of inconsistency when handling the authorization granting in delegation was identified, while in this work granting is handled consistently thanks to the interpretation of accounting. We remark that adding choice to this work can be carried out in expected lines. More interesting would be to consider non-consumptive authorizations, that return to their original scope after (complete) use. We also presented a typing analysis that addresses contextual authorizations, which we also believe is unexplored in the literature in the form we present it here. Our typing rules induce a decidable type-checking procedure, since rules are syntax directed, provided as usual that a (carried) type annotation is added to name restrictions. Albeit the work presented here is clearly of a theoretical nature, we hope the principles developed here may be conveyed to, e.g., the licensing domain where we have identified related patents [2, 3, 6] for the purpose of certifying license usage.

A notion of substitutability naturally arises in our typing analysis and we leave to future work a detailed investigation of a subtyping relation that captures such notion. We believe our approach can be extended by considering some form of usage specifications like the ones mentioned above [4, 12], by associating to each authorization scoping more precise capabilities in the form of behavioral types [11]. This would also allow us to generalize our approach addressing certain forms of infinite behavior, namely considering recursion together with linearity constraints that ensure race freedom. It would also be interesting to resort to refinement types [7] to carry out our typing analysis, given that our types can be seen to some extent as refinements on the domain of names.

Acknowledgments We thank anonymous reviewers for useful remarks and suggestions. This work has been partially supported by the Ministry of Education and Science of the Republic of Serbia, project ON174026, and EU COST Action IC1405 (Reversible Computation).

References

1. Acciai, L., Boreale, M.: Spatial and behavioral types in the pi-calculus. Inf. Comput. **208**(10), 1118–1153 (2010). https://doi.org/10.1016/j.ic.2009.10.011
2. Armstrong, W.J., Nayar, N., Stamschror, K.P.: Management of a concurrent use license in a logically-partitioned computer (2005), US Patent 6,959,291
3. Baratti, P., Squartini, P.: License management system (2003), US Patent 6,574,612
4. Bodei, C., Dinh, V.D., Ferrari, G.L.: Checking global usage of resources handled with local policies. Sci. Comput. Program. **133**, 20–50 (2017). https://doi.org/10.1016/j.scico.2016.06.005
5. Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session types. CoRR **abs/1712.08310** (2017), http://arxiv.org/abs/1712.08310
6. Ferris, J.M., Riveros, G.E.: Offering additional license terms during conversion of standard software licenses for use in cloud computing environments (2015), US Patent 9,053,472
7. Freeman, T.S., Pfenning, F.: Refinement types for ML. In: Wise, D.S. (ed.) Proceedings of the PLDI 1991. pp. 268–277. ACM (1991). https://doi.org/10.1145/113445.113468
8. Ghilezan, S., Jakšić, S., Pantović, J., Pérez, J.A., Vieira, H.T.: Dynamic role authorization in multiparty conversations. Formal Asp. Comput. **28**(4), 643–667 (2016). https://doi.org/10.1007/s00165-016-0363-5
9. Giunti, M., Palamidessi, C., Valencia, F.D.: Hide and new in the pi-calculus. In: Proceedings of EXPRESS/SOS 2012. EPTCS, vol. 89, pp. 65–79 (2012). https://doi.org/10.4204/EPTCS.89.6
10. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. J. Log. Algebr. Program. **78**(8), 665–689 (2009). https://doi.org/10.1016/j.jlap.2008.12.001
11. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016). https://doi.org/10.1145/2873052
12. Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the p-calculus. Logical Methods in Computer Science **2**(3) (2006). https://doi.org/10.2168/LMCS-2(3:4)2006
13. Pantovic, J., Prokic, I., Vieira, H.T.: A calculus for modeling floating authorizations. CoRR **abs/1802.05863** (2018), https://arxiv.org/abs/1802.05863
14. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
15. Swamy, N., Chen, J., Chugh, R.: Enforcing stateful authorization and information flow policies in fine. In: Proceedings of ESOP 2010. LNCS, vol. 6012, pp. 529–549. Springer (2010). https://doi.org/10.1007/978-3-642-11957-6_28
16. Vivas, J., Yoshida, N.: Dynamic channel screening in the higher order pi-calculus. Electr. Notes Theor. Comput. Sci. **66**(3), 170–184 (2002). https://doi.org/10.1016/S1571-0661(04)80421-3