UNIVERSITY OF NOVI SAD

FACULTY OF TECHNICAL SCIENCES
NOVI SAD

Nikola Luburić

Integration of Software Security Design Analysis to the Agile Development Process

- Ph. D. Thesis -

Supervisor
Goran Sladić, PhD, associate professor

Novi Sad, 2019.

Nikola Luburić: *Integration of Software Security Design Analysis to the Agile Development Process*

SERBIAN TITLE:
Integracija bezbednosne analize dizajna softvera u proces agilnog razvoja

SUPERVISOR:
Goran Sladić, PhD, associate professor

LOCATION:
Novi Sad, Serbia

DATE:
September 2019

UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA**
21000 NOVI SAD, Trg Dositeja Obradoviće 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, **RBR**:	
Identifikacioni broj, **IBR**:	
Tip dokumentacije, **TD**:	Monografska dokumentacija
Tip zapisa, **TZ**:	Tekstualni štampani materijal
Vrsta rada, **VR**:	Doktorska disertacija
Autor, **AU**:	Nikola Luburić
Mentor, **MH**:	dr Goran Sladić, vanredni profesor
Naslov rada, **NR**:	Integracija bezbednosne analize dizajna softvera u proces agilnog razvoja
Jezik publikacije, **JP**:	engleski
Jezik izvoda, **JI**:	srpski
Zemlja publikacije, **ZP**:	Srbija
Uže geografsko područije, **UGP**:	Vojvodina
Godina, **GO**:	2019
Izdavač, **IZ**:	Fakultet tehničkih nauka
Mesto i adresa, **MA**:	Trg Dositeja Obradovića 6, 21000 Novi Sad
Fizički opis rada, **FO**: (poglavlja/strana /citata/tabela/slika/grafika/priloga)	6/159/154/13/13/0/0
Naučna oblast, **NO**:	Elektrotehničko i računarsko inženjerstvo
Naučna disciplina, **ND**:	Bezbednost softvera
Predmetna odrednica/Ključne reči, **PO**:	bezbednosna analiza dizajna, modelovanje pretnji, razvoj bezbednog softvera, životni ciklus razvoja bezbednosti, bezbednosna ekspertiza, bezbednost softvera
UDK	
Čuva se, **ČU**:	Biblioteka Fakulteta tehničkih nauka, Trg Dositeraj Obradovića 6, 21000 Novi Sad
Važna napomena, **VN**:	

UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA**
21000 NOVI SAD, Trg Dositeja Obradoviće 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Izvod, **IZ**:		U sklopu disertacije izvršeno je istraživanje u oblasti razvoja bezbednog softvera. Razvijene su dve metode koje zajedno omogućuju integraciju bezbednosne analize dizajna softvera u proces agilnog razvoja. Prvi metod predstavlja radni okvir za konstruisanje radionica čija svrha je obuka inženjera softvera kako da sprovode bezbednosnu analizu dizajna. Drugi metod je proces koji proširuje metod bezbednosne analize dizajna kako bi podržao bolju integraciju spram potreba organizacije. Prvi metod je evaluiran kroz kontrolisan eksperiment, dok je drugi metod evaluiran upotrebom komparativne analize i analize studija slučaja, gde je proces implementiran u kontekstu dve organizacije koje se bave razvojem softvera.	
Datum prihvatanja teme, **DP**:		11.07.2019.	
Datum odbrane, **DO**:			
Članovi komisije, **KO**:	Predsednik:	dr Branko Milosavljević, redovni profesor, FTN, Novi Sad	
	Član:	dr Silvia Gilezan, redovni profesor, FTN, Novi Sad	
	Član:	dr Gordana Milosavljević, vanredni profesor, FTN, Novi Sad	Potpis mentora
	Član:	dr Žarko Stanisavljević, docent, ETF, Beograd	
	Član, mentor:	dr Goran Sladić, vanredni profesor, FTN, Novi Sad	

UNIVERSITY OF NOVI SAD • **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monograph documentation
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Ph.D. thesis
Author, **AU**:	Nikola Luburić
Mentor, **MN**:	Goran Sladić, Ph.D., Associate Professor
Title, **TI**:	Integration of Software Security Design Analysis to the Agile Development Process
Language of text, **LT**:	English
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2019
Publisher, **PB**:	Faculty of Technical Sciences
Publication place, **PP**:	Trg Dositeja Obradovića 6, 21000 Novi Sad
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/	6/159/154/13/13/0/0
Scientific field, **SF**:	Electrical engineering and computing
Scientific discipline, **SD**:	Software security
Subject/Key words, **S/KW**:	security design analysis, threat modeling, secure software engineering, security development lifecycle, security expertise,
UC	
Holding data, **HD**:	Library of Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000
Note, **N**:	

UNIVERSITY OF NOVI SAD • **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Abstract, **AB**:		This thesis presents research in the field of secure software engineering. Two methods are developed that, when combined, facilitate the integration of software security design analysis into the agile development workflow. The first method is a training framework for creating workshops aimed at teaching software engineers on how to perform security design analysis. The second method is a process that expands on the security design analysis method to facilitate better integration with the needs of the organization. The first method is evaluated through a controlled experiment, while the second method is evaluated through comparative analysis and case study analysis, where the process is tailored and implemented for two different software vendors.	
Accepted by the Scientific Board on, **ASB**:		11.07.2019.	
Defended on, **DE**:			
Defended Board, **DB**:	President:	Branko Milosavljević, PhD, Full Professor, FTN, Novi Sad	
	Member:	Silvia Gilezan, PhD, Full Professor, FTN, Novi Sad	
	Member:	Gordana Milosavljević, PhD, Associate Professor, FTN, Novi Sad	Menthor's signature
	Member:	Žarko Stanisavljević, PhD, Assistant Professor, ETF, Belgrade	
	Member, Mentor:	Goran Sladić, PhD, Associate Professor, FTN, Novi Sad	

Acknowledgements

First of all, I would like to express my sincere gratitude to my mentors Professor Jovanka Pantović and Professor Hugo Torres Vieira, for their unselfish help and experienced guidance. Before starting my PhD studies I had no idea what doing research means. They introduced me to the wonderful world of formal methods and shown me how far one needs to go in order to extract the juice out of his work. Above all, I thank them for always encouraging me to "keep up the good work", and for demonstrating what it takes to be a true scientist and a good person.

To my elementary and high school teachers, and my professors at the university, thank you for making me prepared for doing the Ph.D.

I would like to thank my colleagues at the Chair of Mathematics at Faculty of Technical Sciences for letting me be part of the family and for sharing all the good and the bad that our job carries.

To all of my friends, for making my life happier.

To my family, I owe the most. I thank my "in-law" family Radovan, Jasna, and Maja, for their great support. To my sister Draginja, her husband Bojan and my nieces Jelisaveta and Jovana, to my older brother Mladen and his wife Nina, and to my younger brother Aleksandar, thank you for being a perfect family. To my parents Jovanka and Siniša, for their endless love. To my wife Sanja, and our daughters Tamara and Lenka, for their patience, support, and unquestionable love, thank you for being my world.

Abstract

Distributed software systems have changed the way people communicate, learn and run businesses: almost all aspects of human life have become connected to the internet. The system of interconnected computing devices has numerous positive impacts on everyday life, however, it also raises some concerns, among which are security, accessibility and availability issues.

This thesis investigates problems of formal, mathematically based, representation and analysis of controlled usage and sharing of resources in distributed software systems. The thesis is organized into four chapters. The first chapter provides motivation for our work, and the last concludes the thesis. The second and the third chapters are the core of the thesis, the former addresses controlling information passing and the latter addresses controlling usages of resources.

The second chapter presents a model for confidential name passing, called Confidential π-calculus, abbreviated C_π. This model is a simple fragment of the π-calculus that disables information forwarding directly at the syntax level. We provide an initial investigation of the model by presenting some of its properties, such as the non-forwarding property and the creation of closed domains for channels. We also present examples showing that C_π can be used to model restricted information passing, authentication, closed and open-ended groups. We present an encoding of the (sum-free) π-calculus in C_π and we prove the correctness of the encoding via an operational correspondence result.

The third chapter presents a model of floating authorizations. Our process model introduces floating authorizations as first-class entities, encompassing dimensions of accounting, domain, and delegation. We exploit the language of an already existing process algebra for authorizations, and we adopt a different semantic interpretation so as to capture accounting. We define the semantics of our model in two equivalent ways, using a labeled transition system and a reduction relation. We define error processes as undesired configurations that cannot evolve due to lacking authorizations. The thesis also provides a preliminary investigation of the behavioral semantics of our authorization model, showing some fundamental properties and also informing on the specific nature of floating authorizations.

In the third chapter, we also present a typing discipline that allows to statically single out processes that are not errors and that never evolve into errors, addressing configurations where authorization assignment is not statically prescribed in the system specification. We also develop a refinement of our typing discipline to pave the way for a more efficient type-checking procedure. We show an extended example of a scenario that involves the notion of Bring Your Own License, and we exploit this example to provide insight on a possible application of our model in programming language design.

Rezime

Rasprostranjenost distribuiranih softverskih sistema različitih namena promenila je način na koji ljudi komuniciraju, stiču znanja i vode biznise: skoro svi aspekti ljudskog života postali su povezani sa internetom. Ovaj sistem međusobno povezanih računarskih instanci napravio je veliki pozitivan uticaj na svakodnevni život, od brze i jednostavne komunikacije putem društvenih mreža i računarskih platformi dostupnih putem interneta, do distribuiranih sistema za plaćanja i kriptovaluta zasnovanih na blockchain tehnologijama. Međutim, deljenje informacija, prava pristupa bazama podataka i prava pristupa računarskim platformama, kao i deljenje drugih resursa otvara i nove probleme, među kojima su pitanja bezbednosti, pristupačnosti i dostupnosti. Postoji veliki broj primera u kojima su napadači (krakeri) uspeli da zloupotrebe previde programera koji su razvijali sisteme. Jedan takav skoriji primer je i greška koja je omogućila nepravilno generisanje tokena za pristup ličnim profilima na Facebook-u. Tu grešku je za sada nepoznati napadač uspeo da iskoristi da bi došao do ličnih podataka sa skoro 50 miliona naloga [95]. Takvi primeri jasno ukazuju na probleme kontrole deljenja i korišćenja resursa u distribuiranim softverskim sistemima, problemima kojima se ova teza bavi korišćenjem formalnih metoda.

Pouzdanost distribuiranih softverskih sistema može zavisiti od velikog broja faktora i često nije lako čak ni definisati šta se pod pouzdanošću određenog sistema podrazumeva. Jedan od mogućih pristupa kod dizajna i verifikacije takvih sistema je korišćenje formalnih, matematički zasnovanih metoda. Formalne metode predstavljaju tehnike i alate za specifikaciju i verifikaciju kompleksnih (softverskih i hardverskih) sistema zasnovane na matematičkim i logičkim principima. Formalni dizajn obuhvata dve faze: formalnu specifikaciju i verifikaciju. U fazi formalne specifikacije (modeliranja) definiše se sistem koristeći jezik modeliranja, najčešće koristeći preciznu matematičku sintaksu i semantiku. Razvijajući formalnu specifikaciju, uglavnom nastaje i skup teorema koje opisuju osobine tog sistema. U fazi verifikacije, ove teoreme se precizno matematički dokazuju. U konkurentnom računarstvu, neki od poznatih formalnih modela koji se koriste za specifikaciju i verifikaciju osobina sistema su Petrijeve mreže [83, 94], komunicirajući automati sa konačnim brojem stanja [20] i procesni računi [54, 68, 73, 74].

Cilj ove teze je da predstavi dve specifikacije zasnovane na dva procesna računa koje tretiraju neke aspekte bezbednosti i kontrole pristupa u distribuiranim sistemima. Konačni cilj je stvoriti uslove za bolje razumevanje koncepata izučavanih u ovom radu i omogućiti njihovu kasniju ispravnu implementaciju.

Komunikacija putem distribuiranih sistema, ponekad uključujući interakcije sa nepoznatim i nepouzdanim korisnicima, je postala svakodnevna, a u nekim slučajevima čak i nezaobilazna rutina. U mnogim situacijama razmenjena informacija je privatna i zahteva pažljivo rukovanje i korišćenje. Na primer, osetljivi privatni podaci, kao što su broj kreditne kartice ili adresa, moraju biti otkriveni tokom kupovine putem interneta, ali sa druge strane ove informacije ne bi smele biti dalje deljene od strane korisnika ili aplikacije koji prima informaciju. Ovakvi primeri ukazuju na probleme kontrole deljenja informacija u distribuiranim sistemima. Dakle, deljenje informacija sa trećim licima može dovesti do neželjene diseminacije. Čak i u slučajevima kada se

korisnicima generalno može verovati postoji mogućnost previda koji mogu dovesti do zloupotreba.

Problem privatnosti može i mora biti sagledan sa strane tehnologije ali i prava. Jedan od pionira koji su proučavali privatnost iz obe perspektive je pravnik Alan Westin. On je primetio „da će integracija kontrola privatnosti u nove tehnologije zahtevati snažan napor...” [114]. Iako nove tehnologije donose nove pretnje za kontrolu privatnosti, one mogu doneti i nove načine za zaštitu privatnosti [109]. Solove [105] uvodi taksonomiju i navodi četiri vrste narušavanja privatnosti: sakupljanje informacija, invazija, diseminacija i obrađivanje informacija. Nedovoljna kontrola nad deljenjem informacija u distribuiranim sistemima može biti direktno povezana sa diseminacijom.

Solove daje dalju taksonomiju narušavanja privatnosti putem diseminacije, ali sve ove podvrste uglavnom prepoznaju štetu koja može nastati kod otkrivanja i deljenja osetljivih informacija. Komunikacija među učesnicima je centralni aspekt distribuiranih sistema, a kontrola protoka informacija u takvim sistemima često ima svoje poteškoće. Entiteti u takvim sistemima mogu imati različita prava za manipulaciju određenim informacijama. Na primer, korisnik bankovnog računa ima ovlašćenja da koristi broj kartice za plaćanja putem interneta, može povlačiti određena sredstava sa bankovnog računa, itd. Sa druge strane, kod isplate banka može izvršiti uvid u stanje kako bi proverila da li postoji dovoljno sredstava na računu. Ako se za trenutak fokusiramo na kontrolu protoka informacija, možemo uočiti da bi broj kreditne kartice trebalo da može poslati samo korisnik te kartice, ali ne i banka koja prima tu informaciju. To jest, banka ne bi trebalo da ima prava da prosleđuje informaciju u ovom slučaju. Za narušavanje diseminacije informacija, prosleđivanje može biti prepoznato kao jedna od glavnih meta gde kontrola mora biti uspostavljena.

Ako razmatramo prava koja entitet može imati u odnosu na komunikacioni kanal, možemo razlikovati prava na korišćenje kanala za slanje i čitanje, pravo da se kreira novi kanal i da se pošalje jedan njegov kraj drugom korisniku, prava da se prosleđuju primljena imena

kanala, itd. Davanje prava o prosleđivanju imena kanala svim entitetima apriori može kasnije prouzokovati poteškoće oko kontrole diseminacije, jer u tom slučaju kontrola mora da bude sprovedena u celom sistemu.

Posmatrajmo sada jedan jednostavan primer u kom se poverljivo ime kanala *session* šalje od jednog do drugog korisnika, kao što je onaj naveden u odeljku Introduction. U ovom primeru, korisnik *Alice* kreira novi kanal i šalje jedan njegov kraj korisniku *Bob*. Nakon sinhronizazije u kojoj se razmeni ime kanala, ova dva korisnika mogu napraviti privatnu sesiju na kanalu *session*. Međutim, u našem primeru *Bob* odlučuje da prosledi ime kanala *session* nekom trećem korisniku.

U nekim slučajevima može biti čak i poželjno dati prava prosleđivanja imena kanala nekim korisnicima. Na primer, zadaci mogu biti prosleđivani od nadređenog (eng. master) procesa do potčinjenog (eng. slave) procesa, i tada potčinjeni proces može neprimetno da bude uključen u sesiju. U našem primeru, ova situacija može biti posmatrana kao problematična sa tačke gledišta korisnika *Alice*, jer ona i dalje veruje da drugi kraj kanala, koji ona smatra poverljivim, drži *Bob*. Ako je *session* kanal koji je *Alice* kreirala, kojim se može pristupiti nekim njenim poverljivim podacima, i koji je poslat isključivo korisniku *Bob*, možemo reći da *Bob* ne bi trebalo da stekne mogućnost da ga dalje prosleđuje samo zato što je u nekom trenutku primio ime kanala *session*. U svakom slučaju, možemo napraviti razliku između ova dva slanja, jer prvo slanje je izveo korisnik koji je kreirao kanal (*Alice*), a u drugom je kanal zapravo prosleđen od strane učesnika koji je primio kanal (*Bob*).

Nekoliko formalnih modela je do sada predloženo u svrhu opisivanja restrikovanog deljenja imena, kako bi se postiglo da ime može biti razmenjeno samo u okviru unapred definisanog dela sistema. Takav je i model koji uvodi pojam grupe za imena [26] i model koji uvodi pojam skrivanja imena [46]. Međutim, u praksi imamo i slučajeve u kojima ne postoji unapred definisan deo sistema u kom poverljiva informacija može biti razmenjena. Na primer, u prethodnom primeru možemo reći

da *Alice* može u nekom trenutku sama da odluči da pošalje ime kanala *session* drugim učenicima. Generalno, privatne informacije nekada moraju biti deljene i u otvorenim sistemima.

Drugi domen koji ova teza obrađuje je izučavanje kontrole prava pristupa u distribuiranim softverskim sistemima. Za početak, možemo primetiti da kontrola prava pristupa računarskim resursima postaje sve važnija, uprkos sve većoj raspoloživosti takvih resursa. Potreba za kontrolom pristupa može biti motivisana mnogim faktorima, kao što su privatnost, bezbednost i ipak postojanje nekog ograničenja kapaciteta. Primeri ograničenog kapaciteta mogu biti direktno povezani sa fizičkim uređajima, kao što su štampači, mobilni telefoni i procesori, jer svi imaju fizički ograničene mogućnosti. Iako neki virtualni uređaji, kao što su deljena memorijska ćelija i web servis, imaju neograničen potencijal, njihova dostupnost je često ograničena.

Privatnost i bezbednost su neki od centralnih problema koji se pojavljuju kod razvoja distribuiranih sistema. Jedan od razloga je taj što distribuirani sistemi postaju sve više heterogeni i kompleksni, a kontrola prava pristupa u takvim sistemima može biti veoma teška. Opravdanje za ovakve tvrdnje možemo naći skoro svakodnevno, već pomenuti primer greške na Facebook-u je samo jedan u nizu. Takvi primeri su prouzrokovali milionske gubitke kompanija, ali još važnije, sigurnost i privatnost korisnika je u takvim situacijama bila izložena opasnosti. Formalni modeli i verifikacije mogu biti korak bliže ka pouzdanijim distribuiranim softverskim sistemima [21].

Različite metode za kontrolu prava pristupa u distribuiranim softverskim sistemima razvijane su tokom godina. Njihov razvoj pratio je stalne promene u strukturi i veličini sistema. Za male sisteme, i za sisteme sa unapred definisanim brojem učesnika, kontrola prava pristupa resursima obično se postiže korišćenjem lista za kontrolu pristupa (eng. access control lists - ACL). ACL metoda koristi liste sa pravima koje su dodeljene resursima. Pravo pristupa resursu može biti odobreno samo korisniku koji je naveden kao subjekat sa odgovarajućim pravom pristupa na listi datog resursa.

Iako ACL metod daje prirodan način za kontrolu prava pristupa, u velikim sistemima koji su dinamični po pitanju broja i sastava učesnika ovaj metod postaje težak za implementaciju. Razlog za to je što u ACL metodi svaka lista čuva podatke o svakom korisniku individualno, a to može predstavljati veliki trošak pri održavanju sistema. Na primer, posmatrajmo aplikaciju kao što je Facebook, koju koristi preko milijardu korisnika. Imati liste korisnika koji mogu da pristupe resursima, kao što su fotografije ili postovi svakog korisnika, može postati nepraktično.

Upravljanje pristupom na osnovu uloga (eng. role-based access control method - RBAC) [96] je uvedeno kao alternativa ACL metodi. RBAC metoda definiše skup uloga i svakom korisniku dodeljuje se jedna ili više uloga. Na primer, da bi sistem korisniku dozvolio ili odbio pristup fotografiji drugog korisnika na Facebook-u, ne mora se oslanjati na njegov identitet direktno. Praktičnije rešenje je proveriti da li korisnik koji pokušava da pristupi fotografiji ima ulogu „prijatelja" sa vlasnikom fotografije. Pored svih prednosti (i mana) koje RBAC metoda ima u poređenju sa ACL metodom, ona i dalje ima nedostatak da mora postojati centralni mehanizam za izdavanje i proveravanje uloga korisnika.

Upravljanje pristupom na osnovu ključa (eng. capability-based method for access control) [116] je metoda koja je više prilagođena decentralizovanim sistemima. U ovoj metodi, reference koje se ne mogu kopirati kreira i izdaje centralni mehanizam. Jednom izdata referenca ostaje kod korisnika i proverava se samo kada korisnik želi da pristupi resursu. Dakle, u ovoj metodi centralni mehanizam ne mora da drži informacije o kontroli pristupa za svakog korisnika pojedinačno, dovoljno je da proverava validnost referenci (ključeva) samo kada je to potrebno. Takođe, ove reference mogu biti delegirane između dva učesnika, bez potrebe da se o tome obavesti centralni mehanizam za kontolu pristupa.

Još jedan domen koji obuhvata slične principe kao i poslednja navedena metoda za upravljanje pristupom je domen licenci: korisnik može

upotrebiti određenu aplikaciju samo pod uslovom da poseduje odgovarajuću licencu. U ovom domenu takođe možemo naći pojam eksplicitne delegacije. Na primer, korisnik koji želi da uposli aplikaciju na računarskoj platformi dostupnoj putem interneta može delegirati licencu za tu aplikaciju koju već poseduje. Taj pojam poznat je pod nazivom Bring Your Own License [38] (BYOL). Posebna vrsta licenci kao što su licence za konkurentnu upotrebu (eng. concurrent use licenses) nudi dodatnu fleksibilnost kod korišćenja [10]. Kao primer, posmatrajmo jednu kompaniju koja koristi aplikaciju i koja poseduje određeni broj licenci potrebnih za korišćenje te aplikacije. U slučaju licenci za konkurentnu upotrebu, licence mogu biti dostupne svim korisnicima u okviru domena date kompanije, ali broj licenci određuje gornju granicu za broj korisnika koji mogu koristiti aplikaciju u bilo kom trenutku [5].

U ovoj tezi istražujemo probleme formalnog, matematički zasnovanog, modeliranja i analize kontrolisanog korišćenja i deljenja resursa u distribuiranim softverskim sistemima. Teza je organizovana u četiri poglavlja.

Prvo poglavlje daje motivaciju za razvoj modela uvedenih u drugom i trećem poglavlju teze.

Drugo poglavlje ove teze daje jedan novi pristup za proučavanje prvog problema koji smo do sada naveli: ograničene diseminacije poverljivih informacija. U ovom poglavlju uvodimo formalni model koji ograničava komunikacije koje se mogu okarakterisati kao prosleđivanje. U tu svrhu predstavljen je račun nazvan *Confidential π-calculus*, ili skraćeno C_π. Ovaj račun predstavlja jedan fragment čuvenog Milnerovog π-računa [102], koji direktno u sintaksi onemogućava prosleđivanje primljenih imena. Jedini resursi u našem modelu su imena kanala, i mi tretiramo imena kanala kao poverljive informacije. Glavna razlika u poređenju sa originalnim π-računom je ta što u C_π-računu jednom primljena imena kanala kasnije nije moguće poslati. Ovo poglavlje teze se oslanja na publikovani rad

1. I. Prokić. The Cpi-calculus: a model for confidential name passing. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *Electronic Proceedings in Theoretical Computer Science*, pages 115–136. Open Publishing Association, 2019.

ali ga dopunjava i proširuje. Takođe, ovde uvodimo novo pojednostavljeno kodiranje iz π-računa u C_π-račun i predstavljamo kompletan dokaz operacione korespondencije za ovde uvedeno kodiranje. Doprinosi ovog poglavlja u tezi su sledeći:

- Uvođenje novog, jednostavnog fragmenta π-računa koji nam omogućava da predstavimo komuniciranje poverljivih imena ograničavanjem mogućnosti prosleđivanja imena. Činjenica da je uvedeni model fragment uveliko izučavanog π-računa, daje nam mogućnost da iskoristimo već razvijene teorijske rezultate koji postoje za π-račun.

- Uvođenje definicije osobine neprosleđivanja i, kao provera dobre zasnovanosti, pokazivanje da svi procesi iz našeg C_π-računa zadovoljavaju ovu osobinu.

- Koristeći jaku bisimulaciju, bihevioralnu ekvivalenciju iz π-računa, pokazan je jedan bihevioralni identitet koji potvrđuje da u našem računu možemo direktno predstaviti kreiranje zatvorenih domena za kanale.

- Data je detaljna diskusija o ekspresivnosti C_π-računa na nekoliko proširenih primera, koji uključuju reprezentaciju kreiranja zatvorenih domena za kanale, autentikacije, zatvorenih i otvorenih grupa, od kojih svi mogu biti direktno predstavljeni u našem modelu.

- Uvedeno je novo kodiranje π-računa u C_π-račun, čime je pokazano da je naš račun, iako predstavlja tek fragment π-računa koji razmatra samo deo njegove sintakse, podjednako ekspresivan kao i

π-račun. Takođe, u ovom poglavlju dat je detaljan dokaz operacione korespondencije za uvedeno kodiranje.

Centralni pojam svih formalnih modela za konkurentne i distribuirane sisteme je proces. Proces označava entitet koji može da komunicira sa drugim takvim entitetima koristeći zajedničke komunikacione kanale. Neke od prvih i najviše izučavanih procesnih algebri su Milnerov račun komunikacionih sistema (eng. Calculus of Communicating Systems - CCS) [68] i Hoareov račun komunikacionih sekvencijalnih procesa (eng. Communicating Sequential Processes - CSP) [54]. Napomenimo da je CSP poslužio kao osnovni model za programski jezik *Go* koji je razvio Google. Za sveobuhvatniji pregled istorije razvoja procesnih algebri pogledati [8].

Milnerov CCS-račun je jedan od prvih koji je formalno izučavao konkurentne sisteme. Ovaj račun uvodi pojmove paralelne kompozicije, sinhronizacije slanja i primanja na istom imenu, kreiranja privatnih imena i sumacije (izbora). U ovoj tezi operator sumacije nije razmatran, ali verujemo da bi dodavanje ovog operatora moglo da se uradi na uobičajen način. U CCS-računu možemo definisati proces $Alice \mid Bob$ koji označava dva konkurentna potprocesa $Alice$ i Bob, spojena operatorom paralelne kompozicije. Dva konkurentna procesa mogu da se sinhronizuju putem zajedničkog kanala. Recimo, u procesu

$$\overline{chn}.Alice \mid chn.Bob$$

proces na levoj strani $\overline{chn}.Alice$ može da izvede akciju slanja na kanalu chn, dok proces na desnoj strani može da izvede (dualnu) akciju primanja na istom kanalu. Nakon sinhronizacije početni proces se svodi na $Alice \mid Bob$. U CCS-računu procesi mogu i da kreiraju nova imena kanala, čime se modeluje stvaranje privatnih kanala koji nisu dostupni drugim procesima. U procesu

$$((\nu session)Alice) \mid Bob$$

ime kanala $session$ je poznato samo procesu $Alice$ i može biti korišćeno samo za sinhronizacije unutar tog procesa, dok proces Bob

nema nikakvu informaciju o postojanju tog kanala. Ono što CCS-račun ne može da predstavi direktno jeste mobilnost kanala.

Tamo gde je Milner stao sa CCS-računom, nastavio je sa π-računom, koji proširuje CCS da bi dozvolio mobilnost komunikacionih kanala. U π-računu procesi u toku sinhronizacije na kanalu mogu da razmenjuju imena kanala, time stvarajući nove konekcije među sobom. Nekoliko programskih jezika inspirisano je ovim modelom [39, 66, 86, 104, 108, 113].

U π-računu, možemo definisati proces

$$chn!session.Alice \mid chn?x.Bob$$

gde na levoj strani paralelne kompozicije imamo proces koji je spreman da šalje ime kanala *session* putem kanala *chn*, dok na desnoj strani imamo proces koji je spreman da primi bilo koje ime kanala na kanalu *chn*, a zatim da ime x (koje se još zove i „placeholder") unutar procesa *Bob* bude zamenjeno primljenim. Ovaj mehanizam daje novu dimenziju kada se kombinuje sa kreiranjem novih kanala, jer sada kreirani kanali mogu biti razmenjeni među procesima, čime se mogu stvarati privatne konekcije. Ovo je ujedno i poslednji sastojak koji nam je trebao da bismo u π-računu modelovali primer sa prosleđivanjem imena kanala koji smo ranije spominjali:

$$((\nu session)chn!session.Alice) \mid chn?x.forward!x.Bob'$$

U ovom procesu kanal *session* je poznat samo potprocesu sa leve strane paralelne kompozicije. Međutim, nakon sinhronizacije sa potprocesom sa desne strane, početna konfiguracija evoluira u

$$(\nu session)(Alice \mid forward!session.Bob'')$$

gde je ime privatnog kanala *session* sada poznato i desnom potprocesu (to jest, Bob'' predstavlja proces koji se dobije od procesa Bob' kada sva pojavljivanja imena x zamenimo imenom *session*). Dakle, u π-računu domen privatnog imena (što predstavlja deo sistema gde je ime

poznato) se može uvećati nakon sinhronizacije. Ako pretpostavimo da paralelno postoji i treći aktivni process

$$(\nu session)(Alice \mid forward!session.Bob'') \mid forward?y.Carol$$

onda proces koji obuhvata Bob'' može sada proslediti ime kanala *session* tom trećem procesu putem kanala *forward*, a da o tome prethodno nije obavestio *Alice*. Ova diseminacija imena može dovesti do situacije u kojoj je privatnost *Alice* kompromitovana.

C_π-račun diskvalifikuje osobinu prosleđivanja, te stoga $chn?x.forward!x.Bob'$ nije C_π proces. Formalno, naš račun razlikuje imena kanala i imena promenljivih koje se pojavljuju u prefiksu primanja (eng. placeholder). Mi uvodimo dva disjunktna skupa imena, jedan označen sa $\mathcal{C}$ koji čine imena kanala, i drugi označen sa $\mathcal{V}$ koji čine imena promenljivih. Ova distinkcija je iskorišćena kod definisanja jezika našeg modela, jedino imena iz skupa $\mathcal{C}$ mogu biti navedena kao imena za slanje u prefiksu koji definiše ovu akciju. Ovakvo sintaksno ograničenje samo po sebi ne daje uopšteno ograničenje da se imena kanala, koja su posmatrana kao poverljiva informacija, ne mogu razmenjivati, niti ograničava deo sistema u kom ime može biti primljeno. Ono što C_π postiže zapravo je lokalizacija dela sistema koji može poslati ime kanala, a to je onaj deo gde je kanal prvobitno kreiran. Ukoliko je neophodno uspostaviti kontrolu nad slanjem imena nekog kanala, u C_π-računu je dovoljno skoncentrisati se na deo sistema gde je kanal kreiran, dok bi, recimo, u π-računu bilo neophodno kontrolu uspostaviti nad čitavim delom sistema koji zna za dato ime.

Ono što je posledica specifičnosti C_π-računa je to da možemo razlikovati dva nivoa ovlašćenja koja proces može imati u odnosu na neki kanal. Proces koji kreira kanal ima ovlašćenja da komunicira putem kanala, ali takođe može i da pošalje ime kanala drugim procesima. Proces koji u nekom trenutku primi ime kanala stiče pravo da komunicira putem tog kanala, ali ne i da dalje prosleđuje ime tog kanala. Prvi tip procesa u ovoj tezi je nazvan administrator kanala, a drugi korisnik kanala. Svaki administrator je istovremeno i korisnik, ali korisnik ne mora biti i administrator. Još jedna posledica lokalizacije

dela sistema u kom se ime datog kanala može poslati u tezi je iskorišćena i da pokaže kako C_π-račun može biti iskorišćen za modelovanje autentikacije. Naime, sama mogućnost slanja imena nekog kanala zapravo pripada samo administratoru kanala, a onom procesu koji prima to ime zapravo govori sa kojim procesom u tom trenutku komunicira (sa administratorom tog kanala).

Restrikcija koju C_π-račun pravi u odnosu na π-račun zapravo suštinski ne utiče na ekspresivnu moć, a to je i dokazano u samoj tezi. Sama ideja reprezentacije prosleđivanja u C_π-računu je izdvajanje procesa koji bi bili zaduženi isključivo za slanje određenog imena kanala. Drugi procesi bi, ukoliko žele da pošalju neko ime kanala, zapravo umesto slanja samog imena prvo kontaktirali odgovarajući izdvojeni proces koji bi izvršio slanje umesto njih. Ova ideja je u tezi formalizovana u kodiranju π-računa u C_π-račun.

Veliki broj teorijskih istraživanja konkurentnih i distribuiranih sistema direktno je povezan sa π-računom. Mnogi radovi koriste π-račun kao osnovni i proširuju njegovu sintaksu kako bi stekli odgovarajući nivo apstrakcije da modeluju poliadične komunikacije [25, 70], komunikacije višeg reda [71], distribuirane sisteme [50], sigurnost i privatnost [1, 2, 26, 32, 46, 49], i mnoge druge aspekte, uključujući i kontrolu korišćenja resursa koju razmatramo u trećem poglavlju ove teze. Sa druge strane, deo istraživača je koristio sužavanje sintakse π-računa kako bi modelovali asinhrone komunikacije [18, 55], unutrašnju mobilnost [99], i lokalizaciju [67], a naš C_π-račun svakako spada u ovu kategoriju.

Treće poglavlje predstavlja formalni model za izučavanje kontrole pristupa resursima u distribuiranim softverskim sistemima. Ovaj račun u apstraktnom smislu modeluje „capabilities" metodu za kontrolu pristupa, ali takođe i licence za konkurentnu upotrebu, uvodeći pojam deljene autorizacije. Autorizacije se mogu definisati kao funkcije koje određuju prava i privilegije u odnosu na neki resurs. Kao i u

drugom poglavlju, i ovde nam je fokus na sistemima kod kojih je komunikacija centralni pojam, tako da su jedini resursi koje ovde razmatramo zapravo imena komunikacionih kanala. Dakle, autorizacija definiše pravo da se koristi određeni komunikacioni kanal. Deljene autorizacije, koje posmatramo u ovom modelu, definišu prava da se kanal koristi konkurentno. Ovo zapravo znači da jedna autorizacija može biti dostupna većem broju korisnika, ali da je u svakom trenutku može koristiti najviše jedan korisnik. U ovom modelu koristimo destilovane osobine deljenih autorizacija: domen, koji definiše deo sistema u kome je autorizacija implicitno dostupna; brojanje, koje definiše kapacitet; delegacija, koja definiše slanje i primanje samih autorizacija.

Model predstavljen u trećem poglavlju je zapravo ekstenzija π-računa [102], koji se direktno oslanja na pretodno razvijeni račun sa autorizacijama [43, 44]. Iz računa sa autorizacijama [43] preuzeti su sintaksni konstrukti za domen i delegaciju autorizacija. U semantičkom smislu, naš model modifikuje samo značenje domena autorizacije, kako bi dobili mogućnost da obuhvatimo princip o brojanju autorizacija koji proističe iz prirode deljenih autorizacija izučavanih ovde. Treće poglavlje sistematično iznosi rezultate koji su prethodno predstavljeni u publikovanim radovima:

1. J. Pantović, I. Prokić, and H. T. Vieira. A calculus for modeling floating authorizations. In C. Baier and L. Caires, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10854 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2018.

2. I. Prokić, J. Pantović, and H. T. Vieira. A calculus for modeling floating authorizations. *Journal of Logical and Algebraic Methods in Programming*, 107:136 – 174, 2019.

Glavni doprinosi ukupnog rada na modelu koji uvodi deljene autorizacije u π-račun u trećem poglavlju ove teze su sledeći:

- Definisanje novog formalnog računa koji modeluje već spomenute pojmove domena, deljenih resursa, brojanja i delegacije, uvođenjem pojma deljene autorizacije.

- Izučavanje bihevioralne semantike ovog modela. Izvedena bihevioralna karakterizacija pokazuje specifičnu prirodu deljenih autorizacija, naročito odnos između konstrukta za domen autorizacije i konstrukta za paralelnu kompoziciju, koja reflektuje gore spomenuti princip brojanja.

- Uvođenje tipskog sistema koji omogućava izdvajanje procesa koji autorizovano koriste svoje kanale, čak i u prisustvu autorizacija koje su obezbeđene od strane konteksta. Dokazivanje rezultata koji pokazuju da dobro tipiziran process ne samo da uvek koristi svoje kanale autorizovano, već to takođe važi i za sve njegove moguće evolucije.

- Poboljšanje efikasnosti algoritma za proveru tipa uvođenjem drugog tipskog sistema, za koji je pokazano rezultatom tipske korespondencije da je ekvivalentan sa prvim tipskim sistemom.

- Prikazivanje proširenog primera inspirisanog pojmom *Bring Your Own License* iz domena licenci, koji detaljnije opisuje uvedeni model i koji povezuje model sa njegovim mogućim aplikacijama.

- Na osnovu pomenutog proširenog primera pokazan je jedan konkretan pravac za primenu definisanog modela u programskim jezicima. Dat je primer koji posmatra jednu moguću ekstenziju programskog jezika Go[1].

[1] https://golang.org

Model sa deljenim autorizacijama uspostavlja dodatni nivo kontrole korišćenja kanala u odnosu na π-račun. U našem modelu, nije dovoljno da proces ima pristup kanalu, već dodatno mora imati i autorizaciju za korišćenje tog kanala. Sama sintaksa π-računa proširena je sa konstruktima za autorizacije i njihovo delegiranje među procesima. Na primer, proces

$$(license)(Alice \mid Bob)$$

definiše da je jedna autorizacija za korišćenje kanala *license* dostupna procesima *Alice* i *Bob*. Ako, recimo, proces *Bob* prvi započne komunikaciju na kanalu *license* onda konfiguracija data gore postaje

$$Alice \mid (license)LicensedBob$$

gde autorizacija *license* više nije dostupna za *Alice*. Autorizacije mogu biti razmenjene u komunikaciji. Na primer, u

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

proces na levoj strani ima autorizaciju da koristi kanale *auth* i *license*, a prefiks definiše akciju slanja autorizacije za *license* putem kanala *auth*. Sa desne strane, proces može da primi autorizaciju za *license* na kanalu *auth*, i za tu akciju ima odgovarajuću autorizaciju. Nakon sinhronizacije dva procesa, dobijemo

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

gde autorizacija za *license* prelazi sa leve na desnu stranu.

Kao što smo videli u prethodnom primeru, autorizacije zapravo omogućavaju (ili u nedostatku istih, onemogućavaju) komunikacije na kanalima. Ovo važi ne samo za komunikacije u kojima se razmenjuju autorizacije, već i za komunikacije u kojima se razmenjuju imena kanala. Na primer,

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

predstavlja proces u kome ime kanala *license* poslato na kanalu *comm* od potprocesa sa leve strane, može biti primljeno u potprocesu sa desne strane, jer za obe akcije postoje odgovarajuće autorizacije. Sa druge strane, sinhronizacija u procesu

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

nije moguća jer za akcije slanja i primanja postoji samo jedna autorizacija, dok su potrebne dve. U ovoj tezi, ovakvi procesi, koji ne mogu da sinhronizuju svoje dualne akcije zbog nedostatka odgovorajućih autorizacija nazivaju se greškama.

Da bi izdvojili procese koji nisu greške i koji ni u jednoj od mogućih evolucija ne postaju greške, u tezi je predstavljen tipski sistem. Tipski sistem se sastoji od dodele tipova imenima i tipskih pravila, koja definišu uslove koje proces koji se proverava mora zadovoljiti. Ukoliko proces može da prođe definisanu tipsku proveru onda je on „bezbedan", to jest, nije greška i ne svodi se na grešku. Tipovi koje mi ovde dodeljujemo imenima zapravo govore o imenima koja mogu biti bezbedno komunicirana na kanalima. Na primer, posmatrajmo process

$$(exam)(minitest)(alice)alice?x.x!value.0$$

koji može da primi ime kanala i da zatim pošalje *value* na primljenom kanalu. Primanje na *alice* je autorizovano direktno jer je odgovarajuća autorizacija prisutna. Sa druge strane, kasnije slanje je autorizovano samo za imena *exam* i *minitest*. Ako možemo da osiguramo da na kanalu *alice* samo imena *exam* i *minitest* mogu biti komunicirana, tada je ovaj process bezbedan. Stoga, imenu *alice* dodeljujemo tip $\{alice\}(\{exam, minitest\}(\emptyset))$, i to obeležavamo sa

$$alice : \{alice\}(\{exam, minitest\}(\emptyset))$$

kako bismo označili da je *alice* finalno ime (uporediti sa tipom od x datim dole), i da kanal može biti korišćen isključivo za komuniciranje imena *exam* i *minitest*. Poslednja informacija u tipu govori da *exam*

i *minitest* ne mogu biti korišćeni za komunikacije (označeno sa $\emptyset$). Sa druge strane, tip koji bismo dodelili promenljivoj u ovom procesu je $x : \{exam, minitest\}(\emptyset)$, jer x može biti zamenjeno imenima *exam* ili *minitest*, za koje onda treba obezbediti autorizacije. Dakle, za samo ime x autorizacija nije prisutna u procesu u kom se nalazi prefiks, ali autorizacije za dve moguće zamene imena jesu, zbog čega ovakve indirektne autorizacije zovemo kontekstualne autorizacije. Tipski sistem predstavljen u ovoj tezi tretira i direktne i kontekstualne autorizacije. Tipske pretpostavke skupljaju se u tipsko okruženje, obično obeleženo sa Δ, i u odnosu na takvo okuženje vrši se provera procesa pomoću pravila koja se definišu za svaki sintatički konstrukt pojedinačno. Tipsko tvrđenje, koje je oblika $\Delta \vdash_\rho P$, govori da proces P koristi svoje kanale kako je to propisano u tipskom okruženju Δ i da proces poseduje dovoljno autorizacija ako bi bio smešten u kontekst koji bi mu obezbedio dodatne autorizacije navedene u multiskupu ρ. Ove ideje su takođe formalizovane u trećem poglavlju teze.

Četvrto poglavlje sadrži sažetak postignutih rezultata kandidata, pregled literature i razmatra pravce daljih istraživanja.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

EPEI	Every Part Every Interval
TPS	Toyota Production System
TSP	Transport Service Provider

Chapter 1

Introduction

Widespread dissemination of various distributed software systems has changed the way people communicate, learn and run businesses: almost all aspects of human life have become connected to the internet. The system of interconnected computing devices has numerous positive impacts on everyday life, from fast and easy communication through social networks, on-line available computing platforms, to the distributed electronic currency systems based on blockchain technologies. However, sharing information, storage capabilities, computing capabilities, and other resources raises some concerns, among which are security, accessibility and availability issues. There are a plethora of examples in which attackers were able to misuse developers oversights. A recent one is a bug that enabled an incorrect generation of access tokens on Facebook, enabling attackers to steal personal data from almost 50 million accounts [95]. Such examples show the need for controlling information sharing and resource usage in distributed systems, the problems this thesis addresses relying on the use of formal methods.

Ensuring reliability and correctness of software systems is very difficult, and design and verification of such systems, at least in some critical cases, should be mathematically based. Formal methods are techniques that allow for the specification and verification of complex

(software and hardware) systems based on mathematics and formal logic. Formal design usually involves two phases: formal specification and verification. In the formal specification phase a system is defined using a modeling language, usually by means of precise mathematical syntax and semantics. By building a formal specification, one may show in a rigorous way a set of properties of the system. These theorems are validated through mathematical proofs. Formal models for concurrency such as Petri nets [83, 94], communicating state machines [20] and process calculi [54, 68, 73, 74], are examples of formal approaches that can be used to specify and validate application behavior.

The aim of this thesis is to provide formal, process calculi based, models for some security and access control aspects in distributive software systems, paving the way for better understanding of the concepts studied here so as to move towards supporting their correct implementation.

The thesis presents two process algebras, one for modeling controlled information sharing and the other for modeling controlled access to resources in distributed software systems. Accordingly, the rest of the Introduction is divided into two subsections.

1.1 Controlling information sharing

Communication in distributed systems, sometimes involving interactions with unknown and untrusted parties, is an everyday routine and in some cases even indispensable. Often, the exchanged information is private and requires careful handling, in a sense of how it is used. For example, private data, such as credit card number or address, must be revealed for online shopping, but the same kind of information should not be shared further along by the receiving party. Such scenarios bring forward the problem of controlling information sharing in distributed systems. In particular, sharing private information with third

parties may cause undesired dissemination. Even when the parties are trusted, there is a possibility of oversights and unintended misusages.

The privacy problem in distributed systems needs to be perceived from a point of view of technology and law. One of the pioneers in studying privacy in this perspective is Alan Westin, a legal scholar. He noted that "Building privacy controls into emerging technologies will require strong effort..." [114]. Although new technologies bring new threats to privacy, they can also bring new ways to protect privacy [109]. In the work of Solove [105], that deals with the taxonomy of privacy, four kinds of privacy violation are distinguished: information collection, invasions, information dissemination, and information processing. This concept is also elaborated in [61]. Uncontrolled information sharing in distributed systems can be directly related to the information dissemination violation. Solove gives a further detailed taxonomy for information dissemination violation, but all these sub-kinds approximately acknowledge the damage that revealing and spreading sensitive information can cause.

Communication among parties in distributed systems is central and controlling the flow of confidential information poses some obstacles. The entities in such systems can have different capabilities over the information. For example, a credit card holder can send a credit card number or withdraw a certain amount from a bank account; the bank should not be able to disclose a credit card number, but, for example, in order to accept or decline a withdrawal request, the bank should be able to check the funds available in the account. Focusing on controlling the flow of information, we may notice that the party receiving the credit card number should not acquire the ability to send it later on to third parties, i.e., the information should not be forwarded. For the information dissemination violation, the forwarding capability can be recognized as one of the targets where the control may be required.

Considering capabilities over a communication channel one can: use the channel only for writing or reading, create the channel and send its end-points to other entities, forward the received end-point, etc.

Giving the forwarding capability to all parties in the communication a priori induces a problem of post festum control of the dissemination, since the control has to be conducted on all the entities in the system.

Let us consider a simple example where a confidential channel *session* is shared between two parties. Consider that participant *Alice* is the creator of the channel and that she shares one end-point of the channel with *Bob*. The two parties can establish a private interaction over channel *session*, but if *Bob* forwards his end-point to a third party perhaps *Alice*'s expectation will be frustrated.

In some cases, it may be appealing to have the forwarding capability. For instance, when a task is forwarded from a master to a slave process, a slave process is a third party and can seamlessly step in. In our example, this situation may be considered as problematic from the point of view of *Alice*, as she can still believe that the shared end-point of channel *session*, which she may consider confidential, is held by *Bob*. If *session* is a channel created by *Alice*, pointing to some private data and shared exclusively with *Bob*, one might debate that *Bob* should not acquire the forwarding capability only by receiving *session*. In any case, we can distinguish two different kinds of channel end-point sharing in the example, as the first one is conducted by the channel creator (*Alice*), while the second can be recognized as forwarding (by *Bob*).

Several formal models have been developed for the purpose of restricting name-sharing considering a statically determined name scope, such as the ones with name grouping [26] and name hiding [46]. However, there are cases when there is no statically predefined scope for a piece of private information. For instance, in the example above it may be the case that *Alice* can decide at some point in the future to share an end-point of *session* with others. In general, we need private information to be shared also in open-ended systems.

Our approach to addressing the problem mentioned above, i.e., to mathematically reason on restricted dissemination of confidential information, is to develop a formal model which disables forwarding

directly at the syntax level. Our model, which we name *Confidential π-calculus*, abbreviated C_π, is a fragment of the π-calculus [102]. In C_π the only resources are channels, and, hence, we treat channel names as confidential information: channels once received cannot be forwarded later on. This is the only difference with respect to the π-calculus. The contribution of the first part of the thesis is the following:

- We present a simple fragment of the π-calculus that allows modeling confidential name passing by restricting forwarding. Being a fragment of a well-established model, such as the π-calculus, gives us the possibility to directly apply to our model a notable amount of theory already developed.
- We formally characterize the non-forwarding property and, as a sanity check of the process model, we show that all C_π processes respect this property.
- Based on the notion of behavioral equivalence relation from the π-calculus we present a behavioral identity attesting that closed domains for channels can be directly represented in our model.
- We further elaborate on the expressiveness of the C_π-calculus by showing examples that include closed domains for channels, authentication, and open-ended groups, all of which are also directly represented in the model.
- We investigate the expressive power of our model when compared to the π-calculus and show that the π-calculus can be encoded into the C_π-calculus. We show an operational correspondence result for the given encoding.

1.2 Access control

Another aspect addressed in this thesis is a study of access control in distributed software systems. The control of access to computational resources is becoming more important, despite their availability is growing in scale every year, since such growth encompasses different forms of resource access and raises some issues. The need to control access can be motivated by many reasons, such as privacy, security and capacity constraints. As examples of capacity constraints consider that physical devices (printers, cell phones, processors, etc.) all have finite capabilities, while some virtual devices (shared memory cell, web service, etc.) have infinite potential but their availability is frequently limited.

Privacy and security are central issues in the development of distributed systems. One of the reasons is that distributed systems are nowadays highly complex and heterogeneous, and the access control in such systems can be an error-prone task. Justifications for these claims are appearing almost every day, one of which is the incorrect access token generation on Facebook mentioned previously. For such bad examples companies suffered the loss of millions of dollars, but even worse, privacy and security of millions of users have been compromised. Formal modeling and verification can be a step towards more reliable distributed software systems [21].

Various methods for controlling access in distributed systems have been proposed in the past, as the scale and the structure of such systems has been changing. Considering systems that are small or in which the number of participants is predefined, access to resources of the system is usually achieved through access control lists (ACL). The ACL method uses lists of permissions attached to resources, where the right to access a resource is granted only if the user is mentioned as a subject (with the right permission) on the access list of the resource. While the ACL method provides a natural way to control access, when it comes to large-scale systems that are highly dynamic this method

becomes hard to implement. This comes from the fact that access control lists keep information of each user individually, which becomes a serious overhead when the number of users of a system is large and frequently changes. As an example consider an application, such as Facebook, that is used by millions of users whose number varies over time. Having lists of users authorized to access each resource (e.g., pictures or posts of users) becomes arguably impractical.

Role-based access control method [96] (RBAC) was introduced as an alternative way to deal with the scalability of ACLs. In the RBAC method, a set of roles is established and a role is assigned to each user. A user can have more than one role. For instance, to allow access to a picture of a Facebook user it might not be necessary to rely directly on the identity of the user attempting to access the picture. In practice, it is enough to check if the user trying to access the picture has the role of a "friend" of the owner of the picture. Besides the advantages (and disadvantages) of the RBAC method with respect to the ACL method, the former still suffers from the fact that there usually must be a central authority that issues and checks the roles of the users.

The capability-based method for access control [97] is more suitable for uncentralized systems. In this method, unforgeable references are created and issued by a central authority, but once issued, these references are held by users. Only when the user wants to access the recourse the capability is checked. Hence, a central authority does not have to keep access control information for each user individually, it only checks the validity of the references when needed, which significantly reduces the load on central authorities [116]. Furthermore, capabilities can be delegated from one user to another without informing the central authority.

Another domain conveying some of the principles of the capability-based access control is that of use licenses: a user is allowed to use a software application only if it owns a proper license, where we also find an explicit notion of delegation. For example, a user, while deploying

his software application on the cloud computing platform, may delegate his own license, potentially losing access to use the application himself, a notion known as Bring Your Own License [38] (BYOL). A specific kind of use licenses, the concurrent use licenses, also provide a flexibility to use the license in a shared way [10]. As an example consider a software application that is licensed to be used by an institution. In this case, licenses are available to all users in the closed domain of institution, but there is a bound on the number of licenses, hence the total number of deployed applications at any point cannot exceed the total number of licenses [5].

We study capabilities and licenses in an abstract way in a process model of floating authorizations. Authorization is a function determining rights and privileges over a resource. As before, our focus is on communication centered systems, and hence, in our model, the only resources considered are communication channels. Thus, an authorization determines the right to use a communication channel. A floating authorization in our model represents the right to use a channel in a shared way like in the setting of concurrent use licenses. In such constellation, we exploit distilled dimensions of floating authorizations: domain (to capture where access may be implicitly granted), accounting (to capture the capacity), and delegation (to capture explicit granting).

Our model is an extension of the π-calculus [102], and it builds on previous developments [43, 44]. From the calculus for authorizations of [43] we adopt the syntax constructs for authorization scoping and delegation. Semantically, we modify the meaning of authorization scoping construct so as to be able to represent the accounting principle arising from the floating nature of authorizations. Hence, our aim is, therefore, to address the specific notion of accounting, emerging from the settings of capabilities and licenses, by modifying an existing model in the minimal necessary way. The contribution of the second part of the thesis is as follows:

- We present a calculus that models the notions of domain, implicit granting, accounting, and delegation of floating authorizations.
- We investigate the behavioral semantics of our model: our behavioral characterizations show the specific nature of floating authorizations, in particular, the correlation between the authorization construct and the parallel composition, reflecting our accounting principle.
- We present a type analysis that singles out processes in which each use of a channel is properly authorized, even in the case of contextual authorizations, and we show type soundness and type safety.
- We introduce another type system that allows for a more efficient implementation than the original one and we show the respective typing correspondence.
- We give an extended example inspired by the notion of *Bring Your Own License* from the licensing setting.
- Based on the extended example, we show one direction towards the application of our model by considering a possible extension of the Go programming language[1].

1.3 Publications and structure of the thesis

Controlled name passing is presented in Chapter 2, where we introduce the C_π-calculus. As noted in Section 1.1, our calculus models interactions in which information can be shared between two parties, but cannot be forwarded by the receiving party. Chapter 2 is based on the paper:

[1] https://golang.org

1. I. Prokić. The Cpi-calculus: a model for confidential name passing. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *Electronic Proceedings in Theoretical Computer Science*, pages 115–136. Open Publishing Association, 2019.

The work reported here extends the work presented in the mentioned paper by a more pedagogical introduction of the model and by encompassing all the results. Furthermore, the encoding presented in this document simplifies the one given in the above paper and completes the proof of operational correspondence.

Controlling resource usages is considered in Chapter 3, where we introduce our calculus of floating authorizations. Our model allows separating resource awareness from resource usages by authorizations. Chapter 3 is based on publications:

1. J. Pantović, I. Prokić, and H. T. Vieira. A calculus for modeling floating authorizations. In C. Baier and L. Caires, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10854 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2018.

2. I. Prokić, J. Pantović, and H. T. Vieira. A calculus for modeling floating authorizations. *Journal of Logical and Algebraic Methods in Programming*, 107:136 – 174, 2019.

where the listed journal extends the listed conference publication.

In Chapter 4, we conclude by presenting the contributions of the candidate, providing an overview of the related literature, and, finally, showing some ideas for improvements and possible extensions of the work presented in this thesis.

Chapter 2

A calculus for confidential name passing

In this chapter, we present the C_π-calculus, which is a fragment of the π-calculus [72, 73, 74, 102]. We start by a small introduction on process algebras.

As in the π-calculus, the building blocks of the C_π language are *processes*. A process represents an entity that can synchronize with other processes through communication links (channels) that they share. Some of the first and well-known process models are a Calculus of Communicating Systems (CCS) [68] and Communicating Sequential Processes (CSP) [54]. The latter has influenced the design of Google's *Go* programming language. See [8] for a more comprehensive overview of the history of process algebras.

The CCS-calculus models concurrent systems formally by introducing the notions of parallel composition, synchronization of input and output action on the same name, the private names and the choice. The choice operator is not considered throughout this thesis, and hence we will not mention its interpretation nor its properties. For instance,

a CCS process $Alice \mid Bob$ represents that $Alice$ and Bob are simultaneously active processes. Furthermore, these processes can synchronize via shared name. For instance, in process

$$\overline{chn}.Alice \mid chn.Bob$$

process $\overline{chn}.Alice$ can synchronize the output action on name chn with the input action of process $chn.Bob$, since both are active in parallel. In that case, the process reduces to $Alice \mid Bob$.

In CCS a name can be specified as private. In process

$$((\nu session)Alice) \mid Bob$$

name $session$ is known only to $Alice$ and can be used only internally, while Bob cannot acquire the name. What CCS fails to capture directly is name mobility.

The π-calculus takes as its basis CCS, but it extends it in an important way by allowing name mobility. Namely, processes now can, while synchronizing their actions (via communication channels), transmit names of channels. This model has also influenced the design of several programming languages [39, 66, 86, 104, 108, 113]. For example, in the π-calculus we may specify

$$chn!session.Alice \mid chn?x.Bob$$

where the left process can send channel name $session$ on channel chn, the right process can receive a name on the same channel and replace the placeholder name x in Bob with the received name. Another important aspect is that a π process can share a private communication channel with other processes via synchronization, establishing private connections. This is the last ingredient needed to represent the scenario given in the Introduction, as we may now write

$$((\nu session)chn!session.Alice) \mid chn?x.forward!x.Bob'$$

where the channel *session* is privately held by the left process. After the synchronization, the above configuration evolves to

$$(\nu session)(Alice \mid forward!session.Bob'')$$

where private channel *session* is received in the right process (i.e., Bob'' represents the process derived from Bob' by replacing each occurrence of name x with name *session*), hence enlarging the scope of the name. Assuming there is a third party active,

$$(\nu session)(Alice \mid forward!session.Bob'') \mid forward?y.Carol$$

the process comprehending Bob'' can forward received name *session* to the third party on channel *forward* without a need to notify *Alice*, hence potentially compromising the privacy of *Alice* (cf. Section 1.1).

The subject of this section, the C_π-calculus, allows reasoning on confidentiality in a fragment of the π-calculus. By restricting forwarding in a way that channel name once received by a process cannot be later sent by the same process, we gain a suitable abstraction level to reason on, e.g., groups [26] and name hiding [46], directly in the C_π without additional language constructs. Since we are dealing only with a fragment of the π-calculus rather than with its extension, our model can reuse all the theory already developed for the π.

Extensive theoretical research conducted in the past is directly connected to the π-calculus in many ways. Many of these works extend the π-calculus syntax by variety of constructs so as to gain a suitable abstraction level to reason about, e.g., polyadic communications [25, 70], higher-order communication [71], distributed systems [50], and many others, including security and privacy [1, 2, 26, 32, 46, 49]. In contrast, some of the research exploits restricting the syntax of the π-calculus to reason on asynchronous communications [18, 55], internal mobility [99], and locality [67].

Overview of the chapter. The syntax of the process model is presented in Section 2.1. The action semantics is presented in Section 2.2,

π	::=		*Prefixes*
		$a!k$	output
	$\mid$	$a?x$	input
	$\mid$	$[a = b]\pi$	matching
P	::=		*Process terms*
		0	termination
	$\mid$	$\pi.P$	prefix
	$\mid$	$P \mid P$	parallel composition
	$\mid$	$(\nu k)P$	name restriction
	$\mid$	$!P$	replication

Table 2.1: Syntax of C_π

followed by an investigation of some basic properties together with the definition of the non-forwarding property in Section 2.2.1. The reduction semantics is briefly introduced in Section 2.3. Section 2.4 presents a behavioral equivalence (strong bisimilarity) and a property called closed domains for channels, capturing that closing the scope of a channel can be represented directly in C_π. Section 2.4.1 briefly presents another behavioral equivalence, the strong barbed equivalence relation, and Section 2.4.2 gives a method for identifying non-forwarding π processes. In Section 2.5 we present some interesting scenarios modeled in C_π, such as authentication schemes (Section 2.5.1), closing the domain for channels (Section 2.5.2), and open-ended groups (Section 2.5.3). Section 2.6 presents an encoding from the π-calculus into the C_π-calculus and the operational correspondence result that validates the encoding and informs on the expressive power of C_π.

2.1 Syntax

In this section, we introduce the language of the C_π-calculus. As we have noted, the building blocks of our language are processes, and processes may communicate using names. The names themselves are an abstraction for communication links. Communication links are modeled as named communication channels that can connect two or more processes.

In C_π, we distinguish *variable* and *channel* names by introducing two disjoint sets for each kind. We use $\mathcal{C}$ to denote the set of channel names, ranged over by $k, l, m, \ldots$, and $\mathcal{V}$ to denote the set of variable names, ranged over by $x, y, z, \ldots$ The union of the two sets is denoted by $\mathcal{N}$, and we let $a, b, c, \ldots$ range over $\mathcal{N}$. The use of each of the sets is explained below when language constructs are introduced.

The syntax of the language is given in Table 2.1. Notice that we do not consider the sum operator [102] since our goal is to study confidentiality in a minimal setting, but we believe the sum can be added following expected lines. The first part of the table introduces the prefixes, used in the definition of three types of processes:

- The output prefix $a!k.P$ describes an action in which the object, channel name k, is sent on the subject, name a, after which the process evolves to P. Notice that only a channel name, i.e., a name from set $\mathcal{C}$, can be the object of an output action. This is the only difference with respect to the π-calculus, where the name of a variable can also appear as the object of an output action. The subject of an output action can be either a channel or variable name, like in the π-calculus. Hence, received names can be used to communicate but cannot be communicated. For example, process $b!k.0$ can send k on b and evolve to 0.

- The input prefix $a?x.P$ describes an action in which a name is received on the subject name a. The received name is substituted in the continuation process P for a variable name x, and

the original process evolves to the term resulting from this substitution. Here, x, also called a placeholder, is bound in process P (see Definition 2.1.1). For example, process $a?x.x!k.0$ can receive a name on a and substitute x by it in the continuation. Assuming the received name is b, the above process evolves to $b!k.0$. Notice that the object of an input can only be a variable name, i.e., name from set $\mathcal{V}$. Hence, process as $a?x.k!x.0$ excluded by the C_π syntax is a π process.

- The match prefix $[a = b]\pi.P$ activates the action prescribed by prefix $\pi.P$ only if $a = b$ and it blocks the action otherwise. For example, process $a?x.[x = b]x!k.0$ receives a name on a, then either: it sends k on b if the name received is b; else if the received name is not b it performs no further actions.

We comment the rest of the process constructs:

- The termination 0 denotes the process that exhibits no actions. Notice that we have used it in the examples above to signal when a thread has performed all its actions.

- The parallel composition $P \mid P$ denotes that two processes are simultaneously active, and possibly can synchronize their actions. For example, in $a!k.0 \mid a?x.x!l.0$ the left thread can synchronize the output action with the input action of the right thread, and can also exhibit any of the individual actions of the two threads.

- The name restriction $(\nu k)P$ denotes the creation of a new (channel) name k, known only to process P. Channel k can be used as a private medium for communications of the components of process P. For example, process $(\nu k)(k!l.0 \mid k?x.0)$ can use channel k for synchronization of the two threads but cannot interact with other processes along channel k. On the other hand, the restricted channel can be shared with other processes if it is communicated in a message. The explanation of sharing restricted

names and its interplay with confidentiality in C_π can be found in Section 2.2, Section 2.4 and Section 2.5.

- The replication $!P$ denotes a process with potentially infinite behavior. Informally, $!P$ can be seen as an infinite parallel composition of the copies of process P, i.e., $P \mid P \mid \ldots$. For example, process $!a!k.Q$ can send k on a and activate the original process in parallel.

We name the operators' precedence from highest to lowest: prefixes, name restriction, replication, and parallel composition. For example, the above process $a!k.0 \mid a?x.x!l.0$ stands for $(a!k.0) \mid (a?x.(x!l.0))$, and $(\nu k)P \mid Q$ stands for $((\nu k)P) \mid Q$.

We denote with $\mathsf{n}(P)$ the set of all names (channels and variables from $\mathcal{N}$) appearing in the process P. Some of these names are said to be *free*, while the rest are called *bound*.

Definition 2.1.1 (Free and bound names)**.** *For any C_π process the set of free names $\mathsf{fn}(P)$ and the set of bound names $\mathsf{bn}(P)$ are defined as follows.*

$$
\begin{aligned}
\mathsf{fn}(0) &= \emptyset \\
\mathsf{fn}(a!k.P) &= \{a,k\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(a?x.P) &= \{a\} \cup (\mathsf{fn}(P) \setminus \{x\}) \\
\mathsf{fn}([a=b]\pi.P) &= \{a,b\} \cup \mathsf{fn}(\pi.P) \\
\mathsf{fn}(P \mid Q) &= \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}((\nu k)P) &= \mathsf{fn}(P) \setminus \{k\} \\
\mathsf{fn}(!P) &= \mathsf{fn}(P)
\end{aligned}
\qquad
\begin{aligned}
\mathsf{bn}(0) &= \emptyset \\
\mathsf{bn}(a!k.P) &= \mathsf{bn}(P) \\
\mathsf{bn}(a?x.P) &= \{x\} \cup \mathsf{bn}(P) \\
\mathsf{bn}([a=b]\pi.P) &= \mathsf{bn}(\pi.P) \\
\mathsf{bn}(P \mid Q) &= \mathsf{bn}(P) \cup \mathsf{bn}(Q) \\
\mathsf{bn}((\nu k)P) &= \{k\} \cup \mathsf{bn}(P) \\
\mathsf{bn}(!P) &= \mathsf{bn}(P)
\end{aligned}
$$

Notice that in $(\nu k)P$ and $a?x.P$ the channel k and variable x are bound and that these are the only operators that bind names. In these two cases, we say that k and x are binding with *scope* P. The scope of a bound name determines the process which is the only one that knows the name. Notice that $\mathsf{fn}(P) = \mathsf{n}(P) \setminus \mathsf{bn}(P)$. For free names, the scope is not predefined since free names may be known by other

processes. We also identify a set of free channels appearing as objects of output prefixes in a process, so as to be able to talk about names a process can send.

Definition 2.1.2 (Free output object names)**.** *For any C_π process P, the set of free output object names $\mathsf{fo}(P)$ is defined as follows.*

$$\begin{aligned}
\mathsf{fo}(0) &= \emptyset \\
\mathsf{fo}(a!k.P) &= \{k\} \cup \mathsf{fo}(P) \\
\mathsf{fo}(a?x.P) &= \mathsf{fo}(P) \\
\mathsf{fo}([a=b]\pi.P) &= \mathsf{fo}(\pi.P) \\
\mathsf{fo}(P \mid Q) &= \mathsf{fo}(P) \cup \mathsf{fo}(Q) \\
\mathsf{fo}((\nu k)P) &= \mathsf{fo}(P) \setminus \{k\} \\
\mathsf{fo}(!P) &= \mathsf{fo}(P)
\end{aligned}$$

To precisely define replacements of names in processes (such as the ones described in the receive actions) we give a precise definition of the substitution. Before that, we introduce a convention that bound names must not be mentioned by substitutions (we will come back to this point).

Definition 2.1.3 (Substitution)**.** *A substitution is a mapping from $\mathcal{N}$ to $\mathcal{N}$ that is not the identity only on a finite subset of $\mathcal{N}$, and that maps $\mathcal{C}$ only to $\mathcal{C}$. We define the support of σ to be finite set $\{a \mid \sigma a \neq a\}$, and the co-support of σ to be finite set $\{\sigma(a) \mid \sigma a \neq a\}$. We write $\mathsf{n}(\sigma)$ for the union of the support and the co-support of σ. If substitution σ is applied to process P, then the resulting process $P\sigma$ is defined as follows.*

$$\begin{aligned}
0\sigma &= 0 \\
(a!k.P)\sigma &= \sigma(a)!\sigma(k).P\sigma \\
(a?x.P)\sigma &= \sigma(a)?x.P\sigma \\
([a=b]\pi.P)\sigma &= ([\sigma(a)=\sigma(b)](\pi.P)\sigma \\
(P \mid Q)\sigma &= P\sigma \mid Q\sigma \\
((\nu k)P)\sigma &= (\nu k)P\sigma \\
(!P)\sigma &= !P\sigma
\end{aligned}$$

If the support of σ is $\{a_1, \ldots, a_n\}$, and $\sigma(a_i) = b_i$, for $i = 1, \ldots, n$, then instead of $P\sigma$ we may also write $P\{b_1, \ldots, b_n/a_1, \ldots, a_n\}$.

For example, we have $(x!l.0)\{k/x\} = k!l.0$. Notice also that substitution applied on a process does not affect the bound names of the process. Only free names can be substituted and the convention preceding Definition 2.1.3 ensures that substitution does not map free names into bound names. This is required to avoid name clashes, the notion explained next.

So far, our syntax allows us to define a process in which a name may be used in distinct binding occurrences, and also to appear free elsewhere. For example, we can write $(\nu k)a!k.0 \mid (\nu k)a?x.x!k.0$, where, the two restricted names are identified with the same k. After synchronization of the two branches, name k from the left branch can clash with the k in the right branch. In order to simplify the handling of bound names, which are to be considered distinct regardless of their identifier, we identify processes up to α-conversion defined next.

Definition 2.1.4 (α-conversion)**.** *Processes P and Q are α-convertible if we may obtain Q from P by a finite number of replacements of subterms $(\nu k)P_1$ and $a?x.P_2$ in P by $(\nu l)P_1\{l/k\}$ and $a?y.P_2\{y/x\}$, where $l \notin \mathsf{n}(P_1)$ and $y \notin \mathsf{n}(P_2)$. If P and Q are α-convertible we write $P \equiv_\alpha Q$.*

Notice that since we identify α-convertible processes, $P \equiv_\alpha Q$ implies $P = Q$. We may then say that process $(\nu k)a!k.0 \mid (\nu k)a?x.x!k.0$ is equal to

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

which allows avoiding the name clash when reasoning on the synchronization.

To avoid name clashes, in general, we use a sort of Barendregt convention as adopted in the π-calculus in [102] Convention 1.1.7, which states that all free names and names of the substitutions are distinct from all bound names in any processes and substitutions under consideration. Furthermore, to avoid explicitly working with α-conversion,

we also assume that all bound names among themselves in any process under consideration are different.

We remark that this treatment of bound and free names is appealing in theoretical works on the π-calculus such as ours, but when a formalization of the π-calculus in some theorem proving systems (e.g., Coq [13], Isabelle/HOL [76]) is conducted, a more precise way of handling free and bound names is needed, like adopting de Bruijn notation of names [35] to the π-calculus [42, 51, 82].

2.2 Action semantics

In the previous section, we informally presented some examples where processes perform inputs, outputs and synchronize their actions. We formalize these notions in this section where we define the action semantics of our model in terms of a labeled transition system, that in turn matches the one of [102] for the sum-free π-calculus. Intuitively, each process evolution involves three parts: the starting process, the action performed and the resulting process. This kind of operational semantics allows us to characterize the behavior of a process relying on the behavior of its subparts, including their interactions. An action of a process describes what the environment can observe when interacting with the process. We now define the actions.

Definition 2.2.1 (Actions)**.** *The observable action α is defined as*

$$\alpha ::= \quad k!l \quad | \quad k?l \quad | \quad (\nu l)k!l \quad | \quad \tau$$

and we denote with $\mathcal{A}$ the set of all actions.

We may recognize that the first two actions correspond to the process prefixes. Prefixes $k!l.P$ and $k?x.P$ describe the potential of a process to perform an action and observable actions $k!l$ and $k?l$ describe the action itself: the first sending and the second receiving the channel l on the channel k. Notice that in our (early) semantics,

the input action already identifies the received channel (l), and does not mention the input prefix variable (x). In action $(\nu l)k!l$ the sent channel l is bound, denoting that process performing the action sends a fresh channel (here l). This allows for scope extrusion, explained later. A process performing invisible action τ evolves internally, hence without interacting with the environment. To retain the same notation as for processes, we denote by $\mathsf{fn}(\alpha)$, $\mathsf{bn}(\alpha)$ and $\mathsf{n}(\alpha)$, the sets of free, bound and all names of action α, respectively. As we noted above, these sets contain only channels, and not variables.

Definition 2.2.2 (Free and bound names of actions)**.** *For observable action α we define a set of free and bound names as follows.*

$$\begin{array}{rclcrcl} \mathsf{fn}(k!l) & = & \{k,l\} & & \mathsf{bn}(k!l) & = & \emptyset \\ \mathsf{fn}((\nu l)k!l) & = & \{k\} & & \mathsf{bn}((\nu l)k!l) & = & \{l\} \\ \mathsf{fn}(k?l) & = & \{k,l\} & & \mathsf{bn}(k?l) & = & \emptyset \\ \mathsf{fn}(\tau) & = & \emptyset & & \mathsf{bn}(\tau) & = & \emptyset \end{array}$$

As for the processes, we extend here our convention that all free names are different from the bound names, and that all bound names are pairwise distinct, not only in all processes and substitutions but also including all actions under consideration. The only exception of this convention is when the scope extrusion is performed, as explained later. This matches Convention 1.4.10 in [102].

The labeled transition relation is the least relation in $\mathcal{P} \times \mathcal{A} \times \mathcal{P}$, where $\mathcal{P}$ is the set of all processes, that satisfies the rules given in Table 2.2. We describe the rules on salient points.

- Rules (OUT), (IN) and (MATCH) directly correspond to the explanations of the corresponding syntactic constructs. For example, considering process $k?x.[x = l]x!m.0$ receives channel l, by rule (IN) we derive

$$k?x.[x = l]x!m.0 \xrightarrow{k?l} [l = l]l!m.0$$

$$\text{(OUT)}\quad k!l.P \xrightarrow{k!l} P \qquad \text{(IN)}\quad k?x.P \xrightarrow{k?l} P\{l/x\} \qquad \text{(MATCH)}\ \frac{\pi.P \xrightarrow{\alpha} P'}{[a=a]\pi.P \xrightarrow{\alpha} P'}$$

$$\text{(RES)}\ \frac{P \xrightarrow{\alpha} P' \quad k \notin \mathsf{n}(\alpha)}{(\nu k)P \xrightarrow{\alpha} (\nu k)P'} \qquad \text{(OPEN)}\ \frac{P \xrightarrow{k!l} Q \quad k \neq l}{(\nu l)P \xrightarrow{(\nu l)k!l} Q}$$

$$\text{(PAR-L)}\ \frac{P \xrightarrow{\alpha} Q \quad \mathsf{bn}(\alpha) \cap \mathsf{fn}(R) = \emptyset}{P \mid R \xrightarrow{\alpha} Q \mid R} \qquad \text{(COMM-L)}\ \frac{P \xrightarrow{k!l} P' \quad Q \xrightarrow{k?l} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{(CLOSE-L)}\ \frac{P \xrightarrow{(\nu l)k!l} P' \quad Q \xrightarrow{k?l} Q' \quad l \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu l)(P' \mid Q')} \qquad \text{(REP-ACT)}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\text{(REP-COMM)}\ \frac{P \xrightarrow{k!l} P' \quad P \xrightarrow{k?l} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P} \qquad \text{(REP-CLOSE)}\ \frac{P \xrightarrow{(\nu l)k!l} P' \quad P \xrightarrow{k?l} P'' \quad l \notin \mathsf{fn}(P)}{!P \xrightarrow{\tau} (\nu l)(P' \mid P'') \mid !P}$$

Table 2.2: LTS Rules.

where the received channel l substitutes the variable x. Since by rule (OUT) we derive

$$l!m.0 \xrightarrow{l!m} 0$$

and since the matched names preceding the output coincide, by (MATCH) we have that

$$[l=l]l!m.0 \xrightarrow{l!m} 0$$

If the received channel in process $k?x.[x = l]x!m.0$ is not l, but, say n, again by (IN) we derive

$$k?x.[x = l]x!m.0 \xrightarrow{k?n} [n = l]n!m.0$$

only now the output action of the final process is blocked due to the mismatch in the prefix, and hence the process cannot perform further actions. This is the consequence of the fact that the only rule in Table 2.2 that deals with the match operator, i.e., (MATCH), cannot be applied since n and l are distinct. Notice also that in the above example the transitions are justified by rules in Table 2.2 in a unique way, in a sense that only rules (IN), (OUT) and (MATCH), respectively, can be applied.

- Rule (RES) lifts the action of the process scoped with channel restriction and ensures that the action does not mention the channel specified in the restriction. The effect of the side condition is that the restricted channel is never mentioned in the action visible to the process's environment, hence keeping the channel private. The only exception is when a channel is sent in a message (cf. rule (OPEN)). For example, for process $k!l.0$, by (OUT), we can derive

$$k!l.0 \xrightarrow{k!l} 0$$

but restricting channel k in the above process we end up with process $(\nu k)k!l.0$ which only action (output) is blocked due to the side condition of (RES). Also, process $(\nu l)k!l.0$ is blocked by this rule, however, it can proceed by applying the rule explained next.

- Rule (OPEN) allows for sending a restricted channel by opening its scope, which, combined with rule (CLOSE-L) (explained later), enables the communication of restricted names, where their scope is enlarged as a consequence (scope extrusion). Going back to the last example, applying rule (OPEN) after (OUT)

we derive

$$(\nu l)k!l.0 \xrightarrow{(\nu l)k!l} 0$$

where the label of the action carries the information that the channel sent is fresh. Notice that the side condition of the rule ensures that the subject of the action is not the restricted one, therefore, processes $(\nu k)k!l.0$ and $(\nu k)k!k.0$, cannot evolve, as no rule of Table 2.2 can be applied.

- In rule (PAR-L) the action of the left branch is lifted at the level of the parallel composition while avoiding the case when the bound channel of the action is specified as free in the right branch. The symmetric rule (PAR-R), where the action originates from the right branch is omitted from the table. For example, by (OUT) and (PAR-L) we may derive

$$k!l.0 \mid k?x.[x = l]x!m.0 \xrightarrow{k!l} 0 \mid k?x.[x = l]x!m.0$$

where the process carry out the action of the left branch and the right branch does not exhibit any action. Rules (PAR-L) and (PAR-R) combined allow to interleave the behavior of both branches, capturing the fact that both branches are active. Moreover, the left branch can synchronize the action with the right branch: this is explained by the next rule.

- Rule (COMM-L) describes the synchronization of the two dual external actions, output and input. After the synchronization the observable action of the overall process is an internal step, i.e., τ, denoting that the process at this point is not interacting with the environment. In the example above, applying (OUT) and (IN) we derive

$$k!l.0 \xrightarrow{k!l} 0 \quad \text{and} \quad k?x.[x = l]x!m.0 \xrightarrow{k?l} [l = l]l!m.0$$

then, by (COMM-L) we have that

$$k!l.0 \mid k?x.[x = l]x!m.0 \xrightarrow{\tau} 0 \mid [l = l]l!m.0$$

We also omit the symmetric cases of the rules (COMM-L) and (CLOSE-L).

- Rule (CLOSE-L) handles the case when the name sent by the left branch is bound. The right branch again performs the input action, and after the synchronization, the scope of the sent channel, previously opened in rule (OPEN), is now closed. As an example, consider process $(\nu l)k!l.0 \mid k?x.x!m.0$. By rules (OUT), (OPEN) and (IN) we get

$$(\nu l)k!l.0 \xrightarrow{(\nu l)k!l} 0 \qquad\qquad k?x.x!m.0 \xrightarrow{k?l} l!m.0$$

and by (CLOSE-L)

$$(\nu l)k!l.0 \mid k?x.x!m.0 \xrightarrow{\tau} (\nu l)(0 \mid l!m.0)$$

The side condition ensures that the received channel is not specified as free in the right branch, thus avoiding unintended name capturing.

- Rules (REP-ACT) allows for a replicated process to perform an action while activating a copy of the original process in parallel. For example, since by (IN) we can derive

$$k?x.[x = l]x!m.0 \xrightarrow{k?l} [l = l]l!m.0$$

Then, by (REP-ACT) we conclude

$$!k?x.[x = l]x!m.0 \xrightarrow{k?l} [l = l]l!m.0 \mid !k?x.[x = l]x!m.0$$

Thus, we may say that process $!k?x.[x = l]x!m.0$ is repeatably available to receive a channel and afterwards send m only if the received channel is l.

- Rules (REP-COMM) and (REP-CLOSE) allow for two copies of the same replicated process to synchronize their actions, where in the

latter rule the communicated channel is fresh. As an example consider process

$$P = k!l.0 \mid k?x.[x = l]x!m.0$$

By rules (OUT) and (PAR-L) we can derive

$$P \xrightarrow{k!l} 0 \mid k?x.[x = l]x!m.0$$

and by rules (IN) and (PAR-R) also

$$P \xrightarrow{k?l} k!l.0 \mid [l = l]l!m.0$$

Then, by rule (REP-COMM) we have

$$!P \xrightarrow{\tau} (0 \mid k?x.[x = l]x!m.0) \mid (k!l.0 \mid [l = l]l!m.0) \mid !P$$

Let us now consider process

$$Q = (\nu l)k!l.0 \mid k?x.x!m.0$$

By rules (OUT), (OPEN) and (PAR-L)

$$Q \xrightarrow{(\nu l')k!l'} 0 \mid k?x.x!m.0$$

where we α-converted Q by renaming l with a fresh channel l', to represent that the restricted name of this copy of Q is unique. Let us now take another copy of Q and let us by (IN) derive $k?x.x!m.0 \xrightarrow{k?l'} l'!m.0$, where the received channel matches the one sent by the first copy of Q. Since the restricted channel of the second copy of Q is also fresh, and hence, to be distinguished from l (and also l'), we α-convert Q again by renaming l to some fresh channel l'' and then we apply (PAR-R)

$$Q \xrightarrow{k?l'} (\nu l'')k!l''.0 \mid l'!m.0$$

Then, by (REP-CLOSE) we get

$$!Q \xrightarrow{\tau} (\nu l')(0 \mid k?x.x!m.0 \mid (\nu l'')k!l''.0 \mid l'!m.0) \mid !Q$$

Notice that introducing fresh channels by α-converting each copy of process Q resulted that in the last derived process each bound channel is distinct, thus avoiding name clashes.

The derivation in the last example was possible thanks to Definition 2.1.4 since all α-converted processes are implicitly identified. This kind of derivations as in the last example can be formalized by introducing the explicit rule

$$\frac{P \xrightarrow{\alpha} Q \qquad P \equiv_\alpha R}{R \xrightarrow{\alpha} Q}$$

which follows directly by considering processes equal up to α-conversion.

2.2.1 Properties of the labeled transition system

This section presents some basic properties of the labeled transition system (LTS). Let us recall that the LTS introduced in Section 2.2, perfectly matches the one given in [102] (for the sum-free π-calculus). Hence, since C_π-calculus is a fragment of the π-calculus, all the results given in [102] hold also for the C_π. We present here only the results specific to our model.

The distinguishing feature of the C_π-calculus syntax, that variables do not appear as objects of output prefixes, reflects in the evolutions of processes. Namely, the set of free objects of output prefixes in the process is possibly augmented only by opening the scope of a channel. Another specific property is that the set of free objects of output prefixes in the process is invariant to input actions. These results are stated in the next lemma.

Lemma 2.2.3 (Free output objects and transitions)**.** *Let P and P' be C_π processes such that $P \xrightarrow{\alpha} P'$.*

1. *If* $\alpha = k?l$ *then* $\mathsf{fo}(P') = \mathsf{fo}(P)$.
2. *If* $\alpha = k!l$ *then* $l \in \mathsf{fo}(P)$ *and* $\mathsf{fo}(P') \cup \{l\} = \mathsf{fo}(P)$.
3. *If* $\alpha = (\nu l)k!l$ *then* $\mathsf{fo}(P') \subseteq \mathsf{fo}(P) \cup \{l\}$.
4. *If* $\alpha = \tau$ *then* $\mathsf{fo}(P') \subseteq \mathsf{fo}(P)$.

Proof. The proof is by induction on the derivation $P \xrightarrow{\alpha} P'$. We will detail only the second and the third statement.

2. For the base case we have that rule (OUT) must be applied. In that case we have $P = k!l.P_1 \xrightarrow{k!l} P_1 = P'$. Since $\mathsf{fo}(P) = \mathsf{fo}(P_1) \cup \{l\}$ and $l \in \mathsf{fo}(P)$, by Definition 2.1.2, we may conclude the case.

   For the inductive step we have that only rules (MATCH), (RES), (PAR-L), (PAR-R) and (REP-ACT) can be applied. We detail only the case of (PAR-L). In that case $P = P_1 \mid R \xrightarrow{k!l} Q \mid R = P'$ is derived from $P_1 \xrightarrow{k!l} Q$, where the side condition of (PAR-L) is vacuously true since $\mathsf{bn}(k!l) = \emptyset$. By the induction hypothesis we get $\mathsf{fo}(Q) \cup \{l\} = \mathsf{fo}(P_1)$ and $l \in \mathsf{fo}(P_1)$, and hence by Definition 2.1.2 we derive $\mathsf{fo}(Q \mid R) \cup \{l\} = \mathsf{fo}(Q) \cup \{l\} \cup \mathsf{fo}(R) = \mathsf{fo}(P_1) \cup \mathsf{fo}(R) = \mathsf{fo}(P_1 \mid R)$, and $l \in \mathsf{fo}(P_1) \subseteq \mathsf{fo}(P_1 \mid R)$.

3. We only detail the base case, i.e., when rule (OPEN) is applied. Then, $P = (\nu l)P_1 \xrightarrow{(\nu l)k!l} P'$ is derived from $P_1 \xrightarrow{k!l} P'$. By the first part of the proof we get $l \in \mathsf{fo}(P_1)$ and $\mathsf{fo}(P') \cup \{l\} = \mathsf{fo}(P_1)$. Since $l \in \mathsf{bn}((\nu l)P_1)$, by Definition 2.1.1, we conclude $\mathsf{fo}(P') \subseteq \mathsf{fo}((\nu l)P_1) \cup \{l\}$.

□

What we can conclude from the first statement of Lemma 2.2.3 is that if a process receives a channel that is not a free output object of the process, then the received channel also cannot be a free object

output in the resulting process. The second statement of the lemma implies that if channel l is not free object output of process P, then process P cannot perform an output action with object l. Combining the two above statements we may conclude that if a process receives a channel previously not specified as an object of an output prefix in the process, then the received channel will also not appear as an object of an output prefix in the resulting process. We may show that this is preserved also by all possible evolutions of the process. To this end, we first relate the set of free channels appearing in output prefixes of a process and any execution trace. Having this in mind, the next result follows by a direct induction on the size of the trace (m).

Corollary 2.2.4 (Free output objects and traces)**.** *Let $P, P_1, \ldots, P_m$ be C_π processes. If $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} P_m$ then $\mathsf{fo}(P_m) \subseteq \mathsf{fo}(P) \cup \mathsf{bn}(\alpha_1) \cup \ldots \cup \mathsf{bn}(\alpha_m)$.*

The last corollary implies that in the C_π model, a process that receives a (fresh) channel name cannot send it later on. To precisely capture this property, we first give a precise definition of non-forwarding for all π processes.

Definition 2.2.5 (Non-forwarding property)**.** *A π process P_1 satisfies the non-forwarding property if whenever*

$$P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} P_{m+1}.$$

where $l \notin \mathsf{fn}(P_i)$ and $\alpha_i = k?l$, for some i in $1, \ldots, m-1$, then $\alpha_j \neq k'!l$, for any channel k' and any j in $i+1, \ldots, m$.

The next theorem attests that all C_π processes respect the non-forwarding property. As we will see in the next section, Definition 2.2.5 will also be used to reason about the non-forwarding of the π processes.

Theorem 2.2.6 (Non-forwarding of C_π processes)**.** *If P is a C_π process then P satisfies the non-forwarding property.*

Proof. Let $P = P_1$ be a C_π process and let

$$P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} P_{m+1}$$

We show that if $l \notin \mathsf{fn}(P_i)$ and $\alpha_i = k?l$, for some i in $1, \ldots, m-1$, then $\alpha_j \neq k'!l$, for any channel k' and any j in $i+1, \ldots, m$. Since without loss of generality we can assume all bound outputs are fresh and $l \notin \mathsf{fn}(P_i)$ (therefore $l \notin \mathsf{fo}(P_i)$), using Corollary 2.2.4 we get $l \notin \mathsf{fo}(P_j)$, for $j = i+1, \ldots, m+1$. Hence, applying Lemma 2.2.3, we can conclude $\alpha_j \neq k'!l$, for $j = i+1, \ldots, m$. □

Notice that, according to discussion preceding Definition 2.2.5, we could replace the condition $l \notin \mathsf{fn}(P_i)$ in the definition and in Theorem 2.2.6 by $l \notin \mathsf{fo}(P_i)$. Theorem 2.2.6 should come as no surprise, the C_π syntax is restricted with the goal of excluding forwarding, but nevertheless serves as a rigorous sanity check. On the other hand, if we consider the π processes it appears to be nontrivial to differentiate processes that respect the non-forwarding of fresh channel names (cf. Definition 2.2.5). To address this goal, we may rely on comparing π processes with C_π processes and on the result shown here (see Proposition 2.4.10).

2.3 Reduction semantics

In this section, we present the reduction semantics of the C_π-calculus, which again follows directly from the theory developed for the π-calculus [102]. The reduction semantics expresses only internal actions of the processes, and, as we will see, it directly corresponds to the τ transitions of the labeled transition system introduced in Section 2.2. The usefulness of the reduction semantics lays in its simplicity and elegance, in particular when used in proofs. This is also the main reason for us to introduce it in this thesis, as we will use the reduction semantics to show the properties of our encoding in Section 2.6. The

(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) $(\nu k)0 \equiv 0$

(SC-RES-EXTR) $P \mid (\nu k)Q \equiv (\nu k)(P \mid Q)$ if $k \notin \mathsf{fn}(P)$

(SC-RES-SWAP) $(\nu k)(\nu l)P \equiv (\nu l)(\nu k)P$

(SC-MAT) $[a = a]\pi.P \equiv \pi.P$

(SC-REP) $!P \equiv P \mid !P$

Table 2.3: Structural congruence.

(R-COMM) $k!l.P \mid k?x.Q \to P \mid Q\{l/x\}$

(R-PAR) $$\frac{P \to Q}{P \mid R \to Q \mid R}$$

(R-RES) $$\frac{P \to Q}{(\nu k)P \to (\nu k)Q}$$

(R-STRU) $$\frac{P \equiv P' \to Q' \equiv Q}{P \to Q}$$

Table 2.4: Reduction relation.

simplicity comes from the fact that this semantics relies on the structure congruence relation, that permits for term manipulation, allowing to single out two active prefixes willing to synchronize.

The structural congruence relation, denoted by $\equiv$, is the least binary congruence on processes that satisfies rules in Table 2.3. Rules (SC-PAR-INACT), (SC-PAR-COMM) and (SC-PAR-ASSOC) make $(\mathcal{P}, \mid, 0)$ a commutative monoid. Rule (SC-RES-INACT) shows that restricting an inactive process has no effect and that the restriction can be removed. Rule (SC-RES-EXTR) states that if the restricted channel is not specified as free in one of the branches then its scope can be confined only to the other branch, and in the other direction it allows for name extrusion

(see the example below). Rule (SC-RES-SWAP) allows swapping name restrictions, rule (SC-MAT) allows to remove the top matching if the names coincide, and rule (SC-REP) states that a copy of the replicated process can be activated in parallel with the replicated one.

The reduction relation, denoted by $\rightarrow$, is the least binary relation included in $\mathcal{P} \times \mathcal{P}$ that satisfies the rules given in Table 2.4. Rule (R-COMM) allows for two threads that are running in parallel, one sending and the other receiving on the same channel, to synchronize their actions. Rules (R-PAR) and (R-RES) allow for a reduction to take place under parallel composition and channel restriction, respectively, as prefixes involved in the reduction remain active under these constructs. Rule (R-STRU) closes the reduction relation under structural congruence, which thus allows to single out two threads ready to synchronize.

For the sake of illustration, consider process $(\nu l)k!l.0 \mid k?x.x!m.0$ given in the explanation of rule (CLOSE-L) in Section 2.2. By rule (SC-RES-EXTR) we have

$$(\nu l)k!l.0 \mid k?x.x!m.0 \equiv (\nu l)(k!l.0 \mid k?x.x!m.0)$$

Since by (R-COMM) $k!l.0 \mid k?x.x!m.0 \rightarrow 0 \mid l!m.0$ and by (R-RES)

$$(\nu l)(k!l.0 \mid k?x.x!m.0) \rightarrow (\nu l)(0 \mid l!m.0)$$

by (R-STRU) we conclude

$$(\nu l)k!l.0 \mid k?x.x!m.0 \rightarrow (\nu l)(0 \mid l!m.0)$$

As we announced, τ transitions of the action semantics coincide with reductions of the reduction semantic, up to structural congruence. This is a well-known result for the π-calculus, cf. [102] Lemma 1.4.15, and we may directly state this result for our fragment of the π-calculus.

Theorem 2.3.1 (Harmony). *$P \rightarrow Q$ if and only if there is Q_1 such that $Q_1 \equiv Q$ and $P \xrightarrow{\tau} Q_1$.*

We have shown that the set of free object names of a C_π process is preserved by τ transitions in Lemma 2.2.3. Following these lines, we can show that the same property holds in general for π-calculus processes, by extending Definition 2.1.2 to consider free object (channel) names of all π processes. In what follows we show that the set of free object names of a π process is preserved by structural congruence and is not enlarged by the reduction relation. The reduction semantics of the (sum-free) π-calculus [102] relies on the same set of rules as in Table 2.3 and Table 2.4, hence we may refer to these rules when dealing with the π processes.

Lemma 2.3.2 (Free output objects and reductions)**.** *Let P and Q be π processes.*

1. *If $P \equiv Q$ then $\mathsf{fo}(Q) = \mathsf{fo}(P)$.*
2. *If $P \to Q$ then $\mathsf{fo}(Q) \subseteq \mathsf{fo}(P)$.*

Proof. *1.* The only structural congruence rule affecting free names is (SC-MAT): $[a = a]\pi.P \equiv \pi.P$, and by the definition $\mathsf{fo}([a = a]\pi.P) = \mathsf{fo}(\pi.P)$.

2. Follows by induction on $\to$ derivation. The base case is when (R-COMM) is used. Then $k!l.P_1 \mid k?x.P_2 \to P_1 \mid P_2\{l/x\}$. Since $\mathsf{fo}(k!l.P_1 \mid k?x.P_2) = \{l\} \cup \mathsf{fo}(P_1) \cup \mathsf{fo}(P_2)$ and $\mathsf{fo}(P_1 \mid P_2\{l/x\}) \subseteq \mathsf{fo}(P_1) \cup (\mathsf{fo}(P_2) \setminus \{x\}) \cup \{l\}$, the case follows. The rest of the cases follow directly from the induction hypothesis and definition of $\mathsf{fo}(P)$, and only in the case of (R-STRU) the case *1.* of this lemma is applied.

□

The result of the last lemma is used in the proofs of correctness of the encoding of the π-calculus in the C_π-calculus, presented in Section 2.6.

2.4 Behavioral equivalence

In this section, we introduce a behavioral equivalence relation, called strong bisimilarity. Behavioral equivalences are used to answer the question: in which cases are two systems (processes) indistinguishable when inserted in the same interacting environments [36]. The strong bisimilarity is widely used as (the strictest) equivalence relation to proving properties of processes [9, 100, 101]. As we will see, the strong bisimilarity relation defined here for the C_π-calculus in Definition 2.4.1 precisely matches the one of the π-calculus in [102] Definition 2.2.1. Therefore, all the properties for strong bisimilarity relation are inherited, and we fully exploit this convenience to state and explain these properties without giving the proofs, as they can be found in [102].

The observable actions introduced in Section 2.2 give a basis to define the strong bisimilarity. Intuitively, two processes are called strongly bisimilar if a game can be played between them: each labeled action of one process can be reproduced by the other process (and inversely), and the resulting processes can be again paired under the same conditions. Hence, this relation pairs processes exhibiting the same behavior.

Definition 2.4.1 (Strong bisimilarity)**.** *The strong bisimulation is a symmetric relation $\mathcal{R}$ over processes that satisfies*

$$\textit{if } P\mathcal{R}Q \textit{ and } P \xrightarrow{\alpha} P', \textit{ where } \mathsf{bn}(\alpha)\cap\mathsf{fn}(Q) = \emptyset, \textit{ then } Q \xrightarrow{\alpha} Q' \textit{ and } P'\mathcal{R}Q'.$$

The strong bisimilarity, denoted $\sim$, is the largest strong bisimulation, i.e., it includes all strong bisimulation relations over processes.

For instance, $(\nu k)k!l.P \sim 0$, for any process P, since the left-hand side process cannot exhibit the output action specified in the active prefix because of the side condition of rule (RES). Now consider processes

$$(\nu k)((\nu l)k!l.0 \mid k?x.[x = m]\pi.0) \quad \text{and} \quad (\nu k)((\nu l)k!l.0 \mid k?x.0) \quad (2.1)$$

We may show that these two processes are also bisimilar, since

$$\begin{aligned}\mathcal{R} \;=\; \{&((\nu k)((\nu l)k!l.0 \mid k?x.[x=m]\pi.0), (\nu k)((\nu l)k!l.0 \mid k?x.0))\\ &((\nu k)(\nu l)(0 \mid [l=m]\pi.0), (\nu k)(\nu l)(0 \mid 0))\}\end{aligned}$$

is a strong bisimulation relation. Notice that in process $[l = m]\pi.0$ the received channel l is matched with m, and since these two names do not coincide the process cannot exhibit the action specified in prefix π, because of rule (MATCH). Notice also that l is received involving name extrusion since it is bound in the left branch. We now state some of the fundamental properties of the strong bisimilarity, for which the respective proofs can be found in [102].

Proposition 2.4.2 (Equivalence)**.** *Strong bisimilarity is an equivalence relation.*

Proposition 2.4.3 (Structural congruence)**.** *If $P \equiv Q$ then $P \sim Q$.*

Proposition 2.4.4 (Non-input congruence)**.**

(a) If $P \sim Q$ then

1. *$P \mid R \sim Q \mid R$;*
2. *$(\nu k)P \sim (\nu k)Q$;*
3. *$!P \sim !Q$;*
4. *if $\pi = [b_1 = c_1] \ldots [b_n = c_n]a!k$ or $\pi = a!k$ then $\pi.P \sim \pi.Q$.*

(b) If for any channel k it is the case that $P\{k/x\} \sim Q\{k/x\}$ holds and $\pi = [b_1 = c_1] \ldots [b_n = c_n]a?x$ or $\pi = a?x$ then $\pi.P \sim \pi.Q$.

The fact that strong bisimilarity is a non-input congruence is inherited from the π-calculus. It is known that only for some specific sub-calculi of the π-calculus the strong bisimilarity is preserved by the input construct [16, 52, 53, 102]. Strong bisimilarity is a non-input congruence in the presence of matching [17], as it is the case in the

C_π-calculus. For instance, $[x = m]\pi.0 \sim 0$, as we already noted in the example above, but

$$a?x.[x = m]\pi.0 \not\sim a?x.0$$

since it can be the case that the received name is m, and then the action specified by π is activated, while the other process terminates no matter what name is received.

We now present one behavioral equality that is specific for the C_π-calculus. We have shown that the processes given in (2.1) are bisimilar, and we commented there that if the received name is new to the process then matching it with any name of the process will always fail. What specifically holds in the C_π-calculus is that if a process sends a bound name, after which receives a name and matches it with the name previously sent (like in $(\nu l)k!l.k?x.[x = l]\pi.P$), the matching will always fail. The reason is the non-forwarding property of C_π processes: a process that receives l will never be able to send it later on. We exploit this feature in the next result to directly represent the creation of closed domains for channels, that resembles the creation of secure channels with statically determined scope.

Proposition 2.4.5 (Closed domains for channels)**.** *For any C_π process P, channel m and prefix π, the following equality holds*

$$(\nu k)((\nu l)k!l.m?y.[y = l]\pi.0 \mid k?x.P) \sim (\nu k)((\nu l)k!l.m?y.0 \mid k?x.P)$$

Proof. The proof is by coinduction on the definition of the strong bisimulation, by showing that the relation

$$\begin{aligned}\mathcal{R} \;=\; \{&\big((\nu k)((\nu l)k!l.m?y.[y=l]\pi.0 \mid k?x.P), (\nu k)((\nu l)k!l.m?y.0 \mid k?x.P)\big),\\ &\big((\nu k)(\nu l)(m?y.[y=l]\pi.0 \mid Q), (\nu k)(\nu l)(m?y.0 \mid Q)\big),\\ &\big((\nu l)(m?y.[y=l]\pi.0 \mid Q), (\nu l)(m?y.0 \mid Q)\big),\\ &\big((\nu k)(\nu l)([n=l]\pi.0 \mid Q), (\nu k)(\nu l)(0 \mid Q)\big),\\ &\big((\nu l)([n=l]\pi.0 \mid Q), (\nu l)(0 \mid Q)\big),\\ &\big((\nu k)(\nu l)(\nu n)([n=l]\pi.0 \mid Q), (\nu k)(\nu l)(\nu n)(0 \mid Q)\big),\\ &\big((\nu l)(\nu n)([n=l]\pi.0 \mid Q), (\nu l)(\nu n)(0 \mid Q)\big)\\ &\mid \text{for all } n, m \in \mathcal{C}, \text{ such that } n \neq l,\\ &\quad \text{and all processes } P \text{ and } Q, \text{ such that } l \notin \mathsf{fo}(Q)\,\}\end{aligned}$$

is a strong bisimulation, hence, contained in strong bisimilarity (i.e., $\mathcal{R} \subseteq \sim$).

We show that each action of one process can be mimicked by the other process in the pair in $\mathcal{R}$, leading to processes that are again in relation $\mathcal{R}$. Let the process in the first pair

$$(\nu k)((\nu l)k!l.m?y.[y=l]\pi.0 \mid k?x.P) \xrightarrow{\alpha} P'$$

Then, since actions of the starting process can only be actions of its two branches, we conclude that either $\alpha = (\nu l)k!l$ or $\alpha = k?n$ or it is the synchronization of these two actions, in which case $\alpha = \tau$. We reject the first two options since the subject of the action is bound in the starting process and by rule (RES) it cannot be observed outside of the process. Hence, we conclude $\alpha = \tau$ and $P' = (\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid P\{l/x\})$. Then, by applying (OUT), (OPEN), (IN), (CLOSE-L) and (RES), respectively, we observe

$$(\nu k)((\nu l)k!l.m?y.0 \mid k?x.P) \xrightarrow{\tau} (\nu k)(\nu l)(m?y.0 \mid P\{l/x\})$$

and since $l \notin \mathsf{fn}(P)$ and x cannot appear as an object in the prefixes in P we conclude $l \notin \mathsf{fo}(P\{l/x\})$. Hence, we have

$$\big((\nu k)(\nu l)(m?y.[y=l]\pi.0 \mid P\{l/x\}), (\nu k)(\nu l)(m?y.0 \mid P\{l/x\})\big) \in \mathcal{R}$$

The symmetric case is analogous.

Now let us consider processes in the second pair of $\mathcal{R}$. If

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{\alpha} P'$$

then the observable α can originate from one of the branches or from their synchronization.

—*Left branch:* If the observable originate from the left branch, then $\alpha = m?n$, and by (IN), (PAR-L) and (RES)

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{m?n} (\nu k)(\nu l)([n = l]\pi.0 \mid Q)$$

where, by the side condition of (RES) we conclude $n \notin \{k, l\}$. In the same we derive

$$(\nu k)(\nu l)(m?y.0 \mid Q) \xrightarrow{m?n} (\nu k)(\nu l)(0 \mid Q)$$

and $\big((\nu k)(\nu l)([n = l]\pi.0 \mid Q), (\nu k)(\nu l)(0 \mid Q)\big) \in \mathcal{R}$ holds.

—*Right branch:* If the action originates from the right branch, i.e., from $Q \xrightarrow{\alpha} Q'$, we distinguish two cases:

(i) if the derivation is carried out using rules (PAR-R) and (RES) we have that

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{\alpha} (\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q')$$

where, since $l \notin \mathsf{fo}(Q)$, by Lemma 2.2.3 we conclude $l \notin \mathsf{fo}(Q')$. Then, by the same rules

$$(\nu k)(\nu l)(m?y.0 \mid Q) \xrightarrow{\alpha} (\nu k)(\nu l)(m?y.0 \mid Q')$$

and $\big((\nu k)(\nu l)(m?y.[n = l]\pi.0 \mid Q'), (\nu k)(\nu l)(m?y.0 \mid Q')\big) \in \mathcal{R}$ holds.

(ii) if the derivation is carried out using rules (PAR-R), (RES) and (OPEN) we have that

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{(\nu k)\alpha} (\nu l)(m?y.[y = l]\pi.0 \mid Q')$$

then $l \notin \mathsf{n}(\alpha)$. Notice that the scope of channel l cannot be opened this way since $l \notin \mathsf{fo}(Q)$. Hence, process Q cannot perform output action with object l. Then by the same rules

$$(\nu k)(\nu l)(m?y.0 \mid Q) \xrightarrow{(\nu k)\alpha} (\nu l)(m?y.0 \mid Q')$$

and, again, $\big((\nu l)(m?y.[n = l]\pi.0 \mid Q'), (\nu l)(m?y.0 \mid Q')\big) \in \mathcal{R}$ holds.

—*Synchronization of branches:* We again distinguish two cases:

(i) if the derivation follows from

$$m?y.[y = l]\pi.0 \xrightarrow{m?n} [n = l]\pi.0 \qquad \text{and} \qquad Q \xrightarrow{m!n} Q'$$

where we can make the same observation on Q as before to conclude that $l \neq n$, and the derivation relies on rules (COMM-R) and (RES) hence

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{\tau} (\nu k)(\nu l)([n = l]\pi.0 \mid Q')$$

Then, considering $m?y.0 \xrightarrow{m?n} 0$, and the same rules as above we have that

$$(\nu k)(\nu l)(m?y.0 \mid Q) \xrightarrow{\tau} (\nu k)(\nu l)(0 \mid Q')$$

and $\big((\nu k)(\nu l)([n = l]\pi.0 \mid Q'), (\nu k)(\nu l)(0 \mid Q')\big) \in \mathcal{R}$.

(ii) if the derivation follows from

$$m?y.[y = l]\pi.0 \xrightarrow{m?n} [n = l]\pi.0 \qquad \text{and} \qquad Q \xrightarrow{(\nu n)m!n} Q'$$

where as before we can assume $l \neq n$, and derivation relies on rules (CLOSE-R) and (RES) hence

$$(\nu k)(\nu l)(m?y.[y = l]\pi.0 \mid Q) \xrightarrow{\tau} (\nu k)(\nu l)(\nu n)([n = l]\pi.0 \mid Q')$$

then using $m?y.0 \xrightarrow{m?n} 0$, we may observe

$$(\nu k)(\nu l)(m?y.0 \mid P) \xrightarrow{\tau} (\nu k)(\nu l)(\nu n)(0 \mid Q')$$

and $\big((\nu k)(\nu l)(\nu n)([n = l]\pi.0 \mid Q'), (\nu k)(\nu l)(\nu n)(0 \mid Q')\big) \in \mathcal{R}$.

The symmetric cases and the rest of the pairs from $\mathcal{R}$ are analogous. For the rest of the pairs note that in all of them the left branch that appears on the left-hand side of the pairs we have $[n = l]\pi.0$, where $n \neq l$, and hence, it exhibits no transitions and is observationally equivalent to the inactive process 0, which appears in the left branch in the right-hand side of the pairs.

□

In both processes related by strong bisimilarity in Proposition 2.4.5 the left thread creates a new channel l and sends it to the right thread over a channel k that is known only to the two threads. The bisimilarity shows that then the channel l cannot be received afterwards in the left thread. As discussed above, this is a consequence of the non-forwarding property of C_π processes. Furthermore, we may show that channel l will not be exchanged even between sub-processes of process $P\{l/x\}$ (as there would exist a sub-process that violates the non-forwarding property). Hence, in this constellation, channel l will be sent only once and afterwards have a "static" nature since then it can be used only for sending and receiving along the channel and cannot be sent itself. We say that the two processes given in the proposition determine a closed domain for channel l.

2.4.1 Strong barbed equivalence

For the purpose of the results in Section 2.6, we introduce a behavioral equivalence that relies on the reduction relation instead of the labeled transitions. To this end, we introduce the notion of barbs and strong barbed equivalence. Again we fully exploit the theory developed for the π-calculus in [102], where all the details and poofs can be found.

Definition 2.4.6 (Barbs)**.** *For each channel k and process P, we say that $P \downarrow_{\overline{k}}$ holds if P can perform an output action with subject k, and $P \downarrow_k$ holds is P can perform an input action with subject k.*

Based on the definition of barbs, we define the strong barbed bisimilarity.

Definition 2.4.7 (Strong barbed bisimilarity)**.** *Strong barged bisimilarity is the largest symmetric relation $\dot{\sim}$ such that if $P \dot{\sim} Q$ then*

1. *if $P \downarrow_k$ then $Q \downarrow_k$,*
2. *if $P \downarrow_{\overline{k}}$ then $Q \downarrow_{\overline{k}}$,*
3. *if $P \to P'$ then $Q \to Q'$ and $P' \dot{\sim} Q'$.*

The strong barbed bisimilarity is not a congruence relation, it is not even preserved by parallel composition. For instance,

$$k!l.n!m.0 \dot{\sim} k!l.0$$

while $k!l.n!m.0 \mid k?x.0$ and $k!l.0 \mid k?x.0$ are not strong barbed bisimilar since after the reduction the first process has barb n. The relation we are interested in, and that coincides with the strong bisimilarity, closes the strong barbed bisimilarity under parallel composition contexts.

Definition 2.4.8. *Two processes P and Q are strong barbed equivalent, $P \simeq Q$, if for any R holds $P \mid R \dot{\sim} Q \mid R$.*

The proof of the next theorem can be found in [102] Theorem 2.2.9(1).

Theorem 2.4.9 (Strong characterization)**.** *$P \simeq Q$ if and only if $P \sim Q$.*

As noted, we are going to use strong barbed equivalence formally in the proofs in Section 2.6, where we are going to deal with the reduction relation. However, notice that the strong bisimilarity relation offers a much more tractable technique for proving that two processes are related, since relating two processes with strong barbed equivalence requires that they have to be tested when composed in parallel with an infinite number of processes (actually, all of them).

2.4.2 A characterization of the non-forwarding π processes

We have seen in Theorem 2.2.6 that C_π processes satisfy Definition 2.2.5: if the received channel is fresh to the process it will not be forwarded. Considering π processes, the non-forwarding property, in general, does not hold, so this kind of privacy property is hard to guarantee. Therefore, it may be worthwhile to develop a method for distinguishing the π-calculus processes that satisfy the non-forwarding property relying on the C_π-calculus developed here. If we focus only on the π processes that are closed, i.e., do not have free names, we may notice that these processes vacuously satisfy non-forwarding. Furthermore, we may find examples of π processes that are not part of the C_π syntax and not closed but still respect this property. For instance, π process

$$k?x.(\nu l)(l!x.0 \mid l?y.0)$$

that receives a channel on k and then sends the received channel on l. However, since l is restricted in the process, the received channel will be exchanged only between the components of the process and will not be sent to the process environment. However, statically characterizing the non-forwarding by considering that forwarding is performed only on restricted channels is not possible, as restricted channels can be opened. For instance, we may notice that π process $k?x.(\nu l)(k!l.l!x.0 \mid l?y.0)$ does not satisfy the non-forwarding property.

Hence, differentiating a non-forwarding π process may not be an easy task. Here, we propose a method towards the solution of this problem. One may observe that if a π process P is bisimilar to some C_π process Q, then P must satisfy the non-forwarding property (Definition 2.2.5). This is the idea of our next proposition. We remark that since we are dealing with sum-free C_π process, in the proposition we consider also only sum-free π terms. Also, in the next proposition we use Definition 2.4.1 extended here to consider the strong bisimilarity relation over all π processes (as in [102]).

Proposition 2.4.10 (Non-forwarding π processes)**.** *Let P be a π process. If there is a C_π process Q, such that $P \sim Q$, then P satisfies the non-forwarding property.*

Proof. Let $P_1 = P$ be a (sum-free) π process and let $P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} P_{m+1}$. Let us fix $i \in \{1, \ldots, m-1\}$, and assume $l \notin \mathsf{fn}(P_i)$ and $\alpha_i = k?l$. Since without loss of generality we can assume all bound outputs are fresh, we get $\alpha_j \neq (\nu l)k'!l$, for all $j = i+1, \ldots, m$, directly. In addition to the first assumption, let us assume there is $j \in \{i+1, \ldots, m\}$ such that $\alpha_j = k'!l$. Since $P_1 \sim Q_1$ (where $Q_1 = Q$), we conclude there are C_π processes $Q_2, \ldots, Q_{m+1}$ such that

$$Q_1 \xrightarrow{\alpha_1} Q_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} Q_{m+1}$$

and $P_n \sim Q_n$, for all $n = 1, \ldots, m+1$, where $Q_i \xrightarrow{k?l} Q_{i+1}$ and $Q_j \xrightarrow{k'!l} Q_{j+1}$. We now distinguish two cases.

1. If $l \notin \mathsf{fn}(Q_i)$ then we get a direct contradiction with Theorem 2.2.6.

2. If $l \in \mathsf{fn}(Q_i)$, we choose a fresh channel l' and a substitution σ that is defined only on channel l and maps it to l'. Then, from $P_j \xrightarrow{\alpha_j} P_{j+1}$, by consequitive application of [102] Lemma 1.4.8, we conclude $(P_j)\sigma \xrightarrow{(\alpha_j)\sigma} (P_{j+1})\sigma$, for all $j = i+1, \ldots, m$. Since $l \notin \mathsf{fn}(P_i)$ we get $(P_i)\sigma = P_i$. Now from

   $$P_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{i-1}} P_i \xrightarrow{(\alpha_i)\sigma} (P_{i+1})\sigma \xrightarrow{(\alpha_{i+1})\sigma} \ldots \xrightarrow{(\alpha_m)\sigma} (P_{m+1})\sigma,$$

   and $P_1 \sim Q_1$, we again conclude there are C_π processes $Q_2, \ldots, Q_{m+1}$ such that

   $$Q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{i-1}} Q_i \xrightarrow{(\alpha_i)\sigma} Q_{i+1} \xrightarrow{(\alpha_{i+1})\sigma} \ldots \xrightarrow{(\alpha_m)\sigma} Q_{m+1},$$

   where $P_j \sim Q_j$, for all $j = 1, \ldots, i$ and $(P_j)\sigma \sim Q_j$, for all $j = i+1, \ldots, m+1$. Since l' has been chosen to be a fresh channel,

we get $l' \notin \mathsf{fn}(Q_i)$, and since $Q_i \xrightarrow{(k)\sigma?l'} Q_{i+1}$ and $Q_j \xrightarrow{(k')\sigma!l'} Q_{j+1}$, we fall into the first case, and hence, we again get contradiction with Theorem 2.2.6.

□

Proposition 2.4.10 opens a number of possible direction for improvements and modifications. For instance, Proposition 2.4.10 shows that if a π process is bisimilar to a C_π process then it does respect non-forwarding, but it does not give an algorithm for deriving the respective C_π process. Also, we can relax the definition of the non-forwarding, to consider processes that do not forward names received only on some predefined set of channels. Another point is answering the question if a π process respects the non-forwarding property if and only if is bisimilar to a C_π process. Such investigations are left for future work.

2.5 Examples

This section presents several examples as a showcase of the possible usefulness of the C_π-calculus. We may notice that a C_π process that creates a channel always keeps for himself the capability of sending the channel while the other processes that learn about the channel by receiving it can only use the channel to communicate on it. Hence, between the processes in which a channel is known, we may distinguish

- *administrators*, the processes that create the channel: these have the capabilities of communicating on the channel and also sending the channel;
- *users*, the processes that at some point have received the channel: these have only the capabilities of communicating on the channel.

As the capability of sending a channel is never transferred between processes, we can conclude that none of the channel users can become

an administrator for the channel, and furthermore, only administrators can engage new users (but not administrators) by sending the channel.

Let us consider the process given at the beginning of this chapter

$$((\nu session)chn!session.Alice) \mid chn?x.Bob \mid forward?y.Carol$$

adapted here by considering *Alice*, *Bob* and *Carol* to be C_π processes. Then, the administrator of *session* is the process in the scope of the name restriction, while *Bob* becomes a user after the initial synchronization. In our model, *Bob* cannot send *session* to a third party afterwards. Considering that *Bob* wants to send to *Carol* one endpoint of channel *session*, he first needs to notify the administrator, in this case *Alice*, by connecting her with *Carol*. Hence, we can have

- $Bob = channel!forward.0$, where *Bob* sends to *Alice* channel *forward* and then terminates,
- $Alice = channel?y.y!session.Alice'$, where *Alice* receives the channel from *Bob* and decides to send *session* along the received channel, and
- finally, *Carol* can receive *session* along channel *forward* directly from *Alice*.

This example shows that in the C_π any channel extrusion first has to be "approved" by the channel administrators.

2.5.1 Authentication

We stipulated that specifying $(\nu session)Alice$ in the C_π-calculus defines *Alice* as the administrator for channel *session*. Administrator (process *Alice*) can extrude the scope of the channel (*session*) by sending it, but the receiving processes will only become users and never administrators for the received channel. The administrator attribute is something that remains with the creator of the channel and is invariant to process evolution. This locality can be used for authentication of

processes as follows. Assume process *Bob* is a user of channel *session*. Then, *Bob* can test the other process with whom he is communicating on *session* to determine if the other process is an administrator of the channel. For example, we can specify

$$Bob = (\nu privateChn)session!privateChn.privateChn?x.[x = session].Bob'$$

where process *Bob* first establishes a private connection with the other process listening on *session* by sending fresh *privateChn*. Afterwards, *Bob* specifies an input and then matches the received name with *session*. If the received channel is *session* then the other process, say *Alice*, has proven to *Bob* that she is in the domain where *session* was created.

A similar scenario can be used by two administrators of the channel to test each other. For instance, we can have process

$$chn!session.chn?y.[y = session]\pi.Alice \mid chn?x.[x = session]chn!session.Dylan$$

where the two threads first perform a kind of authentication scheme and only then activate $\pi.Alice$ and *Dylan* and their possible interactions. First, the right thread receives a name and matches it with *session*, i.e., we get

$$chn?y.[y = session]\pi.Alice \mid [session = session]chn!session.Dylan'$$

where $Dylan' = Dylan\{session/x\}$, and since the received name is *session* then he sends the channel by himself. The left thread receives the name and also matches it with *session*, i.e., we obtain process $[session = session]\pi'.Alice' \mid Dylan'$, where $\pi'.Alice' = (\pi.Alice)\{session/y\}$. Since the received name is *session*, $Alice'$ continues interacting with $Dylan'$. After this testing, both threads have proven to be administrators for the channel.

2.5.2 Modeling groups and name hiding

The channel creation construct of the π-calculus introduces a notion of a private resource. This private resource can be shared with other

processes through scope extrusion, and this is a very important aspect of the π-calculus. On the other hand, extruding the scope means exposing the private resource to others. We have seen that the C_π-calculus does not prevent the scope extrusion, but it simply confines (with the process that creates the channels) this capability.

Several process models have been proposed for controlling the channel sharing in the π-calculus. The paper by Cardelli et al. [26], extends the π-calculus syntax with the construct for group creation. A type is assigned to each channel, specifying to which group that channel belongs. The operational semantics instrumented with the type information ensures that channels of a group cannot be acquired by a process outside the scope of the group. Furthermore, their typing discipline provides that grouped channels are never communicated on open channels. The work of Giunti et al. [46] extends the syntax of the π-calculus with a construct called hide. The hide construct has similar properties as channel restriction, but it is more rigorous since construct hide does not allow for scope extrusion. Hence, a name specified in the hide construct has a predetermined scope.

Determining the scope of a channel can be achieved in C_π-calculus directly, as already hinted in Proposition 2.4.5. For instance, if the channel creation is placed in a separate thread and then sent to a process, as in

$$(\nu chn)((\nu session)chn!session.0 \mid chn?x.Group_x)$$

then, after the initial synchronization channel *session* will have the final scope determined by $Group_{session}$. This is simply a consequence of $Group_{session}$ being a C_π-process, and not being capable to send channels that were received. Furthermore, since the resulting process is $(\nu chn)(\nu session)(0 \mid Group_{session})$, we may notice that channel *session* actually loses its mobility altogether and behaves more like a CCS channel [68]. Notice that the administrator is now the inactive process so the capability of sending the name is lost. Hence, the scope of a channel can be permanently restricted this way, but differs when

comparing to groups and name hiding where a channel can be communicated inside a predefined scope since in our case the channel cannot be communicated in any process.

We may try to get closer to represent groups and name hiding by combining the channel creation and authentication from Section 2.5.1. The capability of sending a channel is always kept local with the administrator of the channel. We may use this fact to localize the final scope for a protected channel. For instance, we may say that in process

$$(\nu group)((\nu session)Alice \mid Bob)$$

channel *session* can be received only by administrators of channel *group*, which represents the group which may have access to channel *session*. To this end, each time channel *session* is to be sent by process *Alice*, she must authenticate the receiving process as an administrator for channel *group*, by specifying

$$(\nu privateChn)chn!privateChn.privateChn?x.[x = group]privateChn!session$$

This way we can ensure channel *session* will be received only by a process that originates from $(\nu session)Alice \mid Bob$. Comparing again with groups and name hiding we see that now we do have channel mobility only inside a predefined scope. However, in the presence of attackers (that are not C_π processes), this mechanism needs to be further strengthened to protect also the channels that represent groups, such as channel *group*. We leave this exploration and formalization of the relationship between C_π and models with groups and name hiding for future work.

2.5.3 Open-ended groups

All examples of restricting channel sharing we have considered so far have used a predefined scope for a protected channel. This is also the case with works with groups [26] and name hiding [46], in which the scope of channels that are confidential are statically prescribed.

However, we may notice that sometimes protected resources need to be shared in open-ended environments and that the above limitation can be considered too restrictive. We believe that the C_π-calculus, with its administrator-user hierarchy, offers a good base to reason on open-ended groups. For instance, consider process

$$(\nu session)(chn!session.Alice \mid Bob) \mid chn?x.Carol$$

where the leftmost thread is the administrator for channel *session*, *Bob* is a user, and *Carol* does not know name *session*. The administrator (i.e., the process that created the group), can send the name of the group to other processes, and hence can engage new users to the group, in our example obtaining

$$(\nu session)(Alice \mid Bob \mid Carol')$$

where $Carol' = Carol\{session/x\}$. Notice that the control of joining new users to the group is handled by an administrator, as users cannot themselves engage other members to the group.

2.6 Encoding forwarding

Even though the specification power of C_π-calculus is clearly different from the π-calculus, this is not the case when comparing the expressiveness of the computational models. In this section, we show how the π-calculus can be implemented (encoded) in the C_π-calculus which therefore informs on the expressive power of the language. The section is divided into three parts. The first part informally introduces the idea of the encoding, Section 2.6.1 formally presents the encoding and Section 2.6.2 provides the proof of the Operational correspondence result.

As the number of different process models has grown, comparing these various models in a systematic way has been recognized as an important aspect of the research in the field [81]. There are

a number of important results among which are: encoding the λ-calculus into the π-calculus [69], comparing various subcalculi of the π-calculus [15, 55, 75], comparing different process calculi and some separation results [25, 33, 37, 47, 62, 65, 78, 79, 98]. We start this section with an informal presentation of the idea, and afterwards, we present an encoding from the π-calculus into our calculus and we attest the encoding is valid by showing the operational correspondence result, following the criteria given in [47].

In order to represent our encoding in a more compact way we use a fragment of the polyadic version of our calculus. The only difference of the syntax of the polyadic C_π with the syntax introduced in Section 2.1 is that the output $(k!l)$ and the input $(k?x)$ prefix can have as object a tuple of names. Hence, in polyadic C_π we have

$$\pi \ ::= \ k!(l_1, \ldots, l_n) \ \mid \ k?(x_1, \ldots, x_n) \ \mid \ [a = b]\pi$$

When comparing the reduction semantics of the polyadic C_π with the reduction semantic introduced in Section 2.3, the only difference is that now output and input involved in the reduction may have as objects tuples of channels and variables, respectively, and that in order for reduction to take place these two tuples have to be of the same size (arity). Synchronization of actions of different arity is considered an error, known as arity mismatch (cf. the polyadic π-calculus [70]). However, polyadicity in this work is more syntactic sugar, as the fragment of the polyadic C_π-calculus that we are going to use here for the purpose of the encoding of the monadic π-calculus, does not have to deal with arity mismatches and can be represent in the monadic C_π following expected lines (we will return to this point later).

As we noted, the (monadic) C_π-calculus differs from the (monadic) π-calculus in the restriction that input variables cannot be specified as objects of output prefixes. For instance, consider process given at the beginning of this chapter

$$chn!session.Alice \mid chn?x.forward!x.Bob \mid forward?y.Carol \qquad (2.2)$$

in which the leftmost thread sends *session* on *chn* to the thread in the middle. Then the thread in the middle forwards the received channel to the rightmost thread on *forward*. This π process is clearly not a C_π process. We may try to represent the forwarding of channel *session* of the middle thread in the C_π-calculus using the following idea:

- create a process dedicated for sending channel *session*, the process we call *handler of channel session*,
- whenever channel *session* is sent it is sent together with a special channel that allows to communicate with the handler, and
- when a process that received name *session* wants to forward the name, it asks the respective handler to carry out the communication identifying on which channel *session* is to be sent.

Hence, our first attempt to represent the π processes (2.2) in (polyadic) C_π is to

- in parallel with encoded processes from (2.2) add the handler process
$$H = handler?x.x!(session, handler).0$$
that on a special channel *handler* receives a channel and then sends *session* and *handler* on the received channel,
- *Alice* sends channel *session* together with *handler*, so that the receiving process (*Bob*) can address the handler process, i.e., $chn!session.Alice$ is represented as
$$A = chn!(session, handler).Alice'$$
- *Bob* now receives the pair $(session, handler)$ and then creates a private connection between the receiving process (*Carol*) with the handler process by sending to both of them a fresh channel *private*. Hence, $chn?x.forward!x.Bob$ is represented as
$$B = chn?(x, m_x).(\nu private)forward!private.m_x!private.Bob'$$

- *Carol* first receives the private channel from *Bob* on channel *forward* and then receives the pair $(session, handler)$ from the handler process. Hence, process $forward?y.Carol$ is represented as
$$C = forward?z.z?(y, m_y).Carol'$$

Therefore, the process in (2.2) is represented with $A \mid B \mid C \mid H$. In the first reduction step of the process the pair $(session, handler)$ is sent from A (*Alice*) to B (*Bob*) leading to

$$Alice' \mid (\nu private)forward!private.handler!private.Bob'' \mid C \mid H$$

where $Bob'' = Bob'\{session/x\}\{handler/m_x\}$. Then, *Bob* connects processes C (*Carol*) and H (the handler process) in two steps, leading to

$$Alice' \mid (\nu private)(Bob'' \mid private?(y, m_y).Carol' \mid private!(session, handler).0)$$

after which *Carol* can finally receive channel *session* (together with channel *handler*) from the handler process

$$Alice' \mid (\nu private)(Bob'' \mid Carol'' \mid 0)$$

where $Carol'' = Carol'\{session/y\}\{handler/m_y\}$.

There are two additional points we need to take care when following the idea introduced above:

- forwarding a channel can be required an indefinite (possibly infinite) number of times, hence the handler must be repeatedly available to answer forwarding requests,

- for the sake of a faithful representation of the forwarding behavior we need to ensure that forwarder and (final) receiver agree on when the forwarding has taken place since a direct synchronization ensures this.

To address the last two issues, we can refine the $A \mid B \mid C \mid H$ representation of the process in (2.2), by specifying the handler process to be replicated, i.e.,

$$H = !handler?x.x!(session, handler).0$$

and processes B and C to be

$$B = chn?(x, m_x).(\nu private, lock) forward!(private, lock).m_x!private.lock!.Bob'$$

and

$$C = forward?(z, z').z?(y, m_y).z'?z''.Carol'$$

wherein the synchronization on channel *forward* together with channel *private* another fresh channel *lock* is sent from B to C. This channel is used only after C has received $(session, handler)$ from the handler process, to signal that forwarding is completed and to unlock processes Bob'' and $Carol''$ simultaneously upon the synchronization.

2.6.1 The encoding

This section introduces the encoding formally. For the purpose of introducing the renaming policy, used by the encoding, we define three disjoint infinite sets $\mathcal{N}_\pi$, $\mathcal{N}_\varphi$ and $\mathcal{N}_{res}$, all three containing infinite number of channel and variable names, such that the union of these three sets is the set of C_π names $\mathcal{N}$. Here, $\mathcal{N}_\pi$ is the set of all π-calculus names, $\mathcal{N}_\varphi$ is the set of names introduced by the renaming policy φ, and $\mathcal{N}_{res}$ is the set of reserved names (used by the translation function). We let $m_a, m_k, m_x, \ldots$ range over $\mathcal{N}_\varphi$, and $e_1, e_2, y, z, z', \ldots$ range over $\mathcal{N}_{res}$. Notice that, here we assume the set of π-calculus names $\mathcal{N}_\pi$ is strictly contained in the set of C_π names $\mathcal{N}$. Such assumption is possible since $\mathcal{N}_\pi$ and $\mathcal{N}$ (and also $\mathcal{N}_\varphi$ and $\mathcal{N}_{res}$) are infinite countable sets.

Formally, our encoding is a pair $(\llbracket\cdot\rrbracket, \varphi_{\llbracket\,\rrbracket})$, where $\llbracket\cdot\rrbracket$ is a translation function and $\varphi_{\llbracket\,\rrbracket}$ is a renaming policy, cf. [47]. The translation

$$
\begin{aligned}
[\![(\nu k)P]\!] &= (\nu k)(\nu m_k)([\![P]\!] \mid !m_k?x.x!(k, m_k).0) \\
[\![[\tilde{c} = \tilde{d}]a!b.P]\!] &= (\nu e_1)(\nu e_2)[\tilde{c} = \tilde{d}]a!(e_1, e_2).m_b!e_1.e_2!e_1.[\![P]\!] \\
[\![[\tilde{c} = \tilde{d}]a?x.P]\!] &= [\tilde{c} = \tilde{d}]a?(y, z).y?(x, m_x).z?z'.[\![P]\!] \\
[\![P_1 \mid P_2]\!] &= [\![P_1]\!] \mid [\![P_2]\!] \\
[\![!P]\!] &= ![\![P]\!] \\
[\![0]\!] &= 0
\end{aligned}
$$

Table 2.5: Encoding of π processes into C_π processes

function is a mapping from the π-calculus, the source terms, into the C_π-calculus, the target terms. The translation function $[\![\cdot]\!]$ relies on the renaming policy $\varphi_{[\![\,]\!]}$, that is a function $\varphi_{[\![\,]\!]} : \mathcal{N}_\pi \to \mathcal{N}_\pi \times \mathcal{N}_\varphi$, mapping each name a of the source language into a pair of names (a, m_a), where for two pairs (a, m_a) and (b, m_b) if $a \neq b$ then also $m_a \neq m_b$.

Table 2.5 introduces the translation function inductively on the source terms. The first rule in the table translates a name restriction. The resulting process specifies the encoding of the original scoped process $[\![P]\!]$ together with the handler for the restricted channel, both scoped with the restriction for both the channel k and associated channel name m_k (via the renaming policy). The handler process is repeatedly available to be invoked (on m_k) and it receives a channel along which it outputs the channel k together with the "access point" of the handler (m_k). By sending m_k, we make it possible for the process that receives (see the rule for input) to be able to afterwards directly communicate with the handler for k.

The output process is encoded as a process that creates two fresh channels e_1 and e_2, then both channels are sent to the receiving process, on the same name a as in the original process. Then, e_1 is also sent to the handler of name b, thus allowing to create a private

connection between the receiving and handler process. Here, we use $[\tilde{c} = \tilde{d}]\pi$ to abbreviate $[c_1 = d_1] \dots [c_n = d_n]\pi$ or π when the sequence of matchings is empty. Channel e_2 is used only in the synchronization mechanism to ensure that both continuations of the sending and the receiving processes are activated only after the forwarding mechanism is completed, so as to mimic the original behavior (cf. channel *lock* in the example above). We remark that here names e_1 and e_2 are taken to be from the reserved set of names $\mathcal{N}_{res}$, and hence cannot appear as free in $[\![P]\!]$. The same assumption is made for names y, z and z' in the rule for input, hence these names cannot appear as free in the continuation $[\![P]\!]$.

The input process is encoded as a reception of a pair of channels (cf. the encoding of the output prefixed process). When a pair of names (e_1, e_2) is received (from an output process) then the encoding of the input process proceeds by receiving a pair of names (from a handler process), that are then substituted in the continuation process. After that, a channel is received on channel e_2. The last reception is used only for activating the continuations of the output and the input processes (as mentioned above). The encoding is a homomorphism elsewhere.

We may now notice that for the purpose of encoding monadic π-calculus in C_π, the polyadicity is used in a controlled way, as the only place where monadic communications can take place are the invocation of the handler on channel m_l (from set $\mathcal{N}_\varphi$) and unlocking the continuation of the output and input processes on private channel e_2, and the rest of the actions are of arity two. Furthermore, all the actions of arity two are either conducted on a private channel or carrying private channel(s) as a message. This implies that we can represent the behavior of each of the polyadic prefixes by monadic ones straightforwardly. For example, $a!(e_1, e_2)$ in the output process can be represented by $a!e_1.e_1!e_2$, as channel e_1 first sent is fresh, and, hence, $a?(y, z)$ in the input process can be represented by $a?y.y?z$.

2.6.2 Operational correspondence

Our operational correspondence result relates the sum-free π-calculus and the C_π-calculus, relying on the respective reduction semantics. Actually, the reduction semantics of the C_π-calculus, presented in Section 2.3, uses the same set of rules as the reduction semantics given in [102], restricted here only with the syntax of C_π. Hence, we will use $\equiv$, $\rightarrow$ and $\simeq$ to denote structural congruence, reduction, and strong barbed equivalence relation, respectively, for both π and C_π, contextualized whenever required to clarify to which of these languages they belong. We also use $\rightarrow^*$ to denote the transitive closure of $\rightarrow$.

The encoding (Table 2.5) does not introduce any free names, except in the rule for output, where name m_b is introduced. This name is also the one specified in the renaming policy of name b. Hence, assuming that substitutions on pairs of names introduced by the renaming policy are defined component-wise: the first components are mapped to $\mathcal{N}_\pi$, and the second components are mapped to $\mathcal{N}_\varphi$, we have the following result.

Lemma 2.6.1 (Name invariance)**.** *Let P be a π process and let substitutions σ and σ' be such that $\varphi_{[\![\,]\!]}(\sigma(a)) = \sigma'(\varphi_{[\![\,]\!]}(a))$, for all $a \in \mathcal{N}_\pi$. Then $[\![(P)\sigma]\!] = ([\![P]\!])\sigma'$.*

Proof. The proof is by induction on the structure of process P. We detail only the case when $P = a!l.P_1$. Assume $\sigma(a) = b$ and $\sigma(l) = m$. If $\varphi_{[\![\,]\!]}(a) = (a, m_a)$ and $\varphi_{[\![\,]\!]}(b) = (b, m_b)$ then from $\varphi_{[\![\,]\!]}(\sigma(a)) = \sigma'(\varphi_{[\![\,]\!]}(a))$ we conclude $\sigma'(a, m_a) = (b, m_b)$. Likewise, if $\varphi_{[\![\,]\!]}(l) = (l, m_l)$ and $\varphi_{[\![\,]\!]}(k) = (k, m_k)$ we have $\sigma'(l, m_l) = (k, m_k)$. Then,

$$\begin{aligned}
[\![(a!l.P_1)\sigma]\!] &= [\![b!k.(P_1\sigma)]\!] \\
&= (\nu e_1, e_2)b!(e_1, e_2).m_k!e_1.e_2!e_1.[\![(P_1\sigma)]\!] \\
&= (\nu e_1, e_2)b!(e_1, e_2).m_k!e_1.e_2!e_1.([\![P_1]\!])\sigma' \\
&= ((\nu e_1, e_2)a!(e_1, e_2).m_l!e_1.e_2!e_1.[\![P_1]\!])\sigma' \\
&= ([\![a!l.P_1]\!])\sigma'
\end{aligned}$$

by the definition of the encoding, and where $[\![(P_1\sigma)]\!] = ([\![P_1]\!])\sigma'$ holds by induction hypothesis. □

For the operational correspondence, we need only one case of the name invariance result, and we state it in the next corollary.

Corollary 2.6.2 (Encoding and substitution)**.** *Let P be a π process and $k, x \in \mathcal{N}_\pi$ such that $\varphi_{[\![\,]\!]}(k) = (k, m_k)$ and $\varphi_{[\![\,]\!]}(x) = (x, m_x)$. Then*

$$[\![P]\!]\{k/x\}\{m_k/m_x\} = [\![P\{k/x\}]\!].$$

To simplify notation we use the following abbreviations: $(\nu\tilde{k})P$ stands for $(\nu k_1)\ldots(\nu k_n)P$ or P (when $\tilde{k}$ is an empty list), and H_l stands for the handler process $!m_l?x.x!(l, m_l).0$, where $\varphi_{[\![\,]\!]}(l) = (l, m_l)$. We also use $(\nu k, m_k)P$ to abbreviate $(\nu k)(\nu m_k)P$, and $(\nu\tilde{k}, \tilde{m}_k)P$ to abbreviate $(\nu k_1, m_{k_1}), \ldots, (\nu k_n, m_{k_n})P$ or P, where $\varphi_{[\![\,]\!]}(k_i) = (k_i, m_{k_i})$. Furthermore, whenever we write name m_a from $\mathcal{N}_\varphi$ we assume that $\varphi_{[\![\,]\!]}(a) = (a, m_a)$, for $a \in \mathcal{N}_\pi$.

In order to show that our encoding preserves the structural congruence relation, up to strong barbed equivalence, we present an auxiliary result showing that a restricted handler process is behaviorally equivalent to the inactive process.

Proposition 2.6.3 (Restricted handlers)**.** *For any channel name $k \in \mathcal{N}_\pi$ we have that $(\nu m_k)H_k \simeq 0$.*

Proof. The proof is direct. We show that the two processes are bisimilar, by noticing that the only possible action of the process

$$(\nu m_k)H_k = (\nu m_k)!m_k?x.x!(k, m_k).0$$

is an input on m_k that is blocked due to side condition of rule (RES) since the subject of the action is restricted. Hence, the process is strongly bisimilar with an inactive process. Then, by Theorem 2.4.9 (which states $\sim = \simeq$) we conclude the proof. □

The next lemma shows that the encodings of two structurally equivalent processes yield two processes related by the strong barbed equivalence relation. Here we use structural congruence relation of (sum-free) π processes as defined in [102], which matches the rules given in Section 2.3.

Lemma 2.6.4 (Encoding and structural congruence)**.** *If P and Q are π processes such that $P \equiv Q$ then $[\![P]\!] \simeq [\![Q]\!]$.*

Proof. The proof is by induction on the derivation $P \equiv$. We perform the case analysis on the last rule applied. In all cases, except the case 2., we may directly show that if $P \equiv Q$ then $[\![P]\!] \equiv [\![Q]\!]$, and then to conclude by Proposition 2.4.3 ($\equiv \subseteq \sim$) and Theorem 2.4.9 ($\sim = \simeq$).

1. $[a = a]\pi.P \equiv \pi.P$.

   We distinguish two cases for prefix π.

   (a) If $\pi = [\tilde{b} = \tilde{c}]d?x$, then by definition of the encoding and definition of the structural congruence (Table 2.3) relation we have that

   $$\begin{aligned}[\![[a = a]\pi.P]\!] &= [a = a][\tilde{b} = \tilde{c}]d?(y, z).y?(x, m_x).z?z'.[\![P]\!] \\ &\equiv [\tilde{b} = \tilde{c}]d?(y, z).y?(x, m_x).z?z'.[\![P]\!] \\ &= [\![\pi.P]\!].\end{aligned}$$

   (b) If $\pi = [b_1 = c_1]...[b_n = c_n]d!g$, then, since $\equiv$ is a congruence, hence preserved also by channel restriction construct, we may show

   $$\begin{aligned}[\![[a = a]\pi.P]\!] &= (\nu e_1, e_2)[a = a][\tilde{b} = \tilde{c}]d!(e_1, e_2).m_g!e_1.e_2!e_1.[\![P]\!] \\ &\equiv (\nu e_1, e_2)[\tilde{b} = \tilde{c}]d!(e_1, e_2).m_g!e_1.e_2!e_1.[\![P]\!] \\ &= [\![\pi.P]\!].\end{aligned}$$

2. $(\nu k)0 \equiv 0$. By the definition of the encoding and structural congruence, and by Proposition 2.6.3 we observe

   $$[\![(\nu k)0]\!] = (\nu k, m_k)(0 \mid H_k) \equiv (\nu k, m_k)H_k \simeq 0 = [\![0]\!].$$

3. The rest of the cases are analogous.

□

We may present the first main result that attests the correctness of our translation, which says that if the source process P reduces to Q then the encoding of P also reduces, in a number of steps, to the encoding of process Q. Since in the reductions of a source process free names can be exchanged, our result uses "top-level" handlers for all free names specified as objects of the output prefixes of the process. In what follows, we use $\prod_{i\in I} P_i$ to abbreviate $P_1 \mid \ldots \mid P_n$ when $I = \{1,\ldots,n\}$, and 0 when $I = \emptyset$.

Lemma 2.6.5 (Completeness with top-level handlers)**.** *If P and Q are π processes such that $P \to Q$ then*

$$[\![P]\!] \mid H \to^* \simeq [\![Q]\!] \mid H$$

where $H = \prod_{n\in N} H_n$, for any finite $N \subset \mathcal{N}_\pi$, such that $\mathsf{fo}(P) \subseteq N$.

Proof. The proof is by induction on $P \to Q$ derivation.

1. *Base case*: $k!l.P \mid k?x.Q \to P \mid Q\{l/x\}$. Since l is a free object of the prefix in the starting process, we need to show that

$$[\![k!l.P \mid k?x.Q]\!] \mid H \to^* \simeq [\![P \mid Q\{l/x\}]\!] \mid H$$

where $H \equiv H_l \mid H_1$, for some H_1. If we denote $R = [\![k!l.P \mid k?x.Q]\!] \mid H_l \mid H_1$, we have

$$\begin{aligned} R \;&=\; (\nu e_1,e_2)k!(e_1,e_2).m_l!e_1.e_2!e_1.[\![P]\!] \mid k?(y,z).y?(x,m_x).z?z'.[\![Q]\!] \\ &\quad \mid\; !m_l?x.x!(l,m_l).0 \mid H_1 \\ &\to\!\to\; (\nu e_1,e_2)(e_2!e_1.[\![P]\!] \mid e_1?(x,m_x).e_2?z'.[\![Q]\!] \mid e_1!(l,m_l).0) \mid H_l \mid H_1 \end{aligned}$$

where the output process first synchronizes with the receiving process, sending fresh channels e_1 and e_2, and then, with the handler of channel l, by sending e_1, and, thus creates a private connection between the receiving process and the handler. At this point, the handler can synchronize with the receiving process

$$(\nu e_1,e_2)(e_2!e_1.[\![P]\!] \mid e_1?(x,m_x).e_2?z'.[\![Q]\!] \mid e_1!(l,m_l).0) \mid H_l \mid H_1 \to$$

$$(\nu e_1, e_2)(e_2!e_1.[\![P]\!] \mid e_2?z'.[\![Q]\!]\{l/x\}\{m_l/m_x\} \mid 0) \mid H_l \mid H_1$$

where channels l and m_l are finally received in the input process. The encoding of processes P and Q is only unlocked in the synchronization on private channel e_2, and since e_1 and e_2 are from the reserved set of names, hence not free in $[\![P]\!] \mid [\![Q]\!]\{l/x\}\{m_l/m_x\}$, the last derived process can reduce

$$(\nu e_1, e_2)(e_2!e_1.[\![P]\!] \mid e_2?z'.[\![Q]\!]\{l/x\}\{m_l/m_x\} \mid 0) \mid H_l \mid H_1 \rightarrow$$

$$[\![P]\!] \mid [\![Q]\!]\{l/x\}\{m_l/m_x\} \mid H_l \mid H_1$$

By Corollary 2.6.2 we have $[\![Q]\!]\{l/x\}\{m_l/m_x\} = [\![Q\{l/x\}]\!]$. Hence, we have that

$$[\![P]\!] \mid [\![Q]\!]\{l/x\}\{m_l/m_x\} \mid H_l \mid H_1 \equiv [\![P \mid Q\{l/x\}]\!] \mid H$$

and we may conclude by Proposition 2.4.3 ($\equiv \subseteq \sim$) and Theorem 2.4.9 ($\sim = \simeq$).

2. *Case*: $P \mid R \rightarrow Q \mid R$ is derived from $P \rightarrow Q$. By induction hypothesis

$$[\![P]\!] \mid H_1 \rightarrow^* \simeq [\![Q]\!] \mid H_1$$

where $H_1 = \prod_{n \in N} H_n$, for any finite $N \subset \mathcal{N}_\pi$, such that $\mathsf{fo}(P) \subseteq N$. Now, let us take $H_2 = \prod_{n \in \mathsf{fo}(R) \setminus N} H_n$.

Since $[\![P \mid R]\!] \mid H_1 \mid H_2 \equiv [\![P]\!] \mid H_1 \mid [\![R]\!] \mid H_2$ and $[\![Q]\!] \mid H_1 \mid [\![R]\!] \mid H_2 \equiv [\![Q \mid R]\!] \mid H_1 \mid H_2$, by (R-PAR) and (R-STRU) we can derive

$$[\![P \mid R]\!] \mid H_1 \mid H_2 \rightarrow^* \simeq [\![Q \mid R]\!] \mid H_1 \mid H_2$$

where $H_1 \mid H_2 = \prod_{n \in N'} H_n$, for any finite $N' \subset \mathcal{N}_\pi$, such that $\mathsf{fo}(P \mid R) \subseteq N'$.

3. *Case*: $(\nu k)P \to (\nu k)Q$ is derived from $P \to Q$. Again, by induction hypothesis

$$[\![P]\!] \mid H_1 \to^* \simeq [\![Q]\!] \mid H_1$$

where $H_1 = \prod_{n \in N} H_n$, for any finite $N \subset \mathcal{N}_\pi$, such that $\mathsf{fo}(P) \subseteq N$. We now distinguish two cases.

   (a) If $k \in \mathsf{fo}(P)$ then $H_1 \equiv H_k \mid H$, for some H. Then, by (R-RES) we can derive

$$(\nu k, m_k)([\![P]\!] \mid H_k \mid H) \to^* \simeq (\nu k, m_k)([\![Q]\!] \mid H_k \mid H)$$

   Since $k, m_k \notin \mathsf{fn}(H)$, we have

$$(\nu k, m_k)([\![P]\!] \mid H_k \mid H) \equiv (\nu k, m_k)([\![P]\!] \mid H_k) \mid H = [\![(\nu k)P]\!] \mid H$$

   Similarly, $(\nu k, m_k)([\![Q]\!] \mid H_k \mid H) \equiv [\![(\nu k)Q]\!] \mid H$, and by $\mathsf{fo}((\nu k)P) = \mathsf{fo}(P) \setminus \{k\}$, we can conclude the case.

   (b) If $k \notin \mathsf{fo}(P)$ then by (R-PAR) and (R-RES) we can derive

$$(\nu k, m_k)([\![P]\!] \mid H_1 \mid H_k) \to^* \simeq (\nu k, m_k)([\![Q]\!] \mid H_1 \mid H_k)$$

   Since now $k, m_k \notin \mathsf{fn}(H_1)$, similarly as in the previous case we can show $(\nu k, m_k)([\![P]\!] \mid H_1 \mid H_k) \equiv [\![(\nu k)P]\!] \mid H_1$ and $(\nu k, m_k)([\![Q]\!] \mid H_1 \mid H_k) \equiv [\![(\nu k)Q]\!] \mid H_1$. Then, by $\mathsf{fo}((\nu k)P) = \mathsf{fo}(P)$, we can conclude.

4. *Case*: $P' \to Q'$ is derived from $P \to Q$, where $P \equiv P'$ and $Q \equiv Q'$. By induction hypothesis

$$[\![P]\!] \mid H_1 \to^* \simeq [\![Q]\!] \mid H_1$$

where $H_1 = \prod_{n \in N} H_n$, for any finite $N \subset \mathcal{N}_\pi$, such that $\mathsf{fo}(P) \subseteq N$. By Lemma 2.6.4, $P \equiv P'$ implies $[\![P]\!] \simeq [\![P']\!]$ and $Q \equiv Q'$ implies

$[\![Q]\!] \simeq [\![Q']\!]$. Also, since $P \equiv P'$, by Lemma 2.3.2 we have $\mathsf{fo}(P) = \mathsf{fo}(P')$. Then, by Proposition 2.4.4 and Theorem 2.4.9 we get $[\![P]\!] \mid H_1 \simeq [\![P']\!] \mid H_1$. Hence, by definition of strong barbed equivalence

$$[\![P']\!] \mid H_1 \rightarrow^* \simeq [\![Q]\!] \mid H_1 \simeq [\![Q']\!] \mid H_1$$

which completes the proof.

□

Notice that the condition $P \rightarrow Q$ in the previous lemma can be generalized to the case of a sequence of reductions of the source term (i.e., $P \rightarrow^* Q$). This is for the arrival state (Q) also satisfies the lemma conditions, as by Lemma 2.3.2 from $P \rightarrow Q$ we may conclude $\mathsf{fo}(Q) \subseteq \mathsf{fo}(P)$. As a direct consequence of Lemma 2.6.5, we get the operational correspondence result for the encoding of π-calculus processes having no free object names.

Corollary 2.6.6 (Operational correspondence: completeness)**.** *Let P be a (sum-free) π process such that $\mathsf{fo}(P) = \emptyset$. If $P \rightarrow Q$ then $[\![P]\!] \rightarrow^* \simeq [\![Q]\!]$.*

We now proceed to show that our encoding also satisfies the soundness property (cf. [47]). To this end, we define the static contexts in order to determine the active prefixes of a C_π process. Intuitively, an (active) context is a process with a (non-prefixed) "hole", in which processes can be instantiated.

Definition 2.6.7 (Active contexts)**.** *Active contexts for C_π processes are defined as follows.*

$$\mathcal{C}[\cdot] ::= \quad \cdot \quad \mid \quad (P \mid \mathcal{C}[\cdot]) \quad \mid \quad (\mathcal{C}[\cdot] \mid P) \quad \mid \quad ((\nu k)\mathcal{C}[\cdot]) \quad \mid \quad !\mathcal{C}[\cdot]$$

Hence, prefix π inside process P is active only if there exists a context $\mathcal{C}[\cdot]$ and process P' such that $P = \mathcal{C}[\pi.P']$. Notice that, by the

definition of the encoding, only prefixes of the source terms reproduce sequences of prefixes in the target term, except the prefixes introduced by the handlers. However, active prefixes of the handlers are different from others in a target term, as they are all inputs with subject names introduced by the renaming policy (i.e., from $\mathcal{N}_\varphi$), while the subjects of other prefixes are the ones given in the source term (i.e., from $\mathcal{N}_\pi$). Hence, the handlers cannot be engaged in the reduction directly in the target term but can be engaged only in latter reductions. The next lemma shows that all active prefixes of any target term can be singled out using the structural congruence relation and that the target term can be directly related to the corresponding source term. Here we focus on the active prefixes of the target terms that can be engaged in a reduction, hence not the ones given in the handler processes.

Lemma 2.6.8 (Normal form of target and source terms)**.** *Let P be a π process. We have that*

$$\begin{aligned}[\![P]\!] &\equiv (\nu\tilde{k}, \tilde{m}_k)(\prod_{i\in I}[\![\pi_i.P_i]\!] \mid \prod_{j\in J}![\![R_j]\!] \mid H) \quad \textit{and}\\ P &\equiv (\nu\tilde{k})(\prod_{i\in I}\pi_i.P_i \mid \prod_{j\in J}!R_j)\end{aligned}$$

where $H = \prod_{k\in\tilde{k}} H_k$, and if $[\![P]\!] = \mathcal{C}[[\![\pi.Q]\!]]$ then there is some $i \in I$ such that $\pi.Q = \pi_i.P_i$.

Proof. The proof is by induction on the structure of process P.

1. *Case*: If $P = 0$ or $P = \pi.P_1$, the proof follows directly.

2. *Case*: $P = P_1 \mid P_2$. By induction hypothesis we have

$$\begin{aligned}[\![P_l]\!] &\equiv (\nu\tilde{k}_l, \tilde{m}_{k_l})(\prod_{i\in I_l}[\![\pi_i.P_i]\!] \mid \prod_{j\in J_l}![\![R_j]\!] \mid H^l) \quad \text{and}\\ P_l &\equiv (\nu\tilde{k}_l)(\prod_{i\in I_l}\pi_i.P_i \mid \prod_{j\in J_l}!R_j)\end{aligned}$$

   where $H^l = \prod_{k\in\tilde{k}_l} H_k$, and if $[\![P_l]\!] = \mathcal{C}[[\![\pi.Q]\!]]$ then there is $i \in I_l$ such that $\pi.Q = \pi_i.P_i$, for l in $\{1, 2\}$. Without loss of generality

we assume the chosen sets of labels I_1, I_2, J_1 and J_2 are pair-wise disjoint. Hence,

$$\begin{aligned}
[\![P]\!] &= [\![P_1]\!] \mid [\![P_2]\!] \\
&\equiv (\nu \tilde{k}_1, \tilde{m}_{k_1})(\nu \tilde{k}_2, \tilde{m}_{k_2})(\prod_{i \in I_1 \cup I_2} [\![\pi_i.P_i]\!] \mid \prod_{j \in J_1 \cup J_2} ![\![R_j]\!] \mid H^1 \mid H^2) \quad \text{and} \\
P &= P_1 \mid P_2 \\
&\equiv (\nu \tilde{k}_1)(\nu \tilde{k}_2)(\prod_{i \in I_1 \cup I_2} \pi_i.P_i \mid \prod_{j \in J_1 \cup J_2} !R_j)
\end{aligned}$$

Since the active prefixes of $[\![P]\!]$ are the ones of the $[\![P_1]\!]$ and $[\![P_2]\!]$, we may conclude the case.

3. *Case*: $P = (\nu k)P_1$. By induction hypothesis we have

$$\begin{aligned}
[\![P_1]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)(\prod_{i \in I} [\![\pi_i.P_i]\!] \mid \prod_{j \in J} ![\![R_j]\!] \mid H) \quad \text{and} \\
P_1 &\equiv (\nu \tilde{k})(\prod_{i \in I} \pi_i.P_i \mid \prod_{j \in J} !R_j)
\end{aligned}$$

where $H = \prod_{k \in \tilde{k}} H_k$, and if $[\![P_1]\!] = \mathcal{C}[[\![\pi.Q]\!]]$ then there is some $i \in I$ such that $\pi.Q = \pi_i.P_i$. Since $[\![P]\!] = (\nu k, m_k)(P_1 \mid H_k)$ and $\equiv$ is a congruence, we have that

$$\begin{aligned}
[\![P]\!] &\equiv (\nu k, m_k)(\nu \tilde{k}, \tilde{m}_k)(\prod_{i \in I} [\![\pi_i.P_i]\!] \mid \prod_{j \in J} ![\![R_j]\!] \mid H \mid H_k) \quad \text{and} \\
P &\equiv (\nu k)(\nu \tilde{k})(\prod_{i \in I} \pi_i.P_i \mid \prod_{j \in J} !R_j)
\end{aligned}$$

Since the active prefixes of $[\![P]\!]$ that are images of the source active prefixes are the ones of the $[\![P_1]\!]$ we may conclude the case.

4. *Case*: $P = !P_1$. Again, by induction hypothesis

$$\begin{aligned}
[\![P_1]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)(\prod_{i \in I} [\![\pi_i.P_i]\!] \mid \prod_{j \in J} ![\![R_j]\!] \mid H) \quad \text{and} \\
P_1 &\equiv (\nu \tilde{k})(\prod_{i \in I} \pi_i.P_i \mid \prod_{j \in J} !R_j)
\end{aligned}$$

where $H = \prod_{k\in\tilde{k}} H_k$, and if $[\![P_1]\!] = \mathcal{C}[[\![\pi.Q]\!]]$ then there is some $i \in I$ such that $\pi.Q = \pi_i.P_i$. Therefore,

$$\begin{aligned}
[\![P]\!] &\equiv !(\nu\tilde{k}, \tilde{m}_k)(\prod_{i\in I}[\![\pi_i.P_i]\!] \mid \prod_{j\in J}![\![R_j]\!] \mid H) \\
&\equiv (\nu\tilde{k}, \tilde{m}_k)(\prod_{i\in I}[\![\pi_i.P_i]\!] \mid \prod_{j\in J}![\![R_j]\!] \mid H) \mid !(\nu\tilde{k}', \tilde{m}'_k)(\prod_{i\in I}[\![\pi'_i.P'_i]\!] \mid \prod_{j\in J}![\![R'_j]\!] \mid H') \\
&\equiv (\nu\tilde{k}, \tilde{m}_k)(\prod_{i\in I}[\![\pi_i.P_i]\!] \mid \prod_{j\in J}![\![R_j]\!] \mid ![\![Q]\!] \mid H) \quad \text{and}
\end{aligned}$$

$$\begin{aligned}
P &\equiv !(\nu\tilde{k})(\prod_{i\in I}\pi_i.P_i \mid \prod_{j\in J}!R_j) \\
&\equiv (\nu\tilde{k})(\prod_{i\in I}\pi_i.P_i \mid \prod_{j\in J}!R_j) \mid !(\nu\tilde{k}')(\prod_{i\in I}\pi'_i.P'_i \mid \prod_{j\in J}!R'_j) \\
&\equiv (\nu\tilde{k})(\prod_{i\in I}\pi_i.P_i \mid \prod_{j\in J}!R_j \mid !Q \mid H)
\end{aligned}$$

where, in both cases we use α-conversion to distinguish between bound names of the replicated process and the activated copy, and hence, $Q = (\nu\tilde{k}')(\prod_{i\in I}\pi'_i.P'_i \mid \prod_{j\in J}!R'_j) \equiv_\alpha (\nu\tilde{k})(\prod_{i\in I}\pi_i.P_i \mid \prod_{j\in J}!R_j)$. Since the active prefixes of $[\![P]\!]$ are also active prefixes of $[\![P']\!]$ we may conclude the proof.

□

The reduction steps of the target terms can be divided into four kinds. In order to refer to each of a kind in the following results, we will explicitly decorate the reduction arrows. The first kind is the first reduction of the target term itself. As we already noted, the first reduction of a target term, which we denote with $\rightarrow_1$, is always carried out on a channel that is also free in the source term. This is a consequence of the definition of the encoding that the subject of active prefixes of the source terms are preserved, while the active prefixes of the handlers are all inputs with subject names introduced by the renaming policy (i.e., different from all source names). The steps after the first reduction are then uniquely determined. The sending

process must synchronize with the corresponding handler (kind two, denoted with $\rightarrow_2$), as the translating function introduces receiving prefixes on the channels which names are introduced by the renaming policy only in the handler processes. Then, the receiving and the handler process synchronize on a private channel (kind three, $\rightarrow_3$), which name is from the reserved set of names, and afterwards, the sending and the receiving process directly synchronize (kind four, $\rightarrow_4$) also on a private channel, hence activating a process that is a target term (up to structural congruence). Thus, we have

$$\begin{aligned}
[\![[\tilde{a} = \tilde{a}]g!l.P \mid [\tilde{b} = \tilde{b}]g?x.Q]\!] \mid H_l &\equiv (\nu e_1, e_2)(g!(e_1, e_2).m_l!e_1.e_2!e_1.[\![P]\!] \\
&\quad \mid g?(y, z).y?(x, m_x).z?z'.[\![Q]\!]) \mid H_l \\
&\rightarrow_1 \\
&\quad (\nu e_1, e_2)(m_l!e_1.e_2!e_1.[\![P]\!] \\
&\quad \mid e_1?(x, m_x).e_2?z'.[\![Q]\!]) \mid H_l \\
&\rightarrow_2 \\
&\quad (\nu e_1, e_2)(e_2!e_1.[\![P]\!] \\
&\quad \mid e_1?(x, m_x).e_2?z'.[\![Q]\!] \mid e_1!(l, m_l)) \mid H_l \\
&\rightarrow_3 \\
&\quad (\nu e_1, e_2)(e_2!e_1.[\![P]\!] \\
&\quad \mid e_2?z'.[\![Q\{l/x\}]\!]) \mid H_l \\
&\rightarrow_4 \\
&\quad [\![P \mid Q\{l/x\}]\!] \mid H_l
\end{aligned}$$

Notice that the type of reduction is invariant with respect to the application of reduction rules (R-PAR), (R-RES) and (R-STRU). We also denote the intermediate sub-processes derived in these reductions with

$$\begin{aligned}
Pr_1^l(P, Q) &= (\nu e_1, e_2)(m_l!e_1.e_2!e_1.[\![P]\!] \mid e_1?(x, m_x).e_2?z'.[\![Q]\!]) \\
Pr_2^l(P, Q) &= (\nu e_1, e_2)(e_2!e_1.[\![P]\!] \mid e_1?(x, m_x).e_2?z'.[\![Q]\!] \mid e_1!(l, m_l)) \\
Pr_3^l(P, Q) &= (\nu e_1, e_2)(e_2!e_1.[\![P]\!] \mid e_2?z'.[\![Q\{l/x\}]\!])
\end{aligned}$$

We may present now our first result characterizing the structure of the reducing target term and its correlation with the corresponding source term.

Lemma 2.6.9 (A first reduction of a target term)**.** *Let P be a π process and Q a C_π process such that $[\![P]\!] \to Q$. Then*

$$\begin{aligned}[\![P]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)([\![g!l.P_1 \mid g?x.Q_1]\!] \mid [\![R]\!] \mid H) \\ Q &\equiv (\nu \tilde{k}, \tilde{m}_k)(Pr_1^l(P_1, Q_1) \mid [\![R]\!] \mid H) \text{ and} \\ P &\equiv (\nu \tilde{k})(g!l.P_1 \mid g?x.Q_1 \mid R)\end{aligned}$$

where $H = \prod_{k \in \tilde{k}} H_k$.

Proof. Assume $[\![P]\!] \to Q$. By Lemma 2.6.8 we can single out all active prefixes, hence

$$\begin{aligned}[\![P]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)(\prod_{i \in I} [\![\pi_i.P_i]\!] \mid \prod_{j \in J} ![\![R_j]\!] \mid H) \quad \text{and} \\ P &\equiv (\nu \tilde{k})(\prod_{i \in I} \pi_i.P_i \mid \prod_{j \in J} !R_j)\end{aligned}$$

where $H = \prod_{k \in \tilde{k}} H_k$, and if $[\![P]\!] = \mathcal{C}[[\![\pi.Q]\!]]$ then there exist $i \in I$ such that $\pi.Q = \pi_i.P_i$. We can now single out the two prefixes that are involved in the reduction. Therefore,

$$\begin{aligned}[\![P]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)([\![g!l.P_j]\!] \mid [\![g?x.P_s]\!] \mid \prod_{i \in I \setminus \{j,s\}} [\![\pi_i.P_i]\!] \mid \prod_{j \in J} ![\![R_j]\!] \mid H) \quad \text{and} \\ P &\equiv (\nu \tilde{k})(g!l.P_j \mid g?x.P_s \mid \prod_{i \in I \setminus \{j,s\}} \pi_i.P_i \mid \prod_{j \in J} !R_j)\end{aligned}$$

If we denote $R = \prod_{i \in I \setminus \{j,s\}} \pi_i.P_i \mid \prod_{j \in J} !R_j$, we have that

$$\begin{aligned}[\![P]\!] &\equiv (\nu \tilde{k}, \tilde{m}_k)([\![g!l.P_j \mid g?x.P_s]\!] \mid [\![R]\!] \mid H) \quad \text{and} \\ P &\equiv (\nu \tilde{k})(g!l.P_j \mid g?x.P_s \mid R)\end{aligned}$$

Since $(\nu \tilde{k}, \tilde{m}_k)([\![g!l.P_j \mid g?x.P_s]\!] \mid [\![R]\!] \mid H) \to (\nu \tilde{k}, \tilde{m}_k)(Pr_1^l(P_j, P_s) \mid [\![R]\!] \mid H)$, and $[\![P]\!] \to Q$, we conclude

$$Q \equiv (\nu \tilde{k}, \tilde{m}_k)(Pr_1^l(P_j, P_s) \mid [\![R]\!] \mid H)$$

□

Notice that in the previous lemma the reduction is of type 1. Using Lemma 2.6.9 we may characterize all possible evolutions of the target terms after any number of reduction steps. For the rest of this section with $P \to^n Q$ we denote that process P reduces to Q in n reduction steps, i.e., that $P \to P_1 \to \ldots \to P_{n-1} \to Q$. If $P \to^n Q$ we denote by n_i the number of reduction steps of kind i, for $i = 1, 2, 3, 4$, and hence we have that $n = n_1 + n_2 + n_3 + n_4$.

Lemma 2.6.10 (An n-th reduction of the target term)**.** *Let P be a π process and Q a C_π process, such that $[\![P]\!] \mid H \to^n Q$, where $H = \prod_{k \in \mathsf{fo}(P)} H_k$. Then*

$$Q \equiv (\nu \tilde{k}, \tilde{m}_k)\big([\![R]\!] \mid \prod_{i \in I_1} Pr_1^{l_i}(P_i, Q_i) \mid \prod_{j \in I_2} Pr_2^{l_j}(P_j, Q_j) \mid \prod_{s \in I_3} Pr_3^{l_s}(P_s, Q_s) \mid H_1\big) \mid H$$

where I_1, I_2 and I_3 are pair-wise disjoint sets, $H_1 \equiv \prod_{k \in \tilde{k}} H_k$, and $|I_t| = n_t - n_{t+1}$, for $t = 1, 2, 3$, and there exist P' such that $P \to^{n_4} P'$ and

$$P' \equiv (\nu \tilde{k})\big(R \mid \prod_{i \in I}(g_i!l_i.P_i \mid g_i?x.Q_i)\big)$$

where $I = I_1 \cup I_2 \cup I_3$, and where $P' = P$ for $n_4 = 0$.

Proof. The proof is by induction on n. The base case follows directly from Lemma 2.6.9. Assume now $[\![P]\!] \mid H \to^n Q \to Q'$. By induction hypothesis

$$Q \equiv (\nu \tilde{k}, \tilde{m}_k)\big([\![R]\!] \mid \prod_{i \in I_1} Pr_1^{l_i}(P_i, Q_i) \mid \prod_{j \in I_2} Pr_2^{l_j}(P_j, Q_j) \mid \prod_{s \in I_3} Pr_3^{l_s}(P_s, Q_s) \mid H_1\big) \mid H$$

where I_1, I_2 and I_3 are pair-wise disjoint, $H_1 \equiv \prod_{k \in \tilde{k}} H_k$ and $|I_t| = n_t - n_{t+1}$, for $t = 1, 2, 3$, and there exist P' such that $P \to^{n_4} P'$ and

$$P' \equiv (\nu \tilde{k})\big(R \mid \prod_{i \in I}(g_i!l_i.P_i \mid g_i?x.Q_i)\big)$$

where $I = I_1 \cup I_2 \cup I_3$, and where $P' = P$ for $n_4 = 0$. We now have only four cases for the kind of the reduction $Q \to Q'$.

1. Kind 1. The reduction originates from $[\![R]\!] \to Q''$. By Lemma 2.6.9 we have that

$$\begin{aligned}[\![R]\!] &\equiv (\nu \tilde{k'}, \tilde{m}_{k'})([\![R']\!] \mid [\![g!l.P_1 \mid g?x.Q_1]\!] \mid H') \\ Q'' &\equiv (\nu \tilde{k'}, \tilde{m}_{k'})([\![R']\!] \mid Pr_1^l(P_1, Q_1) \mid H') \text{ and} \\ R &\equiv (\nu \tilde{k'})(R' \mid g!l.P_1 \mid g?x.Q_1)\end{aligned}$$

   where $H = \prod_{k \in \tilde{k'}} H_k$. The proof for this case follows by noting that processes Q' and P' have the expected forms, up to structural congruence.

2. Kind 2. The reduction of Q originates from $Pr_1^{l_i}(P_i, Q_i)$ and one copy of the corresponding handler (either from H_1 or H), and their parallel composition evolves to process $Pr_2^{l_i}(P_i, Q_i)$. Process P' remains unchanged.

3. Kind 3. The reduction of Q originates from $Pr_2^{l_i}(P_i, Q_i)$, that evolves to process $Pr_3^{l_i}(P_i, Q_i)$ (using Corollary 2.6.2). Again, process P' remains unchanged.

4. Kind 4. Process $Pr_3^{l_i}(P_i, Q_i)$ (in Q) evolves to $[\![P_i \mid Q_i\{l_i/x\}]\!]$, and the respective $g!l.P_i \mid g?x.Q_i$ (in P') evolves to $P_i \mid Q_i\{l_i/x\}$, which completes the proof.

□

The above result shows that our encoding does not introduce any unexpected computations. That is, for each possible evolution of the target term there is a corresponding evolution of the source term. That is the essence of the soundness result we aim to prove. Formally, our soundness result states that if a target term $[\![P]\!]$ reduces (in a number of steps) to some process Q then the source term P also reduces (in a number of steps) to a process P', where Q can reach $[\![P']\!]$ by reducing

to it (in a number of steps). Similarly to the completeness result, we first show that the soundness of the encoding holds for any π processes if the "top-level" handlers are used.

Lemma 2.6.11 (Soundness with top-level handlers)**.** *Let P be a π-calculus process and Q be a C_π process such that $[\![P]\!] \mid H \rightarrow^n Q$, where $H = \prod_{k \in \mathsf{fo}(P)} H_k$. Then, there is a π-calculus process P' such that $P \rightarrow^{n_1} P'$ and $Q \rightarrow^m [\![P']\!] \mid H$, where $m = 3n_1 - n_2 - n_3 - n_4$.*

Proof. Assume $[\![P]\!] \mid H \rightarrow^n Q$. By Lemma 2.6.10 we have that

$$Q \equiv (\nu \tilde{k}, \tilde{m}_k)\big([\![R]\!] \mid \prod_{i \in I_1} Pr_1^{l_i}(P_i, Q_i) \mid \prod_{j \in I_2} Pr_2^{l_j}(P_j, Q_j) \mid \prod_{s \in I_3} Pr_3^{l_s}(P_s, Q_s) \mid H_1\big) \mid H$$

where $H_1 \equiv \prod_{k \in \tilde{k}} H_k$ and $|I_t| = n_t - n_{t+1}$, for $t = 1, 2, 3$, and there exist a π process P'' such that $P \rightarrow^{n_4} P''$ and

$$P'' \equiv (\nu \tilde{k})\big(R \mid \prod_{i \in I} (g_i!l_i.P_i \mid g_i?x.Q_i)\big)$$

where $I = I_1 \cup I_2 \cup I_3$, and where $P'' = P$ for $n_4 = 0$. Thus, by performing synchronizations of the processes in the product in the process structurally equivalent to P'' we can show that

$$P'' \rightarrow^s (\nu \tilde{k})\big(R \mid \prod_{i \in I} (P_i \mid Q_i\{l_i/x\})\big)$$

where $s = n_1 - n_4$. By performing reductions of kind $2, 3$ and 4 in the process structurally equivalent to Q (three for each $i \in I_1$, two for each $j \in I_2$, and one for each $s \in I_3$) we have that

$$Q \rightarrow^m [\![(\nu \tilde{k})\big(R \mid \prod_{i \in I} (P_i \mid Q_i\{l_i/x\})\big)]\!] \mid H$$

where $m = 3n_1 - n_2 - n_3 - n_4$. □

As a direct consequence of the last lemma, we get the operational soundness result for our encoding.

Corollary 2.6.12 (Operational correspondence: soundness)**.** *Let P be a (sum-free) π-calculus process and Q be a C_π process, such that $\mathsf{fo}(P) = \emptyset$ and $[\![P]\!] \rightarrow^n Q$. Then, there is a π-calculus process P' such that $P \rightarrow^{n_1} P'$ and $Q \rightarrow^m [\![P']\!]$, where $m = 3n_1 - n_2 - n_3 - n_4$.*

Using the observation that whenever $[\![P]\!] \rightarrow^n Q$ then $n_1 \geq \frac{n}{4}$, and Lemma 2.6.11 we can directly conclude that our encoding does not introduce divergence computations (cf. [47]).

Corollary 2.6.13 (Divergence reflection)**.** *Let P be a π process. If $[\![P]\!] \mid H \rightarrow^\omega$, where $H = \prod_{k \in \mathsf{fo}(P)} H_k$, then $P \rightarrow^\omega$.*

2.7 Remarks

Notice that translating the π-calculus terms into the C_π-calculus via the encoding presented in Section 2.6.1 has one positive consequence to what concerns controlling channel sharing. The only processes which are able to send channels originally specified in the source π process are handlers, i.e., handlers are the administrators in the target terms for the channels of the source terms. Now, if a channel from the source language is to be considered confidential, one has a fixed domain in which control needs to be established. This is in contrast to the regular π processes where one cannot statically identify a domain where the channel sending capability is confined to. The mentioned control can be exploited also to bound the number of times channels are communicated, an exploration we leave for future work. We remark that such a notion of accounting is to some extent connected with the notion of accounting of usages of channels via floating authorizations presented in the next chapter.

Chapter 3

A calculus of floating authorizations

As we discussed in the Introduction, controlling access to resources is an important aspect of distributed systems. The limited capacity of resources imposes a need for careful control over their usages, such as the case, for instance, of the router that has limited access points to yield. In order to formally reason on controlling usages of resources in distributed systems, in this chapter we introduce a calculus for modeling floating authorizations, which is based on the work previously published in [92]. In essence, our model allows us to reason on controlling the usages of channels relying on the previous developments [43, 44] that extend the π-calculus [102] with the constructs for authorization manipulation. The main distinction with respect to previous approaches is that in our model, a floating authorization represents the right to use a channel by a process, in such a way that only one thread of the process can use the channel.

Overview of the chapter. We start this chapter by a sequence of examples that gradually and informally introduce our process model in Section 3.1, after which we introduce the syntax in Section 3.2.

The operational semantics is then given by means of a labeled transition system in Section 3.3, and a reduction in Section 3.4, where we also define the notion of error processes and, via Harmony result, we show that the two semantics can be seen as alternatives to each other. Relying on the labeled transition system, Section 3.5 presents a preliminary investigation of the behavioral semantics of the model, including some fundamental results and behavioral (in)equalities that inform on the specific nature of our authorizations. The typing discipline, given in Section 3.6, addresses processes where authorizations for received names may be provided by the context, refined in Section 3.6.6 so as to allow for a more applicable procedure. In Section 3.7 we present an extended example inspired by the Bring Your Own License notion, while in Section 3.8 we discuss possible programming language applications of our formal framework for authorization control, namely by considering an extension of the Go programming language.

3.1 Preview of the model

This section informally introduces our process model. We make use of concurrent use licenses [10] setting to place our examples, which should allow for an intuitive reading of the formalisms introduced throughout later sections.

As noted in the Introduction, we investigate the following dimensions of floating authorizations: domain (to capture where access may be implicitly granted), accounting (to capture the capacity), and delegation (to capture explicit granting).

Domain. We model authorization domains by considering a non-binding scoping construct. For instance, we may have

$$(license)\, University$$

representing that the process *University* is a domain that holds one *license*. In this case, construct $(license)$ authorizes the usage of *license*

inside domain *University* only for one user. This means that if domain *University* is composed of two concurrently active students *Alice* and *Bob* then we have

$$(license)(Alice \mid Bob)$$

where license (*license*) is available for both students, but only one of them can use it. In other words, here (*license*) is "floating", as it can be grabbed by either *Alice* or *Bob*, but not both of them. The authorization is a non-binding scoping construct, meaning that name *license* can be also known in other domains except *University*.

Accounting. In the example above, the capacity of *University* includes one license for usages of *license*. Hence, if *Bob* uses the license and evolves to *LicensedBob*, for the purpose of accounting, we need to denote that the license is not available for *Alice* anymore. In our model, we confine the scope of the license to the user that grabs the license, and the system above evolves to

$$Alice \mid (license)LicensedBob \tag{3.1}$$

in which case *Alice* loses the ability to use the license. The license is implicitly granted to *Bob* only because he was the first one to use it, and afterwards he can continue using the *license*. If now *Alice* tries to use *license* she will get "stuck" as the proper authorization is now missing. Notice that, since the authorization is a non-binding construct, the confinement of the license in (3.1) does not mean that the name is privately held by *LicensedBob*, just the authorization.

In our model resources are used in a non-eager way, as we do not allow a user to be confined with a shared license if his domain already includes the respective license. For instance, in

$$(license)(Alice \mid (license)LicensedBob)$$

the user *LicensedBob* shares one (*license*) with *Alice* and also possesses one by himself. If *LicensedBob* needs the license, he will be granted with the "private" one first, not interfering with *Alice*, but can be also granted the shared *license* if he really needs two licenses.

Delegation. In our model, the licenses can be explicitly exchanged between the users via a delegation mechanism, that allows for sending and receiving a license over a communication channel. For instance, if *Bob* wants to explicitly delegate one authorization (*license*) to *Carol*, we may write process

$$auth\langle license\rangle.UnlicensedBob$$

to represent that *Bob* sends on channel *auth* one authorization for *license*, and then evolves to *UnlicensedBob*. Then, if *Carol* wants to receive the authorization, we may write

$$auth(license).LicensedCarol$$

to represent the dual primitive that allows receiving one authorization for *license* on channel *auth*, after which the process evolves to *LicensedCarol*. Then, the configuration

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

represents a system where the authorization for *license* can be transferred from *Bob* to *Carol*, evolving to

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

where the scope of (*license*) primitive changes accordingly. Notice that the authorizations for channel *auth*, on which the communication is carried, are present at both sending and receiving end. As we noted, the only resources in our model are channels and their usages are always subject to the authorization granting mechanism.

In addition to authorization manipulation constructs, our model comprehends π-calculus constructs for name passing, name generation and replicated input.

Name passing. The authorization delegation mechanism illustrated above does not involve name passing since the name *license* is known to both *Bob* and *Carol* in the first place. Name passing is supported by dedicated primitives. For example, process

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

represents a system where the name *license* can be passed via a synchronization on channel *comm*. If the synchronization take place, continuations *Alice* and $Dylan'$ are activated, where $Dylan'$ is obtained from process *Dylan* by replacing each occurrence of placeholder x with *license*. Again, we remark that the sending and receiving actions on channel *comm* are properly authorized, enabling that the synchronization can take place. The authorizations can also be floating, like in

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

where the synchronization may also occur. However, the synchronization in

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

is not properly authorized since there is only one authorization available for both actions on channel *comm*.

Name restriction and replicated input. The two last constructs of our language are name restriction and replicated input. As an example, consider process

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

representing a system in which in the left-hand side thread a licensing server is specified, used in the right-hand side thread. Construct $(\nu fresh)Domain$ represents the creation of a new name *fresh*, which is known only to scoped process *Domain*, in contrast with authorization scoping (cf. discussion after (3.1)). The thread on the right-hand side

represents a process that first creates a name and then sends it via channel *license*. Then, via channel *license*, the thread receives the authorization to use channel *fresh* and then terminates (denoted with 0). The thread on the left-hand side is repeatably available to receive a name on channel *license*. After that, one authorization scope for the received name is specified that may then be delegated away.

A remark on authorization delegation and name passing. As the communicated names refer to channels the name passing is the mechanism that allows to model systems in which access to channels changes dynamically. However, knowing a name does not mean being authorized to use it. For example, process

$$(comm)comm?x.x!reply.0$$

specifies an authorized reception on *comm*, after which the process outputs *reply* on the received name and then terminates. If the received name is *license*, the process evolves 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. Therefore, the output on *license* is not authorized and cannot take place. However, notice that we do not require an authorization for the name *reply* specified in the output, as communicating a name does not entail usage for the purpose of authorization control. Our design choice to separate name passing and authorization delegation allows us to model systems where unauthorized intermediaries (e.g., brokers) may forward names between authorized parties, without ever being authorized to use such names. For instance, consider process $(comm)comm?x.(forward)forward!x.0$. that requires no further authorizations.

There are two patterns for authorizing names that are received and new to the process. To illustrate the first, consider process

$$(comm)comm?x.(auth)auth(x).LicensedDylan$$

where, for the name received on *comm*, an authorization reception (using placeholder x) on *auth* is specified. This enables to acquire an

authorization to use the received name. Another pattern for acquiring authorizations for received names is to use the authorization scoping directly. For instance, in process

$$(comm)comm?x.(x)LicensedDylan$$

the authorization (x) is instantiated by the received name (cf. example with the licensing server above). The last example shows that the authorization scoping is a powerful mechanism, as it allows for the generation of authorizations for any received name, that therefore should be reserved only to the Trusted Computing Base, while the authorization delegation should be used elsewhere. We may notice that the above combination of name reception and authorization scoping resembles the authorization reception since the result is also an acquired authorization. However, we do not see a direct way to represent authorization delegation with this combination of name passing and authorization scoping, as in the former the delegating party actually loses the respective authorization, while in the latter an additional authorization is created.

3.2 Syntax

This section presents the syntax of our process calculus, which builds on previous work on process calculi for authorizations [43, 44]. We do not introduce any new syntax constructs and we fully exploit the syntax from [43]. Our calculus departs semantically in the interpretation of the accounting principle, and this appears to be crucial to capture the floating nature of the authorizations investigated in this work. We will point to the similarities/differences of the two process algebras throughout the presentation.

Our language also relies on *names*. We assume a countable set of names $\mathcal{N}$, and we let $a, b, c, \ldots, x, y, z, \ldots$ range over $\mathcal{N}$. Hence, here we do not make an explicit distinction between channels and variables, as we did in the previous chapter. Table 3.1 presents the syntax of the

P	::=		*Process terms*
		0	(termination)
	$\mid$	$P \mid P$	(parallel composition)
	$\mid$	$(\nu a)P$	(name restriction)
	$\mid$	$a!b.P$	(output)
	$\mid$	$a?x.P$	(input)
	$\mid$	$(a)P$	(authorization)
	$\mid$	$a\langle b\rangle.P$	(send authorization)
	$\mid$	$a(b).P$	(receive authorization)
	$\mid$	$!(a)a?x.P$	(replicated input)

Table 3.1: Syntax of processes.

language. The first five constructs are adopted from the π-calculus. Their interpretation is the same as for the C_π. To make the chapter self-contained, we briefly repeat the explanations: 0 represents the terminated process; $P \mid P$ represents two processes simultaneously active (that can interact via synchronization in channels); $(\nu a)P$ represents the creation of a channel name a, known only to process P; $a!b.P$ represents the output prefixed process that can send name b on name a and proceed as P; and $a?x.P$ represents the input prefixed process that receives a name on channel a and replaces name x in P with the received name. Notice that here we do not restrict forwarding as in the previous chapter, since it does not serve the goals of the work presented here.

The remaining language constructs, except the replicated input, are adopted from [43] but are given a different interpretation:

- Authorization scoping $(a)P$ represents that process P has one authorization to use channel a for any actions that are composed sequentially. For instance, if $P = a!b.a?x.0$ then both sending and receiving on a are considered to be authorized in

$(a)P$. Conversely, if $P = a!b.0 \mid a?x.0$ then only one of the actions is authorized in $(a)P$. In contrast with name restriction, in $(a)P$ name a is not private to P, hence the name can be known to other processes.

- Authorization sending $a\langle b\rangle.P$ represents the process that delegates one authorization for name b along name a and evolves to P. For instance, if $(b)a\langle b\rangle.P$ then performing the authorization sending the scoping authorization for b will be delegated, and the process evolves to P.

- Authorization receiving $a(b).P$ represents the (dual) process that receives one authorization for name b along name a and evolves to $(b)P$. Notice that the authorization that is lost in the output action now appears in the input action. Hence, the number of authorizations remains stable.

- Replicated input $!(a)a?x.P$ represents the process with an infinite behavior. It receives a name on authorized name a and in P replaces x with the received name, and in parallel activates the original process. For instance, if the received name is b, the above process evolves to $(a)P' \mid !(a)a?x.P$, where P' is obtained from P by replacing x with b.

When compared with the C_π-calculus syntax introduced in Section 2.1, one may notice that here we are adopting a restricted and more controlled version of the replicated processes. However, we will show later that a general replication construct can be encoded using replicated input following standard lines.

As in C_π, name restriction and input are binding names (cf. Definition 2.1.1). Hence, in $(\nu x)P$, $a?x.P$ and $!(a)a?x.P$ the name x is *binding* with scope P. As in Section 2.1, we use $\mathsf{fn}(P)$, $\mathsf{bn}(P)$ and $\mathsf{n}(P)$ to denote the sets of free, bound and all names in P, respectively. In $(a)P$ the name a is free and the names a and b in processes $a\langle b\rangle.P$ and $a(b).P$ are also free. Hence, we extend here Definition 2.1.1 of Section 2.1

to consider constructs for authorization manipulation and we define $\mathsf{fn}((a)P) = \{a\} \cup \mathsf{fn}(P)$ and $\mathsf{fn}(a\langle b\rangle.P) = \mathsf{fn}(a(b).P) = \{a, b\} \cup \mathsf{fn}(P)$, and also $\mathsf{fn}(!(a)a?x.P) = (\{a\} \cup \mathsf{fn}(P)) \setminus \{x\}$. For the rest of presentation we use α_a to abbreviate $a!b, a?x, a\langle b\rangle$ or $a(b)$ (including when $b = a$) and $(\nu\tilde{a})$ to abbreviate $(\nu a_1)\ldots(\nu a_n)$.

Same as in the previous chapter, here we also do not consider the sum operator of the π-calculus [102]. Our aim is to study floating authorizations in a minimal setting and we believe the sum operator can be added to our development following standard lines, as the interplay between choice and authorization scope is the same with respect to the one between communication prefixes and authorization scope. The principal relation that is addressed here in a central way is the one between parallel composition and authorization scope, so as to capture the desired notion of accounting.

The communication can be seen as a core of the behavior of processes (same as in π and C_π). Our model imposes control on this behavior: two processes can communicate on a channel only if both are authorized to use the channel. We present two examples that motivate our operational semantics introduced in the next section. The examples informally describe what sort of communications are to be considered authorized.

Example 3.2.1 (Authorized communications)**.**

1. *Two processes*

$$(a)a!b.P \mid (a)a?x.Q \quad \text{and} \quad (a)(a)(a!b.P \mid a?x.Q)$$

have their output and input actions authorized and can both evolve to the same process $(a)P \mid (a)Q\{b/x\}$*, where the confinement of the authorizations takes place as described in Section 3.1.*

2. *Authorization delegation is another aspect of our language. For instance, process*

$$(a)(b)a\langle b\rangle.P \mid (a)a(b).Q$$

has both actions authorized on name a and the delegating process has the authorization on b. Therefore, two processes can synchronize evolving to $(a)P \mid (a)(b)Q$ where the scope of authorization for b changes accordingly. Notice that for the delegation to take place three authorizations are needed, one for each of the two processes on name a and one for the delegating process for name b.

3. *As already noted, in the communication the authorizations are only confined and are not consumed. These can further be used to authorize the actions of the continuations. For instance,*

$$(a)a!b.a?y.0 \mid (a)a?x.a!c.0$$

can evolve to $(a)a?y.0 \mid (a)a!c.0$, that in turn can further evolve to $(a)0 \mid (a)0$.

Example 3.2.2 (Unauthorized communications)**.**

1. *Both of the processes*

$$(a)a!b.P \mid a?x.Q \quad \text{and} \quad (a)(a!b.P \mid a?x.Q)$$

are considered as stuck, as in both the synchronization is not possible only due to lack of one authorization for a. The left-hand side process lacks one authorization for name a on the receiving end and the right-hand side process also lacks one authorization for name a since only one is available, while two are required for the synchronization to occur. This is one of the main differences with respect to [43, 44], where the operational semantics allows the latter process to evolve, since there both communication ends are considered to be authorized by (a), due to a different interpretation of the accounting principle.

2. *Process*

$$(a)a\langle b\rangle.P \mid (a)a(b).Q$$

is again stuck as it cannot evolve only due to lack of one authorization. The delegation in left-hand side is not authorized as authorization for name b is missing.

We formally define the evolutions of processes in two alternative ways, using a labeled transition system and a reduction semantics, and we show that these are equivalent. Compared to C_π and π, the reduction semantics of this model is more involved, but still more convenient to be used in the proofs than the labeled transition system. Furthermore, we believe our authorization accounting principle can be explained in a more appropriate way when considering the reduction semantics (in particular when addressing the structural congruence relation), so we leave a more detailed account of this principle (including the main difference w.r.t. [44]) for the beginning of Section 3.4.

3.3 Action semantics

As we noted in Section 2.2, a labeled transition system (LTS) can be used to describe the behavior of a process by observing the actions of its sub-processes. We now characterize the observable labels.

Definition 3.3.1 (Actions)**.** *The set of observable actions $\mathcal{A}$, ranged over by α, is defined as*

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

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 $a = b$.

When compared with Definition 2.2.1, we may recognize the communication actions decorated here with annotations that capture lacking authorizations. Intuitively, a communication action is decorated with $(a)^0$ when the action carries sufficient authorizations on a, and $(a)^1$, represents the action lacks an authorization on a. In the action

for authorization delegation two such annotations are present, one for each name involved. As in the C_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. Whenever possible, we omit $(a)^0$ annotations and, thus, we use τ to abbreviate $\tau_{(a)^0}$ and $\tau_{(a)^0(b)^0}$ where no authorizations are lacking. As expected, we also abbreviate tags $(a)^1$ with (a).

By $\mathsf{n}(\alpha)$, $\mathsf{fn}(\alpha)$ and $\mathsf{bn}(\alpha)$ we denote the set of all, free and bound names of α. As in Definition 2.2.2, only the object name of the bound output action is bound, while the rest of the names in all actions are free. Also, we define $\mathsf{n}(\tau) = \emptyset$ and $\mathsf{fn}(\tau_{(a)}) = \mathsf{fn}(\tau_{(a)(b)^0}) = \mathsf{fn}(\tau_{(a)^2(b)^0}) = \{a\}$. Hence, if the internal action is not lacking authorizations for a name, the name is not considered as exposed in the label. The substitution of names is defined analogously as for the C_π processes (cf. Definition 2.1.3). We also identify here α-convertible processes (cf. Definition 2.1.4), and we use the convention that all bound and free names are distinct in all processes, substitutions and actions under consideration.

The labeled 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, that satisfies the rules given in Table 3.2. We now describe the rules.

- Rules (L-OUT), (L-IN), (L-OUT-A), (L-IN-A) directly correspond to explanations given for the communication prefixes. Each label is tagged with the authorizations needed to complete the action, as at this point none of the actions is authorized. Also, the resulting processes are scoped with the authorizations required for the action (and provided in case of authorization reception), so as to implement confinement.

- Rule (L-IN-REP) describes the only possible action of the replicated input. Notice that the replicated input is authorized by definition, for which the action is authorized and the label is not decorated (tag $(a)^0$ is omitted).

- Rule (L-PAR) lifts the action of one of the branches at the level of parallel composition and the side condition ensures the bound name of the action is not specified as free in the parallel process;

- In rule (L-RES) the action of P is lifted at the level of the scoped process, ensuring that the restricted name is not a name of the action.

- Rule (L-OPEN) supports the scope extrusion of the sent restricted name a by opening its scope, which is then to be closed by rule (L-CLOSE), following the lines of the semantics of C_π.

- Rules (L-SCOPE-INT) and (L-SCOPE-EXT) deal with the case of an action that lacks (at least) one authorization on a. This is represented with labels $\tau_{\omega(a)}$ and $(a)\sigma_a$. In the conclusions of the rules, the actions exhibited no longer lack the respective authorization and in the resulting processes the authorization scope is no longer present. With $\omega(a)$ we abbreviate $(a)^2(b)^k$, $(a)^1(b)^k$, and $(b)^{i+j}(a)^1$, for a given b (including the case $b = a$). With $(a)\sigma_a$ we abbreviate the communication actions lacking authorization on a ($(a)\alpha_a, (a)(\nu b)a!b$, and $(a)(b)^i b\langle a\rangle$ where $i \in \{0, 1\}$, which includes $(a)^1 a\langle a\rangle$). In both cases τ_ω and σ_a are obtained from $\tau_{\omega(a)}$ and $(a)\sigma_a$ by the respective exponent decrement for a. 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.

- In rule (L-SCOPE) the action of P is not lacking an authorization on a, thus, the action is lifted at the level of the scoped process.

- In rule (L-COMM) two parallel processes synchronize their dual actions: one process is sending and the other is receiving a name

b on a. The authorizations lacking in sending and receiving actions are then specified in the resulting internal action in ω.

- In rule (L-CLOSE) two parallel processes are also synchronizing their dual actions, only now the sent name b is bound. The scope of the sent name is closed in the final process (the scope was previously opened in (L-OPEN)). The side condition ensures avoiding unintended name capture.

- In rule (L-AUTH) the processes synchronize their dual actions: one is sending (delegating) and the other receiving authorization for b on channel a. The lacking authorizations of the sending and receiving authorization actions are again specified in ω.

Symmetric cases of rules (L-COMM), (L-CLOSE), (L-AUTH) and (L-PAR) are omitted from the table. Let us notice that carried authorization annotations, considered here up to permutation, thus identify, in a compositional way, the requirements for a synchronization to occur.

To illustrate the rules, consider process

$$(b)((a)a!b.a\langle b\rangle.0 \mid (a)a?x.a(x).x!c.0)$$

By (L-OUT) and (L-IN) we have that

$$a!b.a\langle b\rangle.0 \xrightarrow{(a)a!b} (a)a\langle b\rangle.0 \quad \text{and} \quad a?x.a(x).x!c.0 \xrightarrow{(a)a?b} (a)a(b).b!c.0$$

Then, the lacking authorizations in labels, that are already confined to the continuation processes, are removed by (L-SCOPE-EXT), obtaining

$$(a)a!b.a\langle b\rangle.0 \xrightarrow{a!b} (a)a\langle b\rangle.0 \quad \text{and} \quad (a)a?x.a(x).x!c.0 \xrightarrow{a?b} (a)a(b).b!c.0$$

The two branches now can synchronize their actions by rule (L-COMM)

$$(a)a!b.a\langle b\rangle.0 \mid (a)a?x.a(x).x!c.0 \xrightarrow{\tau} (a)a\langle b\rangle.0 \mid (a)a(b).b!c.0$$

The authorization for b scoping over both branches is not lacking in the action, hence by (L-SCOPE) the action is seamlessly lifted

$$(b)((a)a!b.a\langle b\rangle.0 \mid (a)a?x.a(x).x!c.0) \xrightarrow{\tau} (b)((a)a\langle b\rangle.0 \mid (a)a(b).b!c.0)$$

Now the name b is received in the right branch, and the respective authorization, that is floating over both branches, can be explicitly delegated from the left to the right branch. By rules (L-OUT-A) and (L-IN-A) we have

$$a\langle b\rangle.0 \xrightarrow{(a)(b)a\langle b\rangle} (a)0 \quad \text{and} \quad a(b).b!c.0 \xrightarrow{(a)a(b)} (a)(b)b!c.0$$

and again by rule (L-SCOPE-EXT)

$$(a)a\langle b\rangle.0 \xrightarrow{(b)a\langle b\rangle} (a)0 \quad \text{and} \quad (a)a(b).b!c.0 \xrightarrow{a(b)} (a)(b)b!c.0$$

Then, the two branches can synchronize by rule (L-AUTH)

$$(a)a\langle b\rangle.0 \mid (a)a(b).b!c.0 \xrightarrow{\tau_{(b)}} (a)0 \mid (a)(b)b!c.0$$

where the authorization for b is lacking to finish the synchronization. Since the respective authorization is scoping over the process, by (L-SCOPE-INT) we obtain

$$(b)((a)a\langle b\rangle.0 \mid (a)a(b).b!c.0) \xrightarrow{\tau} (a)0 \mid (a)(b)b!c.0$$

where the action on b is now authorized explicitly, and by (L-OUT), (L-SCOPE-EXT), (L-SCOPE), and (L-PAR) we conclude

$$(a)0 \mid (a)(b)b!c.0 \xrightarrow{b!c} (a)0 \mid (a)(b)0$$

Notice, however that we do not need the delegation in order for the final output on b to take place. If we consider process

$$(b)((a)a!b.0 \mid (a)a?x.x!c.0)$$

after the initial synchronization

$$(b)((a)a!b.0 \mid (a)a?x.x!c.0) \xrightarrow{\tau} (b)((a)0 \mid (a)b!c.0)$$

the right branch is "contextually" authorized by the floating authorization for b. Thus, by (L-OUT) (L-SCOPE), (L-PAR), and (L-SCOPE-EXT)

$$(b)((a)0 \mid (a)b!c.0) \xrightarrow{b!c} (a)0 \mid (a)(b)0$$

and we end up with the same configuration as with the first process.

3.4 Reduction semantics

This section introduces the reduction semantics of our process model. Informally, a reduction $P \rightarrow Q$ specifies that process P evolves to process Q in one computational step. The reduction relation relies on the *structural congruence* relation that allows for term manipulation. The structural congruence relation, denoted $\equiv$, is the least binary congruence on processes that satisfies the rules given in Table 3.3. All rules except the last three are standard also in the π-calculus (and also in C_π, cf. Table 2.3). These allow for: $(\mathcal{P}, \mid, 0)$ to be a commutative monoid, discard name restrictions scoping terminated process, two name restrictions to be swapped, name extrusion, and activation of one copy of a replicated input process. The last three rules address authorization scopes, and allow for: swapping the two authorizations, discarding unused authorizations and swapping authorization and name restriction, provided that the two specified names are distinct. These last three rules are adopted from [44]. Structural congruence also provides an insight to the main difference of our work with respect to the work in [44], which is to be explained after Example 3.4.1.

Considering the structural congruence rules in Table 3.3, we may notice that the scope of name restrictions can be extruded by rule (SC-RES-EXTR), but there is no similar rule for extruding the scope

of (a). This is a consequence of the fact that the former binds the name, while in the latter the specified name is free. We elaborate this difference in the next example.

Example 3.4.1 (Extrusion of authorization scope)**.** *Consider the process comprehending two branches*

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

In the right branch a fresh name a is created and one authorization for the same name. Applying structural congruence axiom (SC-RES-EXTR)*, the scope of name a can be extruded to the left branch, obtaining*

$$(b)b?x.x!c.0 \mid (\nu a)(a)(b)b!a.0 \equiv (\nu a)\left((b)b?x.x!c.0 \mid (a)(b)b!a.0\right)$$

On the contrary, even though the name a is not specified as free in the left branch, the scope of the authorization for a cannot be extruded in the same way, as the rules given in Table 3.3 do not allow for this to happen. Thus, we have the inequality

$$(\nu a)\left((b)b?x.x!c.0 \mid (a)(b)b!a.0\right) \not\equiv (\nu a)(a)\left((b)b?x.x!c.0 \mid (b)b!a.0\right)$$

To see why processes related with the last inequality are to be considered as having different behavior, observe that the lhs process evolves (by the standard π-calculus rule for communication) to

$$(\nu a)\left((b)a!c.0 \mid (a)(b)0\right)$$

where the output a is unauthorized. On the other hand, the rhs process evolves to

$$(\nu a)(a)\left((b)a!c.0 \mid (b)0\right)$$

where the action on a is authorized.

The last example shows that authorization scoping construct cannot be manipulated over the parallel composition in the same way as

the name restriction. In [44] the structural congruence relation included rule $(a)(P \mid Q) \equiv (a)P \mid (a)Q$. Adopting this rule in our model would represent introducing/discarding one authorization, thus interfering with our notion of authorization accounting. In our model, we distinguish $(a)(P \mid Q)$ where the authorization is floating over P and Q, while in $(a)P \mid (a)Q$ two authorizations are specified, one for each process. This leads us to believe that the authorization scoping construct cannot be manipulated over parallel composition statically, hence, without examining the behavior of the parallel branches, and also that the structural congruence relation cannot be used in an obvious way to obtain a normal form characterization of processes, as it is the case in [44]. This brings us to a novel development of a non-standard approach.

Intuitively, for reduction to take place the two active prefixes willing to synchronize must be properly authorized, i.e., in the scope of the respective authorizations. In order to statically determine if the communication can occur, we define *static contexts* and operator *drift*. The static context is a process with one (two) non-prefixed, not scoped with name restrictions and not replicated hole(s) in which processes can be instantiated (cf. Definition 2.6.7).

Definition 3.4.2 (Static Contexts)**.** *Static contexts with one and two holes are defined as follows.*

$$\begin{array}{rcl} \mathcal{C}[\cdot] & ::= & \cdot \mid (P \mid \mathcal{C}[\cdot]) \mid (a)\mathcal{C}[\cdot] \\ \mathcal{C}[\cdot_1, \cdot_2] & ::= & (\mathcal{C}[\cdot_1] \mid \mathcal{C}[\cdot_2]) \mid (P \mid \mathcal{C}[\cdot_1, \cdot_2]) \mid (a)\mathcal{C}[\cdot_1, \cdot_2] \end{array}$$

We annotate the holes with $\cdot_1$ and $\cdot_2$ 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 the definition of the contexts does not mention the name restriction construct (νa). This allows us to single out specific names and avoid name clashes. Remaining cases define 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

are used up to structural congruence. For instance, if in context $\mathcal{C}[\cdot_1, \cdot_2] = (b)((a)\ \cdot_1\ \mid (a)\,\cdot_2)$ we instantiate processes $a!b.0$ and $a?x.0$ we obtain process $(b)((a)a!b.0 \mid (a)a?x.0)$.

Operator *drift* has two roles. Firstly, the operator works as a predicate over contexts as it singles out contexts in which hole(s) is(are) under the scope of the requested authorizations. Secondly, when defined, the operator removes the specific authorizations from the original context, so as to capture confinement in the resulting context. In addition, in the presence of a larger number of the authorizations that are to be removed from the context, the operator removes the ones that occur nearest to the hole (for the purpose of fairness).

Operator *drift* for contexts with one hole is defined inductively on the structure of contexts by the rules given in Table 3.4. The operator

$$\mathit{drift}(\mathcal{C}[\cdot]; \rho; \rho')$$

takes a one-hole context $\mathcal{C}[\cdot]$ and two multisets of names ρ and ρ'. The multiset notation is used since the same name can appear more than once. The first multiset ρ contains the names of authorizations that the operator should remove from the context, while the second contains the names of authorizations that have already been removed.

We comment on the rules given in Table 3.4, reading from conclusion to premises.

- In rule (C-END) the operator is defined only if the first multiset is empty. Thus, the operator is defined only in case all authorizations from the first multiset have been actually removed from the context up to the point the hole is reached.

- In rule (C-REM) the operator removes the authorization from the context, that was specified in the first multiset. The removed name is passed from the first multiset ("to be removed") to the second multiset ("has been removed"). The multiset addition operation $\rho \uplus \{a\}$ (or just $\rho \uplus a$) sums the frequencies of the elements.

- In rule (C-SKIP) the operator seamlessly passes the authorization, but only if the name does not occur in the second multiset. The last restriction ensures that the removed authorizations are the ones nearest to the hole, since only authorizations that were not already removed proceeding towards the hole can be preserved. For instance, $\mathit{drift}((a)\ \cdot\ ; \emptyset; \{a\})$ is undefined since the authorization is not specified to be removed from the context by the operator (rule (C-REM) cannot be applied), and also the authorization is specified as already removed from the context (rule (C-SKIP) cannot be applied).
- In rule (C-PAR) the operator seamlessly proceeds towards the one of the branches.

When defining the operator one specifies the context and the names of authorizations that are to be removed, while the multiset of names of authorizations that have been removed is obviously empty. We illustrate the application of the operator with two simple examples.

Example 3.4.3 (*drift* for one hole contexts)**.**

$$\begin{aligned}
&\mathit{drift}((a)\cdot; a; \emptyset) = \cdot \\
&\mathit{drift}((a)((a)\ \cdot\ \mid R); a; \emptyset) = (a)(\cdot \mid R) \\
&\mathit{drift}(\cdot; a; \emptyset)\ \textit{is undefined} \\
&\mathit{drift}((a)(a)\cdot; a, a; \emptyset) = \cdot \\
&\mathit{drift}((a)\cdot; a, b; \emptyset)\ \textit{is undefined} \\
&\mathit{drift}((a)\ \cdot\ \mid (b)0; a, b; \emptyset)\ \textit{is undefined.}
\end{aligned}$$

Example 3.4.4 (*drift* derivation)**.** *Let us again consider the second application of the operator in the example above*

$$\mathit{drift}((a)((a)\ \cdot\ \mid R); a; \emptyset) = (a)(\cdot \mid R)$$

and let us show the derivation conducted. Necessarily, at the top of the derivation, only an axiom can be applied. The only axiom in Table 3.4

rule (C-END) *requires that the first multiset must be empty at the root, in order for the derivation to be defined. Observing all rules, we may notice that only* (C-REM) *manipulates names in the two multisets by transferring names from the first to the second multiset. Hence, the sum of the two multisets is preserved invariant in the rules. Thus, we can conclude the derivation in our example is rooted by* $drift(\cdot; \emptyset; a) = \cdot$ *by axiom* (C-END)*. We then consider the authorization* (a) *that directly scopes the hole, and by* (C-REM) *we have that*

$$\frac{drift(\,\cdot\,; \emptyset; a) = \cdot}{drift((a)\,\cdot\,; a; \emptyset) = \cdot}$$

In this case, rule (C-SKIP) *could not be applied instead, since* a *is in the second multiset. After that, the process in parallel in the context is handled by rule* (C-PAR)

$$\frac{drift((a)\,\cdot\,; a; \emptyset) = \cdot}{drift((a)\ \cdot\ \mid R; a; \emptyset) = \cdot \mid R}$$

Then, we can complete the derivation by rule (C-SKIP)

$$\frac{drift((a)\ \cdot\ \mid R; a; \emptyset) = \cdot \mid R}{drift((a)((a)\ \cdot\ \mid R); a; \emptyset) = (a)(\,\cdot \mid R)}$$

We may observe that starting the derivation from the bottom we could first apply (C-REM) *and deduce* $drift((a)((a)\ \cdot\ \mid R); a; \emptyset) = \mathcal{C}[\cdot]$*, for some* $\mathcal{C}$*. However, in that case the name* a *is transferred to the "has been removed" multiset, and moving up in the derivation, after applying* (C-PAR)*, the derivation gets stuck as none of the rules* (C-REM) *nor* (C-SKIP) *can be applied for the authorization that directly scopes the hole (cf. the example given in the explanation of rule* (C-SKIP)*).*

The only derivations for the operator we are interested in consider in the bottom of the derivation the set of removed authorizations is empty, while at the top of the derivation the set of to be removed

authorizations is empty. This implies that in order for the operator to be defined all the authorizations specified as to be removed from the context at the beginning were found and have been removed in the resulting context. Furthermore, in the derivation tree, the rule (C-REM) is always used above the rule (C-SKIP) for a given name, as ensured by the side condition of the latter rule.

Operator *drift* for contexts with two holes is defined by the rules given in Table 3.5. In this case, the operator

$$drift(\mathcal{C}[\cdot_1, \cdot_2]; \rho_1; \rho_2; \rho'_1; \rho'_2)$$

takes as arguments a two-hole context, two multisets of names ρ_1 and ρ_2 representing the names of authorizations which are to be removed and two multisets of names ρ'_1 and ρ'_2 representing names of authorizations already removed. Multisets indexed with 1 refer to the $\cdot_1$ hole, and multisets indexed with 2 refer to the $\cdot_2$ hole.

In rule (C2-SPL) the operator for the two-hole context, where the context is a parallel composition of two one-hole contexts, is decomposed into two operators for one-hole context, considering the multisets of names ρ_1 and ρ'_1 for the 1 indexed context, and ρ_2 and ρ'_2 for the 2 indexed context. The rest of the rules in Table 3.5 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. We illustrate the applications of the operator for contexts with two holes by the next example.

Example 3.4.5 (*drift* for two hole contexts)**.**

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

Rule (C2-SPL) makes the derivation for the case of two holes to rely on the derivations for the cases of one hole and is possible only if the axioms for empty contexts hold (cf. (C-END)). Hence, the operator for a two-hole context is undefined if the required authorizations for any of the holes are lacking. Again, when defining the operator, none of the authorizations have been removed and multisets ρ'_1 and ρ'_2 are empty. Therefore, we abbreviate $\mathit{drift}(\mathcal{C}[\cdot]; \rho; \emptyset)$ with $\mathit{drift}(\mathcal{C}[\cdot]; \rho)$ and $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \rho_1; \rho_2; \emptyset; \emptyset)$ with $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \rho_1; \rho_2)$.

At this point, we may now proceed to present the reduction rules. The reduction relation $(\rightarrow)$ is the least sub-relation of $\mathcal{P} \times \mathcal{P}$ that satisfies the rules shown in Table 3.6.

- In rule (R-COMM) the two processes can synchronize their dual actions sending and receiving b on name a only if the actions of both processes are authorized in the context, that is if both are under the scope of, at least one per each process, authorization for name a. The resulting process specifies the context where the two authorizations for a have been removed by the *drift* operator, and are confined to the continuations of the communication prefixes P and Q.

- Similarly to the previous rule, in (R-AUTH) the two processes can synchronize delegating and receiving authorization for name b on name a only if in the context the first hole is scoped with authorizations for b and a, and the second hole is scoped with authorization for a. The resulting process uses the context where the three authorizations have been removed by the *drift* operator. The authorization for b is explicitly exchanged since the operator removes the authorization for the delegating process, while the authorization is confined to the receiving process.

- In rule (R-STRU) the reduction relation is closed under structural congruence relation.

- Finally, in rule (R-NEWC) the reduction relation is closed under the restriction construct (νa).

When compared with the reduction rules for C_π given in Table 2.4, we may notice that here we do not have rules that deal with parallel composition and authorization scoping. These constructs are already addressed by the contexts in (R-COMM) and (R-AUTH). Similarly to C_π, here we also have no rule dealing with the replicated input, since using the structural congruence rule (SC-REP), a single copy of replicated process may be secluded and take a part in a synchronization captured by (R-COMM).

We may now show how a general form of replication $!P$ (used in the previous chapter) can be represented by the replicated input. If we consider process

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

where $a \notin \mathsf{fn}(P)$, we may observe that in one step it reduces to

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

where in parallel with the original process a copy of process P is activated. Hence, this way we may mimic the behavior of process $!P$.

When our work is compared with the previous works for authorizations, one may notice that reduction semantics of [44] relies on more standard machinery. Their structural congruence relation is expressive enough to isolate the active prefixes willing to synchronize, while our reduction relation relies on the defintions of static contexts and the *drift* operator. However, one may also notice that the LTS of [44] is more complex than the one presented in the previous section. As we already discussed, our definition of reduction relation relies on the novel technical machinery so as to cope with the notion of accounting we explore here. These differences are further elaborated with technical insights given in Section 3.5.

By an example, we now show a key difference in accounting principles considered in [44] and here.

Example 3.4.6 (On authorization accounting)**.** *As we have noted, reduction semantics of process algebra in [43, 44] introduces the structural congruence rule* $(a)(P \mid Q) \equiv (a)P \mid (a)Q$*. The logic behind is that an authorization for a name given to a process should make all of its threads authorized to use the name. Now consider that this implies*

$$(b)((a)a\langle b\rangle.P|(a)a(b).Q) \equiv (b)(a)a\langle b\rangle.P|(b)(a)a(b).Q \rightarrow (a)P|(b)(b)(a).Q$$

where in the original process one authorization for b *is shared between the two branches, but after applying the mentioned structural congruence rule and authorization delegation we end up in a situation where the rhs branch owns two authorizations for* b*. This is directly in conflict with our authorization accounting principle, since it allows changing the number of authorizations in the system. In our model, process*

$$(b)((a)a\langle b\rangle.P|(a)a(b).Q) \text{ can evolve in one step to } (a)P|(b)(a)Q$$

where the number of authorizations is stable, since upon delegation the only authorization for b *is confined to the process on the rhs.*

3.4.1 Harmony result

In Section 2.3 we have seen that the result of matching tau transitions of the LTS and reductions of the reduction semantics in C_π is simply inherited from the π-calculus (Theorem 2.3.1). In this section, we show the analogous result for our calculus of floating authorizations. We aim to prove that a process P reduces to Q if and only if P has fully authorized transition τ to a process that is structurally equivalent to Q. To this end we present several auxiliary results. Our first auxiliary result allows distinguishing free and bound names of the action.

Lemma 3.4.7 (Inversion on labelling)**.** *Let* $P \xrightarrow{\alpha} Q$*.*

1. *If* $\alpha = (a)^i a!b$ *then* $a, b \in \mathsf{fn}(P)$*.*
2. *If* $\alpha = (a)^i(\nu b)a!b$ *then* $a \in \mathsf{fn}(P)$ *and* $b \in \mathsf{bn}(P)$*.*

3. *If* $\alpha = (a)^i a?b$ *then* $a \in \mathsf{fn}(P)$.

4. *If* $\alpha = (a)^i (b)^j a\langle b\rangle$ *then* $a, b \in \mathsf{fn}(P)$.

5. *If* $\alpha = (a)^i a(b)$ *then* $a, b \in \mathsf{fn}(P)$.

Proof. The proof is by induction on the inference of $P \xrightarrow{\alpha} Q$, and it follows the same lines as proof of Lemma 2.2.3. □

We may now show that the structural congruence relation "agrees" with the LTS, in the sense that if a process can perform an action then a structurally equivalent process can perform the same action and the two resulting processes are again related by structural congruence.

Lemma 3.4.8 (LTS and structural congruence)**.** *If* $P \equiv P'$ *and* $P \xrightarrow{\alpha} Q$, *then there exists some* Q' *such that* $P' \xrightarrow{\alpha} Q'$ *and* $Q \equiv Q'$.

Proof. The proof is by induction on the length of the derivation of $P \equiv P'$. We only detail the case when the last applied rule is (SC-RES-EXTR), i.e., $P_1 \mid (\nu a)P_2 \equiv (\nu a)(P_1 \mid P_2)$, for $a \notin \mathsf{fn}(P_1)$. We have two possibilities to derive $(\nu a)(P_1 \mid P_2) \xrightarrow{\alpha} P'$, and that is by (L-RES) and (L-OPEN).

- *Case* (L-RES): Assume that $(\nu a)(P_1 \mid P_1) \xrightarrow{\alpha} (\nu a)R$, where $a \notin \mathsf{n}(\alpha)$ is derived from $(P_1 \mid P_2) \xrightarrow{\alpha} R$. Then, the possible transitions for $P_1 \mid P_2$ are:

  -(L-PAR): Assume $P_1 \mid P_2 \xrightarrow{\alpha} P_1 \mid P_2'$, where $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_1) = \emptyset$, is derived from $P_2 \xrightarrow{\alpha} P_2'$. We have that

  $$(\nu a)(P_1 \mid P_1) \xrightarrow{\alpha} (\nu a)(P_1 \mid P_2')$$

  Since $a \notin \mathsf{n}(\alpha)$, by (L-RES) we have $(\nu a)P_2 \xrightarrow{\alpha} (\nu a)P_2'$ and by (L-PAR)we conclude

  $$P_1 \mid (\nu a)P_2 \xrightarrow{\alpha} P_1 \mid (\nu a)P_2'$$

  If the symmetric case of rule (L-PAR) is applied, the proof follows the same lines.

-(L-COMM): Assume $P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2'$, where $\alpha = \tau_\omega$ and $\omega = (b)^{i+j}$, is derived from $P_1 \xrightarrow{\alpha_1} P_1'$ and $P_2 \xrightarrow{\alpha_2} P_2'$, where $\alpha_1, \alpha_2 \in \{(b)^i b!c, (b)^j b?c\}$. Then, we have

$$(\nu a)(P_1 \mid P_1) \xrightarrow{\tau_\omega} (\nu a)(P_1' \mid P_2')$$

By Lemma 3.4.7 we conclude $b \in \mathsf{fn}(P_1, P_2)$. Thus, from $a \notin \mathsf{fn}(P_1)$ we have that $b \neq a$. We now distinguish two cases:

* *if* $c = a$ *and* $\alpha_2 = (b)^i b!a$. By (L-OPEN) $(\nu a)P_2 \xrightarrow{(b)^i(\nu a)b!a} P_2'$ and by (L-CLOSE)

$$P_1 \mid (\nu a)P_2 \xrightarrow{\tau_\omega} (\nu a)(P_1' \mid P_2')$$

* *if* $c \neq a$, then by (L-RES) $(\nu a)P_2 \xrightarrow{\alpha_2} (\nu a)P_2'$ and by (L-COMM)

$$P_1 \mid (\nu a)P_2 \xrightarrow{\tau_\omega} P_1' \mid (\nu a)P_2'$$

- (L-CLOSE): Assume $P_1 \mid P_2 \xrightarrow{\alpha} (\nu c)(P_1' \mid P_2')$, where $\alpha = \tau_\omega$ and $\omega = (b)^{i+j}$, is derived from $P_1 \xrightarrow{\alpha_1} P_1'$ and $P_2 \xrightarrow{\alpha_2} P_2'$, where $\alpha_1, \alpha_2 \in \{(b)^i(\nu c)b!c, (b)^j b?c\}$. Hence,

$$(\nu a)(P_1 \mid P_1) \xrightarrow{\tau_\omega} (\nu a)(\nu c)(P_1' \mid P_2')$$

By Lemma 3.4.7 we have $b \in \mathsf{fn}(P_1, P_2)$. Since $a \notin \mathsf{fn}(P_1)$ and assuming all bound names are different, we conclude $a \notin \{b, c\}$. Thus, by (L-RES) $(\nu a)P_2 \xrightarrow{\alpha_2} (\nu a)P_2'$ and by (L-CLOSE) we conclude

$$P_1 \mid (\nu a)P_2 \xrightarrow{\tau_\omega} (\nu c)(P_1' \mid (\nu a)P_2')$$

- (L-AUTH): Assume $P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2'$, where $\alpha = \tau_\omega$ and $\omega = (b)^{i+j}(c)^k$, is derived from $P_1 \xrightarrow{\alpha_1} P_1'$ and $P_2 \xrightarrow{\alpha_2} P_2'$, where $\alpha_1, \alpha_2 \in \{(b)^i(c)^k b\langle c\rangle, (b)^j b(c)\}$. Thus,

$$(\nu a)(P_1 \mid P_1) \xrightarrow{\tau_\omega} (\nu a)(P_1' \mid P_2')$$

By Lemma 3.4.7 we have $c, b \in \mathsf{fn}(P_1, P_2)$. Since $a \notin \mathsf{fn}(P_1)$ we conclude $a \notin \{b, c\}$. Therefore, by (L-RES) $(\nu a)P_2 \xrightarrow{\alpha_2} (\nu a)P_2'$ and by (L-AUTH)

$$P_1 \mid (\nu a)P_2 \xrightarrow{\tau_\omega} P_1' \mid (\nu a)P_2'$$

- *Case* (L-OPEN): Assume that $(\nu a)(P_1 \mid P_2) \xrightarrow{(b)^i(\nu a)b!a} R$ is derived from $P_1 \mid P_2 \xrightarrow{(b)^i b!a} R$, where $a \neq b$. Since $a \notin \mathsf{fn}(P_1)$, by Lemma 3.4.7 we conclude $P_1 \mid P_2 \xrightarrow{(b)^i b!a} R$ could only be derived using rule (L-PAR). Hence, $R = P_1 \mid P_2'$ and $P_2 \xrightarrow{(b)^i b!a} P_2'$. Then, by (L-OPEN) $(\nu a)P_2 \xrightarrow{(b)^i(\nu a)b!a} P_2'$ and by (L-PAR) we conclude

$$P_1 \mid (\nu a)P_2 \xrightarrow{(b)^i(\nu a)b!a} P_1 \mid P_2'$$

□

Using the structural congruence relation, static contexts and the operator *drift* we can characterize a process performing an action and the process that is the result of the action. This is given in our next result.

Lemma 3.4.9 (Inversion on LTS)**.** *Let P and Q be processes.*

1. *If $P \xrightarrow{(a)a!b} Q$ then $P \equiv (\nu\tilde{d})\mathcal{C}[a!b.P']$ and $Q \equiv (\nu\tilde{d})\mathcal{C}[(a)P']$ and $drift(\mathcal{C}[\cdot]; a)$ is undefined.*

2. *If $P \xrightarrow{a!b} Q$ then $P \equiv (\nu\tilde{d})\mathcal{C}[a!b.P']$ and $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P']$, for $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$.*

3. *If $P \xrightarrow{(a)(\nu b)a!b} Q$ then $P \equiv (\nu\tilde{d})(\nu b)\mathcal{C}[a!b.P']$ and $Q \equiv (\nu\tilde{d})\mathcal{C}[(a)P']$ and $drift(\mathcal{C}[\cdot]; a)$ is undefined.*

4. *If $P \xrightarrow{(\nu b)a!b} Q$ then $P \equiv (\nu\tilde{d})(\nu b)\mathcal{C}[a!b.P']$ and $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P']$, for $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$.*

5. *If* $P \xrightarrow{(a)a?b} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a?x.P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}[(a)P'\{b/x\}]$ *and* $drift(\mathcal{C}[\cdot]; a)$ *is undefined.*

6. *If* $P \xrightarrow{a?b} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a?x.P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P'\{b/x\}]$, *for* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$.

7. *If* $P \xrightarrow{(b)a\langle b\rangle} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a\langle b\rangle.P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P']$, *for* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$ *and* $drift(\mathcal{C}[\cdot]; a, b)$ *is undefined.*

8. *If* $P \xrightarrow{(a)a\langle b\rangle} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a\langle b\rangle.P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P']$, *for* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; b)$ *and* $drift(\mathcal{C}[\cdot]; a, b)$ *is undefined.*

9. *If* $P \xrightarrow{a\langle b\rangle} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a\langle b\rangle.P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P']$, *for* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a, b)$.

10. *If* $P \xrightarrow{(a)a(b)} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a(b).P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}[(a)(b)P']$ *and* $drift(\mathcal{C}[\cdot]; a)$ *is undefined;*

11. *If* $P \xrightarrow{a(b)} Q$ *then* $P \equiv (\nu\tilde{d})\mathcal{C}[a(b).P']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)(b)P']$, *for* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$.

12. *If* $P \xrightarrow{\tau} Q$ *then either*

    - $P \equiv (\nu\tilde{d})\mathcal{C}[a!b.P', a?x.P'']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P', (a)P''\{b/x\}]$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$, *or*
    - $P \equiv (\nu\tilde{d})\mathcal{C}[a\langle b\rangle.P', a(b).P'']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P', (a)(b)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$.

13. *If* $P \xrightarrow{\tau_{(a)}} Q$ *then either*

    - $P \equiv (\nu\tilde{d})\mathcal{C}[a!b.P', a?x.P'']$ *and* $Q \equiv (\nu\tilde{d})\mathcal{C}^-[(a)P', (a)P''\{b/x\}]$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$, *or* $\mathcal{C}^-[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$, *and* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *is undefined, or*

- $P \equiv (\nu \tilde{d})\mathcal{C}[a\langle b\rangle.P', a(b).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P', (a)(b)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; \emptyset)$, *or* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; b; a)$, *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is undefined, or*
- $P \equiv (\nu \tilde{d})\mathcal{C}[b\langle a\rangle.P', b(a).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}^-[(b)P', (b)(a)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; b; b)$ *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; b, a; b)$ *is undefined.*

14. If $P \xrightarrow{\tau_{(a)(a)}} Q$ *then either*

- $P \equiv (\nu \tilde{d})\mathcal{C}[a!b.P', a?x.P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}[(a)P', (a)P''\{b/x\}]$ *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *is undefined, or*
- $P \equiv (\nu \tilde{d})\mathcal{C}[a\langle b\rangle.P', a(b).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P', (a)(b)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; b; \emptyset)$, *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is undefined, or*
- $P \equiv (\nu \tilde{d})\mathcal{C}[a\langle a\rangle.P', a(a).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P', (a)(a)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$ *or* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$, *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, a; a)$ *is undefined.*

15. If $P \xrightarrow{\tau_{(a)(b)}} Q$ *and* $a \neq b$ *then* $P \equiv (\nu \tilde{d})\mathcal{C}[a\langle b\rangle.P', a(b).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P', (a)(b)P'']$, *for* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$ *or* $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$, *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is undefined.*

16. If $P \xrightarrow{\tau_{(a)^2(b)}} Q$ *then* $P \equiv (\nu \tilde{d})\mathcal{C}[a\langle b\rangle.P', a(b).P'']$ *and* $Q \equiv (\nu \tilde{d})\mathcal{C}[(a)P', (a)(b)P'']$ *and* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is undefined.*

Proof. The proof is by induction on the derivation of $P \xrightarrow{\alpha} Q$. We comment just on the first two assertions.

1. Suppose $P \xrightarrow{(a)a!b} Q$ and let us show $P \equiv (\nu \tilde{d})\mathcal{C}[a!b.P']$, and $Q \equiv (\nu \tilde{d})\mathcal{C}[(a)P']$, and $drift(\mathcal{C}[\cdot]; a)$ is undefined. The base case follows by rule (L-OUT) $a!bP \xrightarrow{(a)a!b} (a)P$. Here, $a!bP = \mathcal{C}[a!bP]$ and $(a)P = \mathcal{C}[(a)P]$, where $\mathcal{C}[\cdot] = \cdot$. The operator $drift(\mathcal{C}[\cdot]; a)$ is undefined since the second parameter of the operator is not an empty multiset. For induction steps we have next cases of the last applied rule: (L-RES), (L-SCOPE) and (L-PAR).

- *Case* (L-RES): the proof immediately follows from the induction hypothesis.

- *Case* (L-SCOPE): here $P = (c)P_1$ and $Q = (c)Q_1$ and $P \xrightarrow{(a)a!b} Q$ is derived from $P_1 \xrightarrow{(a)a!b} Q_1$, where $c \neq a$. By induction hypothesis

  $P_1 \equiv (\nu \tilde{d})\mathcal{C}[a!b.P']$ and $Q_1 \equiv (\nu \tilde{d})\mathcal{C}[(a)P']$ and $drift(\mathcal{C}[\cdot]; a)$ is undefined.

  Considering all free and bound names are different, by (SC-SCOPE-AUTH) we conclude $(c)P_1 \equiv (\nu \tilde{d})(c)\mathcal{C}[a!b.P']$ and $(c)Q_1 \equiv (\nu \tilde{d})(c)\mathcal{C}[(a)P']$. For $\mathcal{C}'[\cdot] = (c)\mathcal{C}[\cdot]$ we have that

  $P \equiv (\nu \tilde{d})\mathcal{C}'[a!b.P']$ and $Q \equiv (\nu \tilde{d})\mathcal{C}'[(a)P']$ and $drift(\mathcal{C}'[\cdot]; a)$ is undefined

  since $c \neq a$.

- *Case* (L-PAR): follows by similar reasoning.

2. The base case follows by rule (L-SCOPE-INT): $(a)P \xrightarrow{a!b} Q$ is derived from $P \xrightarrow{(a)a!b} Q$. By statement 1. of this Lemma we have

$P \equiv (\nu \tilde{d})\mathcal{C}'[a!b.P']$ and $Q \equiv (\nu \tilde{d})\mathcal{C}'[(a)P']$ and $drift(\mathcal{C}'[\cdot]; a)$ is undefined.

By Lemma 3.4.7 we have $a \in \mathsf{fn}(P)$, thus we conclude $a \notin \tilde{d}$. If we define $\mathcal{C}[\cdot] = (a)\mathcal{C}'[\cdot]$, by (SC-SCOPE-AUTH) we have that $(a)P \equiv (\nu \tilde{d})\mathcal{C}[a!b.P']$. Now we only need to notice that

$$drift(\mathcal{C}[\cdot]; a) = \mathcal{C}'[\cdot]$$

Again, for the induction step we have that the last applied rule can be (L-RES), (L-SCOPE) or (L-PAR), and in all three cases the proof follows similar lines as in the first part. □

We are now ready to show the first implication of our main statement: if a process has τ transition, then it can also reduce to the same process.

Lemma 3.4.10 (Harmony: reduction)**.** *If* $P \xrightarrow{\tau} Q$ *then* $P \to Q$.

Proof. The proof is by induction on the derivation $P \xrightarrow{\tau} Q$. We have three base cases.

- *Case* (L-COMM): $P_1 \mid Q_1 \xrightarrow{\tau} P_2 \mid Q_2$ is derived from $P_1 \xrightarrow{\alpha} P_2$ and $Q_1 \xrightarrow{\overline{\alpha}} Q_2$, where $\alpha, \overline{\alpha} \in \{a!b, a?b\}$. By Lemma 3.4.9 we conclude that

$$P_1, Q_1 \in \{(\nu \tilde{d_1})\mathcal{C}_1[a!b.P_1'], (\nu \tilde{d_2})\mathcal{C}_2[a?x.Q_1']\} \quad \text{and}$$

$$P_2, Q_2 \in \{(\nu \tilde{d})\mathcal{C}_1^-[(a)P_1'], (\nu \tilde{d_2})\mathcal{C}_2^-[(a)Q_1'\{b/x\}]\}$$

  up to structural congruence relation, and where $\mathcal{C}_1^- = \mathit{drift}(\mathcal{C}_1[\cdot]; a)$ and $\mathcal{C}_2^- = \mathit{drift}(\mathcal{C}_2[\cdot]; a)$. If we define $\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}_1[\cdot_1] \mid \mathcal{C}[\cdot_2]$, by (SC-RES-EXTR) we have

$$(\nu \tilde{d_1})\mathcal{C}_1[a!b.P_1'] \mid (\nu \tilde{d_2})\mathcal{C}_2[a?x.Q_1'] \equiv (\nu \tilde{d_1}, \tilde{d_2})(\mathcal{C}[a!b.P_1', a?x.Q_1']) \quad \text{and}$$

$$(\nu \tilde{d_1})\mathcal{C}_1^-[(a)P_1'] \mid (\nu \tilde{d_2})\mathcal{C}_2^-[(a)Q_1'\{b/x\}] \equiv (\nu \tilde{d_1}, \tilde{d_2})\mathcal{C}^-[(a)P_1', (a)Q_1'\{b/x\}]$$

  where $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$. Then, by (R-COMM) and (R-NEWC) we conclude

$$(\nu \tilde{d_1}, \tilde{d_2})\mathcal{C}[a!b.P_1', a?x.Q_1'] \to (\nu \tilde{d_1}, \tilde{d_2})\mathcal{C}^-[(a)P_1', (a), Q_1'\{b/x\}]$$

- *Cases* (L-CLOSE) and (L-AUTH): follow by similar reasoning.

We have four cases for the last applied rule.

- *Case* (L-RES): $(\nu a)P \xrightarrow{\tau} (\nu a)Q$ is derived from $P \xrightarrow{\tau} Q$. By induction hypothesis $P \rightarrow Q$ and by (R-NEWC) we conclude $(\nu a)P \rightarrow (\nu a)Q$.

- *Case* (L-SCOPE-EXT): $(a)P \xrightarrow{\tau} Q$ is derived from $P \xrightarrow{\tau_{(a)}} Q$. By Lemma 3.4.9 we have three cases for the shape of processes P and Q. We only detail the first case, i.e., when

  $$P \equiv (\nu \tilde{d})\mathcal{C}[a!b.P', a?x.P''] \text{ and } Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P', (a)P''\{b/x\}]$$

  where $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$, or $\mathcal{C}^-[\cdot_1; \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$, and $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ is undefined. By rule (SC-SCOPE-AUTH) we derive

  $$(a)P \equiv (\nu \tilde{d})(a)\mathcal{C}[a!b.P', a?x.P'']$$

  Since $\mathit{drift}((a)\mathcal{C}[\cdot_1, \cdot_2]; a; a) = \mathcal{C}^-[\cdot_1, \cdot_2]$, the proof for this case follows.

- *Case* (L-SCOPE): $(c)P \xrightarrow{\tau} (c)Q$ is derived from $P \xrightarrow{\tau} Q$. By Lemma 3.4.9 we distinguish two cases. We comment only the first one, i.e., when

  $$P \equiv (\nu \tilde{d})\mathcal{C}[a!b.P_1, a?b.P_2] \text{ and } Q \equiv (\nu \tilde{d})\mathcal{C}^-[(a)P_1, (a)P_2\{b/a\}]$$

  where $\mathcal{C}^-[\cdot_1, \cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$. Then, by (SC-SCOPE-AUTH) we have

  $$(c)P \equiv (\nu \tilde{d})(c)\mathcal{C}[a!b.P_1, a?b.P_2] \text{ and } (c)Q \equiv (\nu \tilde{d})(c)\mathcal{C}^-[(a)P_1, (a)P_2\{b/a\}]$$

- *Case* (L-PAR): the proof is analogous as for (L-SCOPE).

□

To show the other direction of our main result, we need to be able to reason on the structure of the contexts that appear in the reduction rules (R-COMM) and (R-AUTH), for which the respective *drift* operator

is defined. In the next result, we characterize the structure of the context for which the *drift* of (R-COMM) is defined. The proof follows by a non-surprising induction on the structure of the contexts using the definition of the *drift* operator and is omitted.

Proposition 3.4.11 (Inversion on *drift*)**.** *Let* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *be defined and let* $\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}'[\mathcal{C}_1[\cdot] \mid \mathcal{C}_2[\cdot]]$*. Then, we distinguish four cases.*

- Case (a) in $\mathcal{C}_1$ and in $\mathcal{C}_2$*:*

  $$\mathcal{C}_1[\cdot] = \mathcal{C}'_1[(a)\mathcal{C}''_1[\cdot]] \text{ and } \mathcal{C}_2[\cdot] = \mathcal{C}'_2[(a)\mathcal{C}''_2[\cdot]]$$

  where $drift(\mathcal{C}''_1[\cdot]; a)$ *and* $drift(\mathcal{C}''_2[\cdot]; a)$ *are undefined.*

- Case (a) in $\mathcal{C}_1$ and not in $\mathcal{C}_2$*:*

  $$\mathcal{C}_1[\cdot] = \mathcal{C}'_1[(a)\mathcal{C}''_1[\cdot]] \text{ and } \mathcal{C}'[\cdot] = \mathcal{C}'_3[(a)\mathcal{C}''_3[\cdot]]$$

  where $drift(\mathcal{C}''_1[\cdot]; a)$*,* $drift(\mathcal{C}_2[\cdot]; a)$ *and* $drift(\mathcal{C}''_3[\cdot]; a)$ *are undefined.*

- Case (a) in $\mathcal{C}_2$ and not in $\mathcal{C}_1$*:*

  $$\mathcal{C}_2[\cdot] = \mathcal{C}'_2[(a)\mathcal{C}''_2[\cdot]] \text{ and } \mathcal{C}'[\cdot] = \mathcal{C}'_3[(a)\mathcal{C}''_3[\cdot]]$$

  where $drift(\mathcal{C}''_2[\cdot]; a)$*,* $drift(\mathcal{C}_1[\cdot]; a)$ *and* $drift(\mathcal{C}''_3[\cdot]; a)$ *are undefined.*

- Case (a) not in $\mathcal{C}_1$ and not in $\mathcal{C}_2$*:*

  $$\mathcal{C}'[\cdot] = \mathcal{C}_3[(a)\mathcal{C}'_3[\cdot]] \text{ and } \mathcal{C}'_3[\cdot] = \mathcal{C}_4[(a)\mathcal{C}'_4[\cdot]]$$

  where $drift(\mathcal{C}_1[\cdot]; a)$*,* $drift(\mathcal{C}_2[\cdot]; a)$*,* $drift(\mathcal{C}_4[\cdot]; a, a)$ *and* $drift(\mathcal{C}'_4[\cdot]; a)$ *are undefined.*

The same kind of property can also be stated for the case of (R-AUTH), by considering $\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];a,b;a)$ is defined and $\mathcal{C}[\cdot_1,\cdot_2] = \mathcal{C}'[\mathcal{C}_1[\cdot] \mid \mathcal{C}_2[\cdot]]$. Then, similarly as in the previous proposition, we may distinguish (up to) 14. cases for the distribution of the three authorizations (depending on if a and b are different names).

By the last result and Lemma 3.4.8 we are then able to prove that if a process can reduce it can also have a τ transition, where the two resulting processes are structurally equivalent. This is attested in our next result.

Lemma 3.4.12 (Harmony: LTS)**.** *If $P \to Q$ then there is Q' such that $Q \equiv Q'$ and $P \xrightarrow{\tau} Q'$.*

Proof. The proof is by induction on the derivation $P \to Q$. We obtain two base cases.

- *Case* (R-COMM):

  $$\mathcal{C}[a!b.P', a?x.Q'] \to \mathcal{C}^-[(a)P', (a)Q'\{b/x\}]$$

  where $\mathcal{C}^-[\cdot_1,\cdot_2] = \mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];a;a)$. By rules (L-OUT) and (L-IN) we have that

  $$a!b.P' \xrightarrow{(a)a!b} (a)P' \text{ and } a?x.Q' \xrightarrow{(a)a?b} (a)Q'\{b/x\}$$

  By Proposition 3.4.11 we distinguish four cases for the structure of the context $\mathcal{C}[\cdot_1,\cdot_2] = \mathcal{C}'[\mathcal{C}_1[\cdot] \mid \mathcal{C}_2[\cdot]]$. We comment only the case when

  $$\mathcal{C}_1[\cdot] = \mathcal{C}'_1[(a)\mathcal{C}''_1[\cdot]] \text{ and } \mathcal{C}_2[\cdot] = \mathcal{C}'_2[(a)\mathcal{C}''_2[\cdot]]$$

  where $\mathit{drift}(\mathcal{C}''_1[\cdot];a)$ and $\mathit{drift}(\mathcal{C}''_2[\cdot];a)$ are undefined. Thus, in contexts $\mathcal{C}''_1[\cdot]$ and $\mathcal{C}''_2[\cdot]$ the holes are not in the scope of authorizations (a). Proceeding by induction on contexts $\mathcal{C}''_1[\cdot]$ and $\mathcal{C}''_2[\cdot]$ using rules (L-PAR) and (L-SCOPE) we may show that

  $$\mathcal{C}''_1[a!b.P'] \xrightarrow{(a)a!b} \mathcal{C}''_1[(a)P'] \text{ and } \mathcal{C}''_2[a?x.Q'] \xrightarrow{(a)a?b} \mathcal{C}''_2[(a)Q'\{b/x\}]$$

By (L-SCOPE-EXT) we obtain

$(a)\mathcal{C}''_1[a!b.P'] \xrightarrow{a!b} \mathcal{C}''_1[(a)P']$ and $(a)\mathcal{C}''_2[a?x.Q'] \xrightarrow{a?b} \mathcal{C}''_2[(a)Q'\{b/x\}]$

Again, by induction on contexts $\mathcal{C}'_1[\cdot]$ and $\mathcal{C}'_2[\cdot]$ using rules (L-PAR) and (L-SCOPE) we have

$$\mathcal{C}'_1[(a)\mathcal{C}''_1[a!b.P']] \xrightarrow{a!b} \mathcal{C}'_1[\mathcal{C}''_1[(a)P']] \quad \text{and}$$

$$\mathcal{C}'_2[(a)\mathcal{C}''_2[a?x.Q']] \xrightarrow{a?b} \mathcal{C}'_2[\mathcal{C}''_2[(a)Q'\{b/x\}]]$$

By (L-COMM)

$\mathcal{C}'_1[(a)\mathcal{C}''_1[a!b.P']] \mid \mathcal{C}'_2[(a)\mathcal{C}''_2[a?x.Q']] \xrightarrow{\tau} \mathcal{C}'_1[\mathcal{C}''_1[(a)P']] \mid \mathcal{C}'_2[\mathcal{C}''_2[(a)Q'\{b/x\}]]$

and again applying induction on the structure of context $\mathcal{C}'[\cdot]$ and using rules (L-PAR) and (L-SCOPE) we conclude

$$\mathcal{C}[a!b.P', a?x.Q'] \xrightarrow{\tau} \mathcal{C}'[\mathcal{C}'_1[\mathcal{C}''_1[(a)P']] \mid \mathcal{C}'_2[\mathcal{C}''_2[(a)Q'\{b/x\}]]]$$

Now we just need to notice that $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a) = \mathcal{C}'[\mathcal{C}'_1[\mathcal{C}''_1[\cdot]] \mid \mathcal{C}'_2[\mathcal{C}''_2[\cdot]]$.

- *Case* (R-AUTH): apply similar reasoning.

In the induction step, we have two possibilities for the last applied rule.

- *Case* (R-NEWC): $(\nu a)P \to (\nu a)Q$ is derived from $P \to Q$. By induction hypothesis $P \xrightarrow{\tau} Q'$, for $Q \equiv Q'$. By (L-RES) we have $(\nu a)P \xrightarrow{\tau} (\nu a)Q'$. Since $\equiv$ is a congruence, from $Q \equiv Q'$ we conclude $(\nu a)Q \equiv (\nu a)Q'$.

- *Case* (R-STRU): $P \to Q$ is derived from $P \equiv P' \to Q' \equiv Q$. By induction hypothesis $P' \xrightarrow{\tau} Q''$, for $Q' \equiv Q''$. By Lemma 3.4.8 we get $P \xrightarrow{\tau} Q'''$, where $Q''' \equiv Q''$. Now we can conclude the case by noting that we have $Q''' \equiv Q'' \equiv Q' \equiv Q$.

□

We are now ready to state our Harmony result. The proof follows directly from Lemma 3.4.12 and Lemma 3.4.10.

Corollary 3.4.13 (Harmony)**.** *$P \to Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.*

3.4.2 Error processes

In this section, we define a notion of an error process, which is a process that uses its resources in an unauthorized way. In our model, in order for synchronization to take place the proper authorizations are needed. In the case of LTS semantics, a process cannot evolve internally, i.e., have a τ transition, if the actions are not authorized. In the case of the reduction semantics, a process cannot reduce if it does not have the proper authorizations. This is how we define the *error* processes, as processes that cannot reduce only because of the lack of the required authorizations. By the closer inspection of the reduction rules, we may notice that this characterization actually says that the premise of the rules (R-COMM) and (R-AUTH) is not satisfied and that the *drift* operator is not defined.

Definition 3.4.14 (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.*

The definition of the error processes directly relies on the definition of the reduction relation: the structural congruence relation and the definition of contexts are used to identify the prefixes ready to synchronize, for which the respective authorizations are missing as the application of *drift* is undefined.

An alternative would be to characterize the error processes using the LTS, as having only incomplete τ transitions, i.e., τ_ω transitions, when ω is different from $(a)^0(b)^0$. The rest of this section is devoted to showing this claim is correct. First, we identify some auxiliary results. Our first result shows that if a one-hole context, in which a prefixed process is inserted, does not provide the proper authorization(s), then the action of the overall process will also lack the authorization(s). The proof, which we omit, follows in expected lines by induction on the structure of the context.

Lemma 3.4.15 (Unauthorized processes: external actions)**.**

1. *If* $drift(\mathcal{C}[\cdot]; a)$ *is undefined then*
   - $\mathcal{C}[a!b.P'] \xrightarrow{(a)a!b} \mathcal{C}[P']$, *and*
   - $\mathcal{C}[a?x.P'] \xrightarrow{(a)a?b} \mathcal{C}[P'\{b/x\}]$, *and*
   - $\mathcal{C}[a(b).P'] \xrightarrow{(a)a(b)} \mathcal{C}[(a)(b)P']$.
2. *If* $drift(\mathcal{C}[\cdot]; a, b)$ *is undefined then either*
   - $drift(\mathcal{C}[\cdot]; a)$ *is undefined and* $drift(\mathcal{C}[\cdot]; b)$ *is undefined and* $\mathcal{C}[a\langle b\rangle.P'] \xrightarrow{(a)(b)a\langle b\rangle} \mathcal{C}[(a)P']$, *or*
   - $drift(\mathcal{C}[\cdot]; a)$ *is undefined and* $\mathcal{C}[a\langle b\rangle.P'] \xrightarrow{(a)a\langle b\rangle} \mathcal{C}^-[(a)P']$, *where* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; b)$, *or*
   - $drift(\mathcal{C}[\cdot]; b)$ *is undefined and* $\mathcal{C}[a\langle b\rangle.P'] \xrightarrow{(b)a\langle b\rangle} \mathcal{C}^-[(a)P']$, *where* $\mathcal{C}^-[\cdot] = drift(\mathcal{C}[\cdot]; a)$.

We are ready to characterize the internal actions of the process obtained by inserting two prefixed processes in a two-hole context that does not provide the proper authorizations. Our first result focuses on the case when the internal action is a result of name passing.

Lemma 3.4.16 (Unauthorized processes: name passing)**.**

(i) If $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$, $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$, *and* $drift(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$ *are all three undefined then* $\mathcal{C}[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} Q$, *for some* Q *and* $\omega = (a)(a)$.

(ii) If $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$, *and only one of* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; \emptyset)$ *or* $drift(\mathcal{C}[\cdot_1, \cdot_2]; \emptyset; a)$ *are undefined then* $\mathcal{C}[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} Q$, *for some* Q *and* $\omega = (a)$.

Proof. The proof proceeds by induction on the structure of context $\mathcal{C}[\cdot_1, \cdot_2]$. We detail only the first assertion.

- *Case* $\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}_1[\cdot_1] \mid \mathcal{C}_2[\cdot_2]$: by rule (C2-SPL) we observe that both $\mathit{drift}(\mathcal{C}_1[\cdot_1], a)$ and $\mathit{drift}(\mathcal{C}_2[\cdot_2], a)$ are undefined. By Lemma 3.4.15, we have that

  $$\mathcal{C}_1[a!b.P'] \xrightarrow{(a)a!b} \mathcal{C}_1[P'] \text{ and } \mathcal{C}_2[a?x.P''] \xrightarrow{(a)a?b} \mathcal{C}_2[P''\{b/x\}]$$

  By rule (L-COMM), we conclude

  $$\mathcal{C}_1[a!b.P'] \mid \mathcal{C}_2[a?x.P''] \xrightarrow{\tau(a)(a)} \mathcal{C}_1[P'] \mid \mathcal{C}_2[P''\{b/x\}]$$

- *Case* $\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}_1[\cdot_1, \cdot_2] \mid R$: by (C2-PAR) we may observe $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; a; a)$, $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; a; \emptyset)$, and $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; \emptyset; a)$ are all three undefined. By induction hypothesis, $\mathcal{C}_1[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} Q$, for some Q and $\omega = (a)(a)$. Finally, by (L-PAR) we derive $\mathcal{C}_1[a!b.P', a?x.P''] | R \xrightarrow{\tau_\omega} Q|R$.

- *Case* $\mathcal{C}[\cdot_1, \cdot_2] = (c)\mathcal{C}_1[\cdot_1, \cdot_2]$ and $a \neq c$: by (C2-SKIP), $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; a; a)$, $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; a; \emptyset)$, and $\mathit{drift}(\mathcal{C}_1[\cdot_1, \cdot_2]; \emptyset; a)$ are all three undefined. By induction hypothesis, $\mathcal{C}_1[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} Q$, for some Q and $\omega = (a)(a)$. Applying (L-SCOPE) the proof completes.

□

A similar result can be stated for the reduction that is a result of authorization delegation. The proof of our next lemma follows similar reasoning as the proof of Lemma 3.4.16.

Lemma 3.4.17 (Unauthorized processes: delegation)**.** *If* $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is undefined then* $\mathcal{C}[a\langle b\rangle.Q, a(b).R] \xrightarrow{\tau_\omega} Q$ *and* $\omega = (a)^i(b)^j$*, where* $i + j \geq 1$.

We can now state the main result.

Proposition 3.4.18 (Error Transitions)**.** *Process P is an error if and only if $P \xrightarrow{\tau_\omega} Q$ for some Q and $\tau_\omega \neq \tau$.*

Proof. ($\Leftarrow$) The proof follows directly from cases 13.-16. of Lemma 3.4.9. ($\Rightarrow$) If P is an error, then, by Definition 3.4.14, $P \equiv (\nu\tilde{c})\mathcal{C}[\alpha_a.P', \alpha'_a.P'']$, where we distinguish two cases:

1. $\alpha_a = a!b$, $\alpha'_a = a?x$ and $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ is undefined. By Lemma 3.4.16 we have that $\mathcal{C}[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} Q$, for some Q and $\omega \in \{(a), (a)(a)\}$. By successive application of (L-RES)

$$(\nu\tilde{c})\mathcal{C}[a!b.P', a?x.P''] \xrightarrow{\tau_\omega} (\nu\tilde{c})Q,$$

   and by Lemma 3.4.8 we conclude $P \xrightarrow{\tau_\omega} Q'$, for some Q' such that $Q' \equiv (\nu\tilde{c})Q$.

2. $\alpha_a = a\langle b\rangle$, $\alpha'_a = a(b)$ and $\mathit{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ is undefined. Follows the same reasoning as the previous case, by application of Lemma 3.4.17.

□

We remark that we adopt Definition 3.4.14 for the purpose of our typing analysis. The labeled transition system and the reduction semantics inform differently on our model. The labeled transition system is more explicit when considering authorization manipulation, while the reduction semantics explicitly identifies the two processes ready to synchronize, which makes the latter more suitable to be used in Section 3.6 in the proofs of our typing system.

3.5 Behavioral semantics

In Section 2.4 we introduced a behavioral equivalence relation in C_π, strong bisimilarity, that is actually the same relation as in the π-calculus. This section presents a preliminary investigation of the

behavioral semantics of our authorization model. We introduce the strong bisimilarity relation by relying on the labeled transition system given in the previous section. The definition of the strong bisimilarity here follows the same lines as Definition 2.4.1.

Definition 3.5.1 (Strong bisimilarity)**.** *A binary relation $\mathcal{R}$ is a strong bisimulation on processes if $\mathcal{R}$ is a symmetric relation and for all $(P, Q) \in \mathcal{R}$ the following holds:*

If $P \xrightarrow{\alpha} P'$, for some α and P' where $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$, then $Q \xrightarrow{\alpha} Q'$ for some Q' such that $(P', Q') \in \mathcal{R}$.

Strong bisimilarity, noted with $\sim$, is the union of all strong bisimulations.

Same as in the π-calculus (and in the C_π), the strong bisimilarity defined above is also an equivalence relation (cf. Proposition 2.4.2). The reflexive and symmetric property are direct from the definition and the transitive property can be shown, up to α-conversion, in expected lines. Our aim is to first show some standard properties for the strong bisimilarity defined in this section, as the ones given in Section 2.4. The first result shows that the strong bisimilarity embeds the structural congruence.

Proposition 3.5.2 (Structural congruence)**.** $\equiv \subseteq \sim$.

Proof. The proof proceeds by coinduction on the definition of strong bisimulation, showing that relation

$$\mathcal{R} = \{(P, Q) \mid P \equiv Q\}$$

is a strong bisimulation, hence $\mathcal{R} \subseteq \sim$. Let $(P, Q) \in \mathcal{R}$. By Lemma 3.4.8 we have that if $P \equiv Q$ and $P \xrightarrow{\alpha} P'$, then there exists some Q', such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$. Thus, $\mathcal{R}$ is a strong bisimulation relation. □

The last result attests that structurally congruent processes exhibit the same behavior and as such is a form of a sanity check of our reduction semantics. Next, we show another standard property, that the strong bisimilarity is a non-input congruence.

Theorem 3.5.3 (Non-input congruence)**.**

(a) If $P \sim Q$ then

1. $P \mid R \sim Q \mid R$
2. $(\nu a)P \sim (\nu a)Q$
3. $a!b.P \sim a!b.Q$
4. $(a)P \sim (a)Q$
5. $a\langle b\rangle.P \sim a\langle b\rangle.Q$
6. $a(b).P \sim a(b).Q$

(b) If $P\{b/x\} \sim Q\{b/x\}$, for all b, then

1. $a?x.P \sim a?x.Q$
2. $!(a)a?x.P \sim !(a)a?x.Q$

Proof. The proof is by coinduction on the definition of strong bisimulation.

(a) 1. Follows by coinduction considering the relation

$$\mathcal{R} = \{((\nu\tilde{a})(P \mid R), (\nu\tilde{a})(Q \mid R)) \mid P \sim Q\}$$

and showing that it is a strong bisimulation, i.e., $\mathcal{R} \subseteq \sim$. Let us assume that $((\nu\tilde{a})(P \mid R), (\nu\tilde{a})(Q \mid R)) \in \mathcal{R}$ and

$$(\nu\tilde{a})(P \mid R) \xrightarrow{\alpha} P_1 \tag{3.2}$$

for some P_1 and α, where $\mathsf{bn}(\alpha) \cap \mathsf{fn}((\nu\tilde{a})(Q \mid R)) = \emptyset$. We show that then there is some Q_1 such that $(\nu\tilde{a})(Q \mid R) \xrightarrow{\alpha} Q_1$ and $(P_1, Q_1) \in \mathcal{R}$. The proof is analogous when first considering an action of $(\nu\tilde{a})(Q \mid R)$. We distinguish three cases for deriving (3.2): either it is derived from the observation on P or on R or from the synchronization between

P and R.
-*Observation on* P : In this case we have

$$(\nu\tilde{a})(P \mid R) \xrightarrow{\alpha} (\nu\tilde{a}')(P' \mid R)$$

is derived from

$$P \xrightarrow{\alpha'} P'$$

where either $\alpha = \alpha'$ and $\tilde{a} = \tilde{a}'$, or $\alpha = (b)^i(\nu c)b!c$ and $\alpha' = (b)^i b!c$ and $\tilde{a} = \tilde{a}', c$. By $\mathsf{bn}(\alpha') \cap \mathsf{fn}(Q) = \emptyset$ and $P \sim Q$ we conclude that $Q \xrightarrow{\alpha'} Q'$ where $P' \sim Q'$. Thus, we can derive

$$(\nu\tilde{a})(Q \mid R) \xrightarrow{\alpha} (\nu\tilde{a}')(Q' \mid R)$$

and we conclude $((\nu\tilde{a}')(P' \mid R), (\nu\tilde{a}')(Q' \mid R)) \in \mathcal{R}$.
-*Observation on* R : Follows similar lines.
-*Synchronization between* P *and* R :
We give only the case when (3.2) is derived by (L-CLOSE), since other cases, including the symmetric case for (L-CLOSE), follow similar lines. Consider

$$(\nu\tilde{a})(P \mid R) \xrightarrow{\tau_\omega} (\nu\tilde{a})(\nu b)(P' \mid R')$$

is derived from

$$P \xrightarrow{(\nu b)\sigma_1} P' \qquad \text{and} \qquad R \xrightarrow{\overline{\sigma}_1} R'$$

Since without loss of generality we can assume $b \notin \mathsf{fn}(Q)$, by $P \sim Q$ we have $Q \xrightarrow{(\nu b)\sigma_1} Q'$ and $P' \sim Q'$. Thus,

$$(\nu\tilde{a})(Q \mid R) \xrightarrow{\tau_\omega} (\nu\tilde{a})(\nu b)(Q' \mid R')$$

and we conclude $((\nu\tilde{a})(\nu b)(P' \mid R'), (\nu\tilde{a})(\nu b)(Q' \mid R')) \in \mathcal{R}$.

2. Follows similar lines, by showing that relation

$$\mathcal{R} = \{((\nu a)P \sim (\nu a)Q) \mid P \sim Q\} \cup \sim$$

is contained in $\sim$ by conduction on the definition of strong bisimulation.

3. Follows by showing that relation

$$\mathcal{R} = \{(a!b.P, a!b.Q) \mid P \sim Q\} \cup \sim$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation. Notice that the pair of processes related by $\mathcal{R}$ are either in $\sim$, in which case we conclude the proof directly, or have only the same observable (the output), in which case the pair of continuing processes is in $\sim$.

4. Follows by showing that relation

$$\mathcal{R} = \{((a)P, (a)Q) \mid P \sim Q\} \cup \sim$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation. Let $((a)P, (a)Q) \in \mathcal{R}$. and let

$$(a)P \xrightarrow{\alpha} P'.$$

There are three cases for the last applied rule while deriving the latter. We detail only the case when the last applied rule is (L-SCOPE-INT). Then, $\alpha = \tau_\omega$ and $(a)P \xrightarrow{\tau_\omega} P'$ is derived from $P \xrightarrow{\tau_{\omega(a)}} P'$. By $P \sim Q$ we have that $Q \xrightarrow{\tau_{\omega(a)}} Q'$, where $P' \sim Q'$. By (L-SCOPE-INT) we have $(a)Q \xrightarrow{\tau_a} Q'$, which finishes the proof.

5. Follows by showing that relation

$$\mathcal{R} = \{(a\langle b\rangle.P, a\langle b\rangle.Q) \mid P \sim Q\} \cup \sim$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation. Note that both process $a\langle b\rangle.P$ and $a\langle b\rangle.Q$

have only one observable action leading them to processes $(a)P$ and $(a)Q$, which are bisimilar by $P \sim Q$ and statement 4. of this Theorem.

6. Follows similar lines as 5., with witnessing relation

$$\mathcal{R} = \{(a(b).P, a(b).Q) \mid P \sim Q\} \cup \sim$$

(b) 1. Follows the similar lines as 3., considering relation

$$\mathcal{R} = \{(a?x.P, a?x.Q) \mid (\forall b)P\{b/x\} \sim Q\{b/x\}\} \cup \sim$$

Notice that if $(a?x.P, a?x.Q) \in \mathcal{R}$ the two processes have only input observable. Performing the same input leads to processes $P\{b/x\}$ and $Q\{b/x\}$, for some b, that are bisimilar by the assumption.

2. Follows by showing that relation

$$\mathcal{R} = \{(R \mid !(a)a?x.P, S \mid !(a)a?x.Q) \mid R \sim S \wedge (\forall b)P\{b/x\} \sim Q\{b/x\}\}$$

is a strong bisimulation, i.e., $\mathcal{R} \subseteq \sim$. Let $(R \mid !(a)a?x.P, S \mid !(a)a?x.Q) \in \mathcal{R}$ and let

$$R \mid !(a)a?x.P \xrightarrow{\alpha} P' \tag{3.3}$$

for some P' and α such that $\mathsf{bn}(\alpha) \cap \mathsf{fn}(S \mid !(a)a?x.Q) = \emptyset$. We distinguish three cases for deriving (3.3): either it is derived from the observation on R or on $!(a)a?x.P$ or from the synchronization between R and $!(a)a?x.P$.
-*Observation on* R :

$$R \mid !(a)a?x.P \xrightarrow{\alpha} R' \mid !(a)a?x.P$$

is derived from $R \xrightarrow{\alpha} R'$. Since $\mathsf{bn}(\alpha) \cap \mathsf{fn}(S) = \emptyset$ and $R \sim S$ it follows that $S \xrightarrow{\alpha} S'$, where $R' \sim S'$. By $\mathsf{bn}(\alpha) \cap \mathsf{fn}(!(a)a?x.Q) = \emptyset$ we conclude

$$S \mid !(a)a?x.Q \xrightarrow{\alpha} S' \mid !(a)a?x.Q$$

and $(R' \mid !(a)a?x.P, S' \mid !(a)a?x.Q) \in \mathcal{R}$.
-*Observation on* $!(a)a?x.P$:

$$R \mid !(a)a?x.P \xrightarrow{a?b} R \mid (a)P\{b/x\} \mid !(a)a?x.P$$

is derived from $!(a)a?x.P \xrightarrow{a?b} (a)P\{b/x\} \mid !(a)a?x.P$. Then, we can also derive

$$S \mid !(a)a?x.Q \xrightarrow{a?b} S \mid (a)Q\{b/x\} \mid !(a)a?x.Q$$

and we only have to show that $R \mid (a)P\{b/x\} \sim S \mid (a)Q\{b/x\}$. Since $P\{b/x\} \sim Q\{b/x\}$ for any b, by statement (a) 4. of this Theorem, we have $(a)P\{b/x\} \sim (a)Q\{b/x\}$. Using $R \sim S$, statement (a) 1. of this Theorem and transitivity and commutativity of strong bisimilarity we conclude $R \mid (a)P\{b/x\} \sim S \mid (a)Q\{b/x\}$.
-*Synchronization between* R *and* $!(a)a?x.P$:
We will detail only the case when the synchronization is derived by rule (L-COMM). In this case

$$R \mid !(a)a?x.P \xrightarrow{\tau_\omega} R'|(a)P\{b/x\} \mid !(a)a?x.P$$

is derived from

$$R \xrightarrow{(a)^i a!b} R' \quad \text{and} \quad !(a)a?x.P \xrightarrow{a?b} (a)P\{b/x\} \mid !(a)a?x.P$$

By $R \sim S$ we derive $S \xrightarrow{(a)^i a!b} S'$, where $R' \sim S'$. Then, also

$$S \mid !(a)a?x.Q \xrightarrow{\tau_\omega} S'|(a)Q\{b/x\} \mid !(a)a?x.Q.$$

Similarly as in the previous case, we can show that $R'|(a)P\{b/x\} \sim S'|(a)Q\{b/x\}$.

□

Theorem 3.5.3 asserts that a computational context cannot distinguish between the behaviors of the bisimilar processes, as placing two bisimilar processes in the same context results in two processes that are also bisimilar. Thus, each language construct can be seen as a proper function of behavior, as their composition with equivalent (object) behaviors yields equivalent (image) behaviors.

As we noted in Section 3.4, our accounting principle makes it difficult to manipulate authorization scope construct over the parallel composition. Using the strong bisimilarity relation we can formalize these intuitions. In the following, by $P \not\sim Q$ we denote that P and Q are not bisimilar.

Proposition 3.5.4 (Behavioral inequalities)**.** *For each of the following inequalities there exist processes P and Q and name a that witness them.*

1. $(a)(P \mid Q) \not\sim (a)P \mid (a)Q$.
2. $(a)(a)(P \mid Q) \not\sim (a)P \mid (a)Q$.
3. $(a)(P \mid Q) \not\sim P \mid (a)Q$ *if* $a \notin \mathsf{fn}(P)$.
4. $(a)(a)P \not\sim (a)P$.
5. $(a)P \not\sim P$ *if* $a \notin \mathsf{fn}(P)$.

Proof. To prove each inequality we give a proper counter-example.

1. Consider processes $P = a!b.0$ and $Q = a?x.0$. Then, process $(a)P \mid (a)Q$ has τ-transition

   $$(a)a!b.0 \mid (a)a?x.0 \xrightarrow{\tau} (a)0 \mid (a)0$$

   while process $(a)(P \mid Q)$ has only τ-transition with pending authorization

   $$(a)(a!b.0 \mid a?x.0) \xrightarrow{\tau_{(a)}} (a)0 \mid (a)0$$

2. Consider processes $P = a\langle a\rangle.0$ and $Q = 0$. We have that

$$(a)(a)(P \mid Q) \xrightarrow{a\langle a\rangle} (a)0 \mid 0$$

while the only possible action for $(a)P \mid (a)Q$ is

$$(a)P \mid (a)Q \xrightarrow{(a)a\langle a\rangle} (a)0 \mid (a)0$$

3. Consider processes $P = b?x.x!c.0$ and $Q = 0$. Then, we have

$$(a)(P \mid Q) \xrightarrow{(b)b?a} (a)((b)a!c.0 \mid 0) \xrightarrow{a!c} (b)(a)0 \mid 0$$

while the only possible action for $P \mid (a)Q$ after receiving a on b is pending on authorization (a)

$$P \mid (a)Q \xrightarrow{(b)b?a} (b)a!c.0 \mid (a)0 \xrightarrow{(a)a!c} (b)(a)0 \mid (a)0$$

4. Consider $P = a\langle a\rangle.0$. We may observe

$$(a)(a)P \xrightarrow{a\langle a\rangle} (a)0 \quad \text{while} \quad (a)P \xrightarrow{(a)a\langle a\rangle} (a)0$$

5. Consider again process $P = b?x.x!c.0$. Then,

$$(a)P \xrightarrow{(b)b?a} (a)(b)a!c.0 \xrightarrow{a!c} (b)(a)0$$

On the other hand

$$P \xrightarrow{(b)b?a} (b)a!c.0 \xrightarrow{(a)a!c} (b)(a)0$$

□

The last proposition formally attests our reduction semantics design choices. The results given in 1. and 3. show that the structural

congruence rules that we do not adopt (that relate authorization scoping and parallel composition) as mentioned in Section 3.4 indeed are unsound. Regardless if an authorization is distributed to both or to a single branch of the parallel composition the obtained process may exhibit a different behavior. Even if the name specified in the authorization is not free in one of the branches, the mentioned distribution may affect the behavior of a process. We remark that $(a)(P \mid Q) \not\sim P \mid (a)Q$ given in 3. also hold when $a \in \mathsf{fn}(P)$ (e.g., $(a)(a!b.0 \mid 0) \not\sim a!b.0 \mid (a)0$). Statement 1. provides the main evidence that the structural congruence rule $(a)(P \mid Q) \equiv (a)P \mid (a)Q$ introduced in the operational semantics of [44] conflicts with our design choices, namely with our accounting principle. Notice also that the difference of authorization scoping and name restriction is exposed in 3., showing that scope extrusion of the first operator is not safe since the name specified in the authorization is free while the name restriction is a binder.

Statement 2. attests that even the symmetric distribution of two authorizations over parallel composition may change the process behavior. Our accounting authorizations principle is attested in 4.: providing a different number of the same authorization to a process can yield processes that exhibit different behaviors. Similarly to 3., statement 5. reflects the fact that the authorization is a non-binder construct. Therefore, as a result of name passing, authorizations for names not free in the process may eventually be used.

Next, we present several equations relating authorization scoping and active prefixes.

Proposition 3.5.5 (Behavioral (in)equalities)**.**

1. *For any process P, names a, b and prefix α_b such that $b \neq a$ and $\alpha_b \neq b\langle a \rangle$ we have that $(a)\alpha_b.P \sim \alpha_b.(a)P$.*

2. *There exist process P, name a and prefixes α_b, α_c, where a does not occur, such that $(a)\alpha_b.\alpha_c.P \not\sim \alpha_b.\alpha_c.(a)P$.*

3. *For any process P, name a and prefix α_b such that $\alpha_b \neq a\langle a \rangle$, we have that $(a)(a)\alpha_b.P \sim (a)\alpha_b.(a)P$.*

4. *For any process P and name a we have that* $(a)(a)a\langle a\rangle.P \sim (a)(a)a\langle a\rangle.(a)P$.

5. *For any process P, names $a, b, c_1, \ldots, c_n$ we have that* $(c_1)\ldots(c_n)(a)(b)a\langle b\rangle.P \sim (a)(b)a\langle b\rangle.(c_1)\ldots(c_n)P$.

Proof.

1. Follows by showing that relation

   $$\mathcal{R} = \{((a)\alpha.P, \alpha.(a)P) \mid \alpha \notin \{a!b, a?x, !(a)a?x, a\langle b\rangle, b\langle a\rangle, a(b)\}\} \cup \sim$$

   is contained in $\sim$ by coinduction on the definition of strong bisimulation. Note that the only action of both processes $(a)\alpha.P$ and $\alpha.(a)P$ is determined by the prefix α, and that the action is not pending on the authorization (a). Each action of one process can be mimicked by the other leading to the same process, hence the proof follows by reflexivity of strong bisimilarity.

2. Consider $\alpha = b?x$ and $\beta = x!c$. Then one possible transition for the first process is

   $$(a)\alpha.\beta.P \xrightarrow{(b)b?a} (a)(b)a!c.P\{a/x\} \xrightarrow{a!c} (b)(a)P\{a/x\}$$

   while

   $$\alpha.\beta.(a)P \xrightarrow{(b)b?a} (b)a!c.(a)P\{a/x\} \xrightarrow{(a)a!c} (b)(a)P\{a/x\}$$

3. The witnessing relation in this case is

   $$\mathcal{R} = \{((a)(a)\alpha.P, (a)\alpha.(a)P) \mid \alpha \in \{a!b, a?x, a\langle c\rangle, c\langle a\rangle, a(b)\}, c \neq a\} \cup \sim$$

4. The proof follows by considering witnessing relation

   $$\mathcal{R} = \{((a)(a)(a)a\langle a\rangle.P, (a)(a)a\langle a\rangle.(a)P)\} \cup \sim$$

5. Follows directly from statements 1., 2., and 4. of this Proposition, Theorem 3.5.3 and Proposition 3.5.2.

□

The last proposition shows principles that can be used when trying to obtain a semantic normal form characterization of processes. Statements 1., 3. and 4. attest that an authorization can be pushed and pulled across the active prefix if the authorization is not needed to perform the action specified by the prefix, or in the presence the needed authorizations. Statement 2. shows that the authorization can only be pushed across the immediately active prefix. The first four statements are generalized in 5.: in the presence of the required authorizations all others can be pushed across the active prefix.

To conclude, we remark that the inequalities we have presented in this section can be seen as a justification of our novel approach to define the reduction semantics (using contexts and the *drift* operator) since a normal form characterization of processes in our model seems hard to obtain. Regardless of the fact that the equalities given in Proposition 3.5.5 show we can manipulate authorizations over active prefixes, the inequalities given in Proposition 3.5.4 inform on the difficulty in manipulating authorization scoping over parallel composition.

3.6 Type analysis

Thus far, we have identified the syntax and semantics of our process algebra aiming to model floating authorizations. We also recognized undesired configurations as error processes (Definition 3.4.14) in which resources are used in an unauthorized way. In that perspective, a natural question arises if there is a way of identifying processes in which all possible usages of channels are properly authorized. In this section, we introduce a typing discipline that can statically identify *safe* processes, that are not errors and never reduce to errors. This section is divided into five parts. In Section 3.6.1 we give a brief overview of the

background on type theory and type systems. Section 3.6.2 presents the idea of our types by means of examples. The type system is then formalized in Section 3.6.3, and the results are given in Section 3.6.4. We further elaborate on our typing principles in Section 3.6.5. In Section 3.6.6 we propose another type system that allows for a more efficient type-checking procedure, and we show the equivalence of the two type systems in Section 3.6.6.1.

3.6.1 Background on types

Type theory, in computer science also referred to as type systems, covers the wide field of mathematics, logic and computer science. The first type system appears at the beginning of the 20th century in the work of Whitehead and Russel [115], presenting the ramified type theory as a response to Russell's paradox. The core idea there was to avoid interpreting mathematical entities as sets, but instead to assign a *type* to any mathematical entity. The idea of using the type theory as an alternative to the set theory as a fundamental of mathematics has been explored also by Chwistek and Ramsey in work on the simple theory of types [28, 93], Church in the simply typed lambda calculus [27], Martin-Löf in the intuitionistic type theory [64], Coquand in the calculus of constructions [29], the homotopy-type theory [107], etc.

In computer science, type systems are extensively studied in many aspects. An important direction in the study of type systems is the Curry-Howard correspondence, that connects the two distinct research fields: computation and logic [22, 31, 45, 57, 112]. Type theories have also been used as foundations for developing programming languages. For instance, Agda [19] is based on the unified theory of depended types [63], that is an extension of Martin-Löf's type theory. Also, an interactive theorem prover Coq [13] works within Coquand's calculus of constructions.

Another application directed field of the study of type systems is within the programming languages field. Many programming languages have integrated type systems that ensure programs do not go

wrong. The type system is responsible for excluding bad behaviors called *type errors*. A typing system maps the language terms into types and "can be regarded as calculating a kind of static approximation to the run-time behaviors of the terms in a program" [84]. A type system should offer the *safety* guarantee, i.e., that a typable program does not produce errors at run-time. Another guarantee, called *completeness*, ensures that a type system does not discard any programs that behave well at run time, but is sometimes hard or not worthwhile to obtain (e.g., because of the cost). Let us notice that a type system does not prevent all possible bad behaviors, but only specific ones. The design of the type system heavily depends on the kind of errors one wants to eliminate. When compared to traditional programs, concurrent programs may introduce new kinds of errors, e.g., races and deadlocks, which can be hard to deal with.

Various type systems have been proposed for the π-calculus based process models. The first one is a sorting system [70] proposed by Milner, developed to deal with arity mismatch errors in the polyadic π-calculus. After that, types have been used to impose control on the usages of the π-calculus channels, including: linearity [59], groups [26], i/o types [85], etc. Later on, more sophisticated type disciplines appeared, where types more precisely inform on how the channels are to be used. Such types are also referred to as behavioral types [58], which includes session types [24, 56, 103] and conversation types [23].

The type systems usually use a notion of a type assignment, typing environment, and a typing judgment. By writing $a : T$ we mean that a type T has been assigned to name a. A typing environment Δ is a finite collection of the type assignments, and writing $\Delta(a) = T$, or equivalently $\Delta \vdash a : T$, we mean that in environment Δ type T has been assigned to name a. A typing judgment $\Delta \vdash P$ means that process P uses its channels as prescribed by Δ. These notions are used in what follows.

3.6.2 Introducing types by examples

To informally introduce our type analysis we return to the university scenario presented in Section 3.1. Let us consider process

$$(exam)(minitest)(alice)alice?x.x!value.0$$

that can receive a channel and afterwards output *value* on the received channel. The reception on *alice* is authorized directly, since the respective authorization is present. On the other hand, the later output action is authorized only for names *exam* and *minitest*. If on channel *alice* only *exam* or *minitest* can be received, then the two specified authorizations suffice. Depending on the received name, one of the authorizations can be used. If a name *viva* is received then the two authorizations obviously cannot authorize the later output action.

We may conclude the above process is authorization safe if it is placed in a context that matches the assumptions for names communicated in *alice*. This is the information our types record: which are the names that can be safely communicated in a channel. For the process above, we may say that only names *exam* and *minitest* can be communicated in channel *alice*. Furthermore, if we take that *exam* and *minitest* can only receive values that are not subject to authorization control, then $\{alice\}(\{exam, minitest\}(\emptyset))$ can represent the type of channel *alice*. Here, the type registers that *alice* is a final name (cf. type of x below), and that the channel can be used to communicate names *exam* and *minitest*. The last information the above type carries is that *exam* and *minitest* cannot be used for communication (typed with $\emptyset$). Thus, the type information can ensure that the above process is safe since all names that will possibly be used for communications are *contextually authorized*.

We now return to analyze the use of the input variable x. Considering that x can be substituted by either *exam* or *minitest* (that, as noted above, are not to be used for channel communication) the type of x is $\{exam, minitest\}(\emptyset)$. Here, we need to consider all possible replacements of a name, so as to uniformly address names that are

bound in inputs. Our types, denoted $\gamma(T)$, are having 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 type assignment

$$alice : \{alice\}(\{exam, minitest\}(\emptyset))$$

provides the information on the contexts in which the process above can be safely placed. For example, we may compose the above process with $(alice)alice!minitest$ where on *alice* name *minitest* can be sent, since *minitest* is one of the names expected on *alice*. Let us now consider process

$$(exam)(minitest)((alice)alice?x.x!value.0 \mid (bob)bob?x.x!value.0) \tag{3.4}$$

where authorizations for *exam* and *minitest* are now shared between the two threads, that both receive a name and then output a value on the received name. This process is also safe if the typing assignments

$$alice : \{alice\}(\{exam\}(\emptyset)) \text{ and } bob : \{bob\}(\{minitest\}(\emptyset))$$

are respected also by the context in which process is inserted. The assumptions in types specifies what a process anticipates from a (typed) context. Thus, we may also type the process (3.4) with assumptions

$$alice : \{alice\}(\{minitest\}(\emptyset)) \text{ and } bob : \{bob\}(\{exam\}(\emptyset)).$$

where now context should provide only *minitest* is to be sent on *alice* and *exam* is to be sent on *bob*.

We remark that the information of which student will be using which authorization is something that is not statically prescribed in the system, since the students share the authorizations. However, if both names *exam* and *minitest* are used by the two students we may ensure that the system is authorization safe, since the students will grab the corresponding authorizations (one per each student). Each of

the typing specifications above ensures that only *exam* and *minitest* can be received by the students, but also clearly provide a specific association between which name can be received by each student (*Alice* can receive *exam* and that *Bob* can receive *minitest*, or with the order reversed).

3.6.3 Typing discipline

Following the intuition provided in the previous section, in this section we formalize our typing system. Before that, we first introduce some auxiliary notions that deal with name generation: *symbol* annotations and *well-formedness*.

We may observe that our process model includes name restrictions and our types carry name identifiers. Since bound names are subject to α-conversion, we introduce a symbolic representation of bound names when they are carried in type specifications. Without loss of generality, we refine the process model for the purpose of the type analysis by adding an explicit symbolic representation of name restrictions. This allows us to avoid more involved handling of bound names in typing environments.

To this end, we introduce a countable set of symbols $\mathcal{S}$, disjoint with the set of names $\mathcal{N}$. We let $\mathbf{r}, \mathbf{s}, t, \ldots$ range over $\mathcal{S}$. Also, we introduce a special symbol κ, that is not in $\mathcal{N} \cup \mathcal{S}$. We introduce a unique association of restricted names and symbols, by refining the syntax of the construct $(\nu a)P$ with two constructs

$$(\nu a : \mathbf{r})P \quad \text{and} \quad (\nu a : \kappa)P$$

tagged with a symbol from $\mathcal{S}$ or with symbol κ, respectively. By $\mathsf{sym}(P)$ we denote the set of all symbols from $\mathcal{S}$ in process P. Names associated with symbols from $\mathcal{S}$ may be provided contextual authorizations (cf. rule (T-NEW) below), while names associated with symbol κ may not (cf. rule (T-NEW-REP) below).

In this section, we adopt the reduction semantics, which we adapt here to consider decorated name restrictions. We refine definition of

structural congruence by omitting 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 omitting of (SC-RES-INACT) is not new in process models where name restriction carries a type information (cf. [3]).

Since we want symbols from $\mathcal{S}$ to uniquely represent restricted channels, we are actually interested only in processes that have unique occurrences of such symbols.

Definition 3.6.1 (Well-formedness)**.** *A process is* well-formed *if it has unique occurrences of symbols from $\mathcal{S}$, and no occurrences of symbols from $\mathcal{S}$ in the body of replicated inputs.*

We may now show that well-formedness is invariant with respect to (adapted) structural congruence and reduction.

Proposition 3.6.2 (Preservation of well-formedness)**.** *If P is well-formed and $P \equiv Q$ or $P \to Q$ then Q is also well-formed and $\mathsf{sym}(P) = \mathsf{sym}(Q)$.*

Proof. The proof is by induction on the derivations, performing the case analysis on the last rule applied. We detail only the case when the last applied reduction rule is (R-NEWC). Let $P = (\nu a : \mathbf{r})P'$, $Q = (\nu a : \mathbf{r})Q'$, and $P \to Q$ be derived from $P' \to Q'$. Since P is well-formed $\mathbf{r} \notin \mathsf{sym}(P')$ and P' is well-formed. By induction hypothesis we have that Q' is well-formed and $\mathsf{sym}(P') = \mathsf{sym}(Q')$. Thus, $\mathbf{r} \notin \mathsf{sym}(Q')$, Q is well-formed and $\mathsf{sym}(P) = \mathsf{sym}(Q)$. □

The syntax of types is formally defined as

$$\gamma ::= \varphi \mid \kappa \quad \text{and} \quad T ::= \gamma(T) \mid \emptyset$$

where $\varphi \subset \mathcal{N} \cup \mathcal{S}$. Types inform on safe instantiations of names that are subject to contextual authorizations. Type γ stands for a subset

of $\mathcal{N} \cup \mathcal{S}$, in which case is denoted with φ, or symbol 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 $a : T$. If $a : \gamma(T)$, then for $\gamma = \varphi$ the set γ characterizes with what names a may be instantiated. If instead $\gamma = \kappa$, then a is not subject to contextual authorizations. By $names(T)$ we denote the set of names that occur in T and by $names(\Delta)$ the set of names that occur in all entries of Δ.

Table 3.7 shows the rules for our typing system. A typing judgment $\Delta \vdash_\rho P$ states that P uses channels as prescribed by Δ and that P is safe if it is placed in a context that provides the authorizations given in ρ, which is a multiset of names (from $\mathcal{N}$). To illustrate why a notion of multiset is required consider that process $a!b.0 \mid a?x.0$ can be typed with $a : \{a\}(\{b\}(\emptyset)), b : \{b\}(\emptyset) \vdash_\rho a!b.0 \mid a?x.0$ where certainly $\{a, a\}$ is contained in ρ, identifying that the process can only be inserted in contexts that provide two authorizations on name a, for one sending and the other for receiving on a.

We now describe the typing rules.

- Axiom (T-STOP) asserts that a terminated process can be typed by any Δ and ρ.

- In rule (T-PAR) processes P_1, P_2 and their parallel composition are typed using the same Δ. Furthermore, if P_1 and P_2 are safe when inserted in contexts that provide authorizations ρ_1 and ρ_2, respectively, then $P_1 \mid P_2$ is safe if inserted in the context that provides the sum of authorizations from ρ_1 and ρ_2. The well-formedness is ensured with $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$.

- In rule (T-NEW) process P is typed with an environment that includes an entry for a, and process $(\nu a : \mathbf{r})P$ is typed by removing that entry and replacing each occurrence of name a in the environment by the corresponding symbol $\mathbf{r}$. We use $\Delta\{\mathbf{r}/a\}$ to denote the environment obtained by substituting a by $\mathbf{r}$ in every

assumption in Δ, hence in every type. The well-formedness is ensured by $\mathbf{r} \notin \mathsf{sym}(P)$. Condition $a \notin \rho, names(T)$ claims that the context cannot provide authorization for name a and ensures consistency of the typing assumption.

- The only difference of rule (T-NEW-REP) with respect to rule (T-NEW) is that no replacement is performed in Δ and that a is also not mentioned in the environment. The environment must already refer to symbol κ in whatever pertains to the restricted name. For instance, process $(\nu b : \kappa)(a)a!b.0$ can be typed only if the assumption for a is $\gamma(\kappa(T))$, for some γ and T, where κ represents that the names communicated in a (hence, including b) are never subject to contextual authorizations.

- In rule (T-AUTH) processes $(a)P$ and P are typed using the same Δ, since authorization scoping is non-binding construct. If P is safe when the context provides authorizations $\rho \uplus \{a\}$, then $(a)P$ can be safely placed in a context that provides authorizations ρ.

- In rule (T-OUT) process P is typed using an environment with entries for a and b. In the type of b all possible instantiations for the name (identified in γ'') are safe to be communicated on name a (formalized by $\gamma'' \subseteq \gamma'$, where γ' is from the type of a). Furthermore, the continuation type T of b must be the same as T given in the type of a. Then, process $a!b.P$ is typed using the same environment. The rule specifies two ways in which the action on a is authorized: (1) authorization is provided by the context directly ($a \in \rho$); (2) authorizations for all instantiations of name a are provided by the context ($\gamma \subseteq \rho$). To illustrate case (1) consider that in $(b)b?a.(a)a!c.0$ the latter output is authorized regardless of name replacements. For case (2), consider that if we assume a can be replaced only by d, processes $(b)b?a.(d)a!c.0$ and $(b)(d)b?a.a!c.0$ are safe, while $(b)b?a.(e)a!c.0$ and $(b)(e)b?a.a!c.0$ are not. The option given in case (2) is fundamental to address contextual authorizations. Notice that the

condition $\gamma \subseteq \rho$ implies γ contain only names and no symbols, since ρ is defined as a multiset of names.

- Rule (T-IN) follows similar principles as the previous one.

- In rule (T-REP-IN) process P is typed under an environment with an entry for x and the process is safe if inserted in a context that provides one authorization for a. Then, the replicated input is typed by removing the entry for x from the environment, which must precisely match the specification given in the type of a. The input variable should not be mentioned in any type, provided with $x \notin \rho, names(\Delta)$. The restriction that P contains no symbols from $\mathcal{S}$ ensures the unique association of symbols and names when copies of the replicated process are activated (see example (3.8) in Section 3.6.5). Process $!(a)a?x.P$ can be inserted in any context that conforms with Δ and provides any authorizations.

- In rules (T-DELEG) and (T-RECEP) the typing environment does not change from premises to conclusion. The authorization(s) required to use the subject name a is(are) addressed in the same way as in rules (T-OUT) and (T-IN). The authorization for b in rule (T-RECEP) is handled as in rule (T-AUTH). In rule (T-DELEG), the authorization for b is added to ρ, as the context in which the process may be inserted needs to provide an additional authorization for this name.

We may now observe that the symbolic representation of a bound name in the typing environment reflects the fact that a contextual authorization cannot be provided by the process that receives such (unforgeable) name via name extrusion. For instance, process $(a)(d)a?x.x!c.0 \mid (\nu b : \mathbf{r})(a)a!b.0$ is not safe, as regardless of provided contextual authorization (that includes (a) and (d)) for the received name in the left branch, the name sent by the right branch is fresh and necessary different from all names of the mentioned authorizations. This process is 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.

3.6.4 Type safety

In this section, we provide a proof that a typed process is safe, i.e., that it is not an error and that it never reduces to an error. First, we assert that a typed process is always well-formed. The proof follows directly from the typing rules.

Proposition 3.6.3 (Typed implies well-formed)**.** *If $\Delta \vdash_\rho P$ then P is well-formed.*

We now introduce a notion of *well-typed* processes.

Definition 3.6.4 (Well-typed processes)**.** *Process P is well-typed if $\Delta \vdash_\emptyset P$ and Δ only contains assumptions of the form $a : \{a\}(T)$ or $a : \kappa(T)$.*

The assumption that a typed process is at top level also well-typed, used in the later results, should seem as natural. Entries of the typing environment are associated only to free names of the typed process, that at top level are all names of channels. Hence, names are typed by the assumption $a : \{a\}(T)$ (cf. (T-NEW)), or by $a : \kappa(T)$ (cf, (T-NEW-REP)). Also, at top-level, a typed process should own enough authorizations for all of his actions. In that case, no authorizations need to be provided by the context ($\rho = \emptyset$).

We now proceed to present our results. First we identify the properties that follow directly from the typing rules.

Lemma 3.6.5 (Inversion on typing)**.** *Directly from Table 3.7 we have the following.*

1. *If* $\Delta \vdash_\rho (\nu a : \mathbf{r})P$ *then* $\Delta', a : \{a\}(T) \vdash_\rho P$, *where* $\Delta = \Delta'\{\mathbf{r}/a\}$ *and* $\mathbf{r} \notin \mathsf{sym}(P)$ *and* $a \notin \rho, names(T)$.

2. *If* $\Delta \vdash_\rho (\nu a : \kappa)P$ *then* $\Delta, a : \kappa(T) \vdash_\rho P$, *where* $a \notin \rho, names(T, \Delta)$.

3. *If* $\Delta \vdash_\rho (a)P$ *then* $\Delta \vdash_{\rho \uplus \{a\}} P$.

4. *If* $\Delta \vdash_\rho a!b.P$ *then* $\Delta \vdash_\rho P$, *where* $\Delta(a) = \gamma(\gamma'(T))$, $\Delta(b) = \gamma''(T)$, $\gamma'' \subseteq \gamma'$ *and if* $a \notin \rho$ *then* $\gamma \subseteq \rho$.

5. *If* $\Delta \vdash_\rho a?x.P$ *then* $\Delta, x : T \vdash_\rho P$, *where* $\Delta(a) = \gamma(T)$, $x \notin \rho, names(\Delta)$ *and if* $a \notin \rho$ *then* $\gamma \subseteq \rho$.

6. *If* $\Delta \vdash_\rho !(a)a?x.P$ *then* $\Delta, x : T \vdash_{\{a\}} P$ *where* $\mathsf{sym}(P) = \emptyset$ *and* $\Delta(a) = \gamma(T)$ *and* $x \notin \rho, names(\Delta)$.

7. *If* $\Delta \vdash_\rho a\langle b\rangle.P$ *then* $\Delta \vdash_{\rho'} P$, $\Delta(a) = \gamma(T)$, *where* $\rho = \rho' \uplus \{b\}$ *and if* $a \notin \rho'$ *then* $\gamma \subseteq \rho'$.

8. *If* $\Delta \vdash_\rho a(b).P$ *then* $\Delta \vdash_{\rho \uplus \{b\}} P$, *where* $\Delta(a) = \gamma(T)$ *and if* $a \notin \rho$ *then* $\gamma \subseteq \rho$.

9. *If* $\Delta \vdash_\rho P_1 \mid P_2$ *then* $\Delta \vdash_{\rho_1} P_1$ *and* $\Delta \vdash_{\rho_2} P_2$, *where* $\rho = \rho_1 \uplus \rho_2$ *and* $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$.

The following two results (Weakening and Strengthening) are fundamental to prove Subject Congruence, which in turn is crucial to prove Subject Reduction. The Weakening result shows how in the typing judgment the typing environment Δ can be enlarged by an entry, and, also, that the multiset ρ can be enlarged. The Strengthening result shows how in the typing judgment an entry in the typing environment Δ can be removed. We write $a \leftrightarrow \mathbf{r}$ (resp. $a \leftrightarrow \kappa$) to denote that name a is bound in the process, or the process is in a context were the name a is bound with $(\nu a : \mathbf{r})$ (resp. $(\nu a : \kappa)$).

Lemma 3.6.6 (Weakening)**.** *Let* $\Delta \vdash_\rho P$.

1. *If* $a \notin \mathsf{fn}(P) \cup \rho$, *then*

   (a) $a \leftrightarrow \mathbf{r}$ *and* $\mathbf{r} \notin \mathsf{sym}(P)$ *and* $\Delta' = \Delta\{a/\mathbf{r}\}$ *implies* $\Delta', a : \{a\}(T) \vdash_\rho P$;

   (b) $a \leftrightarrow \kappa$ *implies* $\Delta, a : \kappa(T) \vdash_\rho P$.

2. $\Delta \vdash_{\rho \uplus \rho'} P$.

Proof. The proof is by induction on the depth of the derivation $\Delta \vdash_\rho P$. *1*. We detail only two cases, when the last applied rule is (T-OUT) or (T-REP-IN).

- *Case* (T-OUT): Let $\Delta \vdash_\rho b!c.P$ be derived from $\Delta \vdash_\rho P$, where $\Delta(b) = \gamma(\gamma'(T'))$, $\Delta(c) = \gamma''(T')$, $\gamma'' \subseteq \gamma'$, and if $b \notin \rho$ then $\gamma \subseteq \rho$. Then, we distinguish two cases.

  (a): If $a \leftrightarrow \mathbf{r}$ and $\mathbf{r} \notin \mathsf{sym}(P)$ and $\Delta' = \Delta\{a/\mathbf{r}\}$, by induction hypothesis $\Delta', a : \{a\}(T) \vdash_\rho P$. By $\Delta'(b) = (\gamma(\gamma'(T')))\{a/\mathbf{r}\}$ and $\Delta'(c) = (\gamma''(T'))\{a/\mathbf{r}\}$, we conclude $\gamma''\{a/\mathbf{r}\} \subseteq \gamma'\{a/\mathbf{r}\}$. Since $b \notin \rho$ implies $\gamma \subseteq \rho$, and ρ is a multiset of names, then $\mathbf{r} \in \gamma$ implies $b \in \rho$. If $\mathbf{r} \notin \gamma$ then $\gamma\{a/\mathbf{r}\} = \gamma$. Thus, by (T-OUT) we may conclude $\Delta', a : \{a\}(T) \vdash_\rho b!c.P$.

  (b): If $a \leftrightarrow \kappa$ then by induction hypothesis $\Delta, a : \kappa(T) \vdash_\rho P$. Since Δ is not changed, the result follows directly by (T-OUT).

- *Case* (T-REP-IN): Let $\Delta \vdash_\rho !(b)b?xP$ be derived from $\Delta, x : T' \vdash_{\{b\}} P$, with $\Delta(b) = \gamma(T')$, where without loss of generality we can assume that $x \neq a$. Again, we have two cases.

  (a): If $a \leftrightarrow \mathbf{r}$ and $\mathbf{r} \notin \mathsf{sym}(P)$ and $\Delta' = \Delta\{a/\mathbf{r}\}$, by induction hypothesis $\Delta', x : T'\{a/\mathbf{r}\}, a : \{a\}(T) \vdash_{\{b\}} P$. Since $\Delta'(b) = (\gamma(T'))\{a/\mathbf{r}\}$ by (T-REP-IN) we may conclude $\Delta', a : \{a\}(T) \vdash_\rho !(b)b?xP$.

  (b): If $a \leftrightarrow \kappa$ then by induction hypothesis $\Delta, x : T', a : \kappa(T) \vdash_{\{b\}} P$. Again, by (T-REP-IN) the proof for this case follows.

2. We detail only the case when the last applied rule is (T-IN). Let $\Delta \vdash_\rho a?x.P$ be derived from $\Delta, x : T \vdash_\rho P$, where $\Delta(a) = \omega(T)$, $x \notin \rho, names(\Delta)$ and $a \notin \rho$ implies $\omega \subseteq \rho$. By induction hypothesis $\Delta, x : T \vdash_{\rho \uplus \rho'} P$. Without loss of generality we can assume that x is new to ρ', i.e. $x \notin \rho'$. Observing that $a \notin \rho \uplus \rho'$ implies $\omega \subseteq \rho \uplus \rho'$, by (T-IN) we conclude $\Delta \vdash_{\rho \uplus \rho'} a?x.P$. □

Lemma 3.6.7 (Strengthening)**.** *Let* $a \notin \mathsf{fn}(P) \cup \rho$.

(*a*) *If* $\Delta, a : \{a\}(T) \vdash_\rho P$ *and* $a \leftrightarrow \mathbf{r}$ *and* $\mathbf{r} \notin \mathsf{sym}(P)$ *then* $\Delta' \vdash_\rho P$, *where* $\Delta' = \Delta\{\mathbf{r}/a\}$.

(*b*) *If* $\Delta, a : \kappa(T) \vdash_\rho P$ *then* $\Delta \vdash_\rho P$.

Proof. The proof is by induction on the depth of the derivation $\Delta \vdash_\rho P$. We comment only the case when the last applied rule is (T-IN). We distinguish two cases.

(*a*) Let $a \leftrightarrow \mathbf{r}$ and $\mathbf{r} \notin \mathsf{sym}(P)$ and $\Delta' = \Delta\{\mathbf{r}/a\}$, and $\Delta, a : \{a\}(T) \vdash_\rho b?x.P$. Since $a \notin \mathsf{fn}(b?xP)$, and without loss of generality, we can conclude $a \neq b$ and $a \neq x$. By Lemma 3.6.5 we have that $\Delta, a : \{a\}(T), x : T' \vdash_\rho P$ and $\Delta(b) = \gamma(T')$, and $b \notin \rho$ implies $\gamma \subseteq \rho$. By induction hypothesis we have $\Delta', x : T'\{\mathbf{r}/a\} \vdash_\rho P$. Since $a \notin \rho$ then $a \in \gamma$ implies $b \in \rho$. If $a \notin \gamma$ then $\gamma\{\mathbf{r}/a\} = \gamma$. Thus, by (T-IN) we derive $\Delta' \vdash_\rho b?x.P$.

(*b*) If $a \leftrightarrow \kappa$ and $\Delta, a : \{a\}(T) \vdash_\rho b?x.P$. Using the same arguments as in the first part of the proof, we can again assume $a \neq b$ and $a \neq x$. By Lemma 3.6.5 we again have that $\Delta, a : \kappa(T), x : T' \vdash_\rho P$ and $\Delta(b) = \gamma(T')$, and $b \notin \rho$ implies $\gamma \subseteq \rho$. By induction hypothesis $\Delta, x : T' \vdash_\rho P$. Thus, by (T-IN) we derive $\Delta \vdash_\rho b?x.P$.

□

Our next result shows that typing is preserved under the structural congruence.

Lemma 3.6.8 (Subject congruence)**.** *If* $\Delta \vdash_\rho P$ *and* $P \equiv Q$ *then* $\Delta \vdash_\rho Q$.

Proof. The proof is by induction on the depth of the derivation $P \equiv Q$. We comment only three cases, when the last applied rule is (SC-PAR-INACT), (SC-RES-EXTR) or (SC-REP).

- *Case* $P \mid 0 \equiv P$. Assume $\Delta \vdash_\rho P \mid 0$. By Lemma 3.6.5 we have $\Delta \vdash_{\rho_1} P$ and $\Delta \vdash_{\rho_2} 0$, where $\rho_1 \uplus \rho_2 = \rho$. By Lemma 3.6.6 we get $\Delta \vdash_\rho P$.

  Now assume $\Delta \vdash_\rho P$. By (T-STOP) we obtain $\Delta \vdash_\emptyset 0$ and by (T-PAR) we conclude $\Delta \vdash_\rho P \mid 0$.

- *Case* $P \mid (\nu a : \mathbf{r})Q \equiv (\nu a : \mathbf{r})(P \mid Q)$ or $P \mid (\nu a : \kappa)Q \equiv (\nu a : \kappa)(P \mid Q)$, if $a \notin \mathsf{fn}(P)$.

  To show implication from right to the left we have two cases.

  (*a*) If $a \leftrightarrow \mathbf{r}$ then from $\Delta \vdash_\rho P \mid (\nu a : \mathbf{r})Q$ by Lemma 3.6.5 we have that

  $$\Delta \vdash_{\rho_1} P \text{ and } \Delta \vdash_{\rho_2} (\nu a : \mathbf{r})Q$$

  where $\rho_1 \uplus \rho_2 = \rho$, and $\mathsf{sym}(P) \cap \mathsf{sym}((\nu a : \mathbf{r})Q) = \emptyset$. Applying Lemma 3.6.5 again we obtain

  $$\Delta', a : \{a\}(T) \vdash_{\rho_2} Q$$

  where $\Delta' = \Delta\{a/\mathbf{r}\}$ and $a \notin \rho_2$ and $\mathbf{r} \notin \mathsf{sym}(Q)$. Since $\mathsf{sym}(P) \cap \mathsf{sym}((\nu a : \mathbf{r})Q) = \emptyset$ we conclude $\mathbf{r} \notin \mathsf{sym}(P)$, and from $a \notin \mathsf{fn}(P)$ and $a \in \mathsf{bn}((\nu a : \mathbf{r})Q)$ without loss of generality can conclude $a \notin \rho_1$. Then, by Lemma 3.6.6 we have

  $$\Delta', a : \{a\}(T) \vdash_{\rho_1} P$$

  where $\Delta' = \Delta\{a/\mathbf{r}\}$. By (T-PAR) we obtain $\Delta', a : \{a\}(T) \vdash_\rho P \mid Q$ and by (T-NEW) we conclude $\Delta \vdash_\rho (\nu a : \mathbf{r})(P \mid Q)$.

(*b*) If $a \leftrightarrow \kappa$ then from $\Delta \vdash_\rho P \mid (\nu a : \kappa)Q$ by Lemma 3.6.5 we have

$$\Delta \vdash_{\rho_1} P \text{ and } \Delta \vdash_{\rho_2} (\nu a : \kappa)Q$$

where $\rho_1 \uplus \rho_2 = \rho$. By Lemma 3.6.5 again

$$\Delta, a : \kappa(T) \vdash_{\rho_2} Q$$

where $a \notin \rho_2$. By Lemma 3.6.6 we have $\Delta, a : \kappa(T) \vdash_{\rho_1} P$. By (T-PAR) we derive $\Delta, a : \kappa(T) \vdash_\rho P \mid Q$ and by (T-NEW-REP) we conclude $\Delta \vdash_\rho (\nu a : \kappa)(P \mid Q)$.

To show implication from left to the right we again have two cases.

(*a*) If $a \leftrightarrow \mathbf{r}$ then from $\Delta \vdash_\rho (\nu a : \mathbf{r})(P \mid Q)$ by Lemma 3.6.5 we obtain $\Delta', a : a(T) \vdash_\rho P \mid Q$, where $\Delta' = \Delta\{a/\mathbf{r}\}$ and $\mathbf{r} \notin \mathsf{sym}(P \mid Q)$ and $a \notin \rho$. By Lemma 3.6.5

$$\Delta', a : \{a\}(T) \vdash_{\rho_1} P \text{ and } \Delta', a : \{a\}(T) \vdash_{\rho_2} Q$$

where $\rho_1 \uplus \rho_2 = \rho$ and $\mathsf{sym}(P) \cap \mathsf{sym}(Q) = \emptyset$. Since $a \notin \mathsf{fn}(P) \cup \rho_1$ and $\mathbf{r} \notin \mathsf{sym}(P)$ by Lemma 3.6.7 we have $\Delta \vdash_{\rho_1} P$. Using $\mathbf{r} \notin \mathsf{sym}(Q)$ and $a \notin \rho_2$ by (T-NEW) we derive $\Delta \vdash_{\rho_2} (\nu a : \mathbf{r})Q$, and by (T-PAR) follows $\Delta \vdash_\rho P \mid (\nu a : \mathbf{r})Q$.

(*b*) If $a \leftrightarrow \kappa$ then from $\Delta \vdash_\rho (\nu a : \kappa)(P \mid Q)$ by Lemma 3.6.5 we have $\Delta, a : \kappa(T) \vdash_\rho P \mid Q$ where $a \notin \rho$. By Lemma 3.6.5

$$\Delta, a : \{a\}(T) \vdash_{\rho_1} P \text{ and } \Delta, a : \{a\}(T) \vdash_{\rho_2} Q$$

where $\rho_1 \uplus \rho_2 = \rho$. Since $a \notin \mathsf{fn}(P) \cup \rho_1$ by Lemma 3.6.7 we obtain $\Delta \vdash_{\rho_1} P$. Using (T-NEW-REP) we have $\Delta \vdash_{\rho_2} (\nu a : \kappa)Q$, and by (T-PAR) $\Delta \vdash_\rho P \mid (\nu a : \kappa)Q$.

- *Case* $!(a)a?x.P \equiv !(a)a?x.P \mid (a)a?x.P$. We show only one implication. Assume $\Delta \vdash_\rho !(a)a?x.P \mid (a)a?x.P$. By Lemma 3.6.5 we have

$$\Delta \vdash_{\rho_1} !(a)a?x.P \text{ and } \Delta \vdash_{\rho_2} (a)a?x.P$$

where $\rho_1 \uplus \rho_2 = \rho$. By the same Lemma we derive $\mathsf{sym}(P) = \emptyset$ and $\Delta, x : T \vdash_{\{a\}} P$, where $\Delta(a) = \gamma(T)$ and $x \notin \rho_1, \mathit{names}(\Delta)$. By (T-REP-IN) we conclude $\Delta \vdash_\rho !(a)a?x.P$.

□

In order to prove that reduction preserves typing we also need another auxiliary result that connects a typing and name substitutions of a process.

Lemma 3.6.9 (Substitution)**.** *Let* $\Delta, x : \gamma(T) \vdash_\rho P$ *and* $x \notin \mathit{names}(\Delta)$.

1. *If* $\Delta(a) = \{a\}(T)$ *and* $a \in \gamma$ *then* $\Delta \vdash_{\rho\{a/x\}} P\{a/x\}$.
2. *If* $\Delta(a) = \kappa(T)$ *and* $\kappa = \gamma$ *then* $\Delta \vdash_{\rho\{a/x\}} P\{a/x\}$.

Proof. The proof is by induction on the depth of the derivation $\Delta \vdash_\rho P$. We detail two cases:

- *Case* $\Delta, x : \gamma(\gamma'(T)) \vdash_\rho x!b.P$. By Lemma 3.6.5 we have $\Delta, x : \gamma(\gamma'(T)) \vdash_\rho P$, where $\Delta(b) = \gamma''(T)$ and $\gamma'' \subseteq \gamma'$, and $x \notin \rho$ implies $\gamma \subseteq \rho$. By induction hypothesis $\Delta \vdash_{\rho\{a/x\}} P\{a/x\}$. We now distinguish two cases.

  1. $\Delta(a) = \{a\}(\gamma'(T))$ and $a \in \gamma$. Then, $x \in \rho$ implies $a \in \rho\{a/x\}$, and if $x \notin \rho$ by $\gamma \subseteq \rho$ we again derive $a \in \gamma \subseteq \rho = \rho\{a/x\}$.
  2. $\Delta(a) = \kappa(\gamma'(T))$ and $\kappa = \gamma$. Since ρ is a multiset of names, we conclude $x \in \rho$ must hold. Hence, $a \in \rho\{a/x\}$.

  In both cases by (T-OUT) we conclude $\Delta \vdash_{\rho\{a/x\}} (x!b.P)\{a/x\}$.

- *Case* $\Delta, x : \gamma(T) \vdash_\rho b!x.P$. By Lemma 3.6.5 we have $\Delta, x : \gamma(T) \vdash_\rho P$, where $\Delta(b) = \gamma'(\gamma''(T))$, and $\gamma \subseteq \gamma''$, and $b \notin \rho$ implies $\gamma \subseteq \rho$. By induction hypothesis $\Delta \vdash_{\rho\{a/x\}} P\{a/x\}$. Again, we distinguish two cases.

1. $\Delta(a) = \{a\}(\gamma'(T))$ and $a \in \gamma$. Then, $a \in \gamma \subseteq \gamma''$.
2. $\Delta(a) = \kappa(\gamma'(T))$ and $\kappa = \gamma$. Then, $\kappa = \gamma \subseteq \gamma''$.

Thus, in both cases by (T-OUT) we derive $\Delta \vdash_{\rho\{a/x\}} (b!x.P)\{a/x\}$.

□

We may notice that, even though subtyping is not present, the last result uses an inclusion principle ($a \in \gamma$) that already hints on substitutability. Our next results shows that if two processes that both need authorization for the same name are placed in a two-hole context and then typed with $\rho = \emptyset$, then the two authorizations are present in the context and the corresponding *drift* operator is consequently defined. The proof of next lemma follows in expected lines by induction on the structure of the context and here is omitted.

Lemma 3.6.10 (Authorization Safety)**.** *If* $\Delta \vdash_\emptyset \mathcal{C}[P_1, P_2]$ *and* $\Delta \vdash_{\rho_1} P_1$ *and* $\Delta \vdash_{\rho_2} P_2$ *and* $a \in \rho_1 \cap \rho_2$ *then* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *is defined.*

We can now show that a well-typed process is not an error. For the rest of the section, we use $(\nu\tilde{c} : \tilde{\Omega})$ to abbreviate $(\nu c_1 : \Omega_1) \ldots (\nu c_n : \Omega_n)$, where Ω ranges over symbols from $\mathcal{S}$ and ν.

Lemma 3.6.11 (Interaction Safety)**.** *Let* P *be well-typed with* $\Delta \vdash_\emptyset P$.

1. *If* $P \equiv (\nu\tilde{c} : \tilde{\Omega})\mathcal{C}[a!b.P_1, a?x.P_2]$ *then* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *is defined and for* $\mathcal{C}'[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ *and* $Q \equiv (\nu\tilde{c} : \tilde{\Omega})\mathcal{C}'[(a)P_1, (a)P_2\{b/x\}]$ *we have that* $\Delta \vdash_\emptyset Q$.

2. *If* $P \equiv (\nu\tilde{c} : \tilde{\Omega})\mathcal{C}[a\langle b\rangle.P_1, a(b).P_2]$ *then* $drift(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *is defined and for* $\mathcal{C}'[\cdot_1, \cdot_2] = drift(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ *and* $Q \equiv (\nu\tilde{c} : \tilde{\Omega})\mathcal{C}'[(a)P_1, (a)(b)P_2]$ *we have that* $\Delta \vdash_\emptyset Q$.

Proof. The proof is by induction on the structure of the context $\mathcal{C}[\cdot_1, \cdot_2]$. We detail only the first statement. If $\Delta \vdash_\emptyset P$ by Lemma 3.6.8 we have

$$\Delta \vdash_\emptyset (\nu\tilde{c} : \tilde{\Omega})\mathcal{C}[a!b.P_1, a?x.P_2]$$

and by consecutive application of Lemma 3.6.5. 1 and 2, we derive

$$\Delta' \vdash_{\emptyset} \mathcal{C}[a!b.P_1, a?x.P_2]$$

where $\Delta' = \Delta, \Delta''$, and for each $c \in dom(\Delta'')$ we have that $\Delta''(c) = c(T)$ or $\Delta''(c) = \kappa(T)$. By consecutive application of Lemma 3.6.5. 3 and 9

$$\Delta' \vdash_{\rho_1} a!b.P_1 \quad \text{and} \quad \Delta' \vdash_{\rho_2} a?x.P_2$$

for some multisets of names ρ_1, ρ_2. By the same Lemma again

$$\Delta' \vdash_{\rho_1} P_1 \quad \text{and} \quad \Delta', x : \gamma(T) \vdash_{\rho_2} P_2$$

where $\Delta'(a) = \{a\}(\gamma(T))$ or $\Delta'(a) = \kappa(\gamma(T))$, and, thus, we may conclude $a \in \rho_1$, $a \in \rho_2$. Furthermore, we may observe that $\Delta'(b) = \{b\}(T)$ or $\Delta'(b) = \kappa(T)$, and $b \in \gamma$ and $x \notin \rho_2 \cup names(\Delta')$. Hence, we have $\rho_2\{b/x\} = \rho_2$, and by Lemma 3.6.9 we obtain $\Delta' \vdash_{\rho_2} P_2\{b/x\}$. By (T-AUTH) we derive

$$\Delta' \vdash_{\rho_1'} (a)P_1 \quad \text{and} \quad \Delta' \vdash_{\rho_2'} (a)P_2\{b/x\},$$

where $\rho_1 = \rho_1' \uplus \{a\}$ and $\rho_2 = \rho_2' \uplus \{a\}$.

Since $\Delta' \vdash_{\emptyset} \mathcal{C}[a!b.P_1, a?x.P_2]$, and $a \in \rho_1 \cap \rho_2$, by Lemma 3.6.10 we conclude $drift(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ is defined. Thus, by Proposition 3.4.11 we distinguish four cases for the structure of the context

$$\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}''[\mathcal{C}_1[\cdot_1] \mid \mathcal{C}_2[\cdot_2]]$$

We comment only the case when $\mathcal{C}_1[\cdot] = \mathcal{C}_1'[(a)\mathcal{C}_1''[\cdot]]$ and $\mathcal{C}_2[\cdot] = \mathcal{C}_2'[(a)\mathcal{C}_2''[\cdot]]$, where $drift(\mathcal{C}_1''[\cdot]; a)$ and $drift(\mathcal{C}_2''[\cdot]; a)$ are undefined. The latter implies that in contexts $\mathcal{C}_1''[\cdot]$ and $\mathcal{C}_2''[\cdot]$ the holes are not in the scope of authorizations (a). By consecutive application of (T-PAR) and (T-AUTH) we derive

$$\Delta' \vdash_{\rho_1''} (a)\mathcal{C}_1''[a!b.P_1] \text{ and } \Delta' \vdash_{\rho_2''} (a)\mathcal{C}_2''[a?x.P_2]$$

and also

$$\Delta' \vdash_{\rho_1''} \mathcal{C}_1''[(a)P_1] \text{ and } \Delta' \vdash_{\rho_2''} \mathcal{C}_2''[(a)P_2\{b/x\}]$$

for some ρ_1'' and ρ_2''. Since $\mathcal{C}'[\cdot_1, \cdot_2] = \mathcal{C}''[\mathcal{C}_1'[\mathcal{C}_1''[\cdot_1]] \mid \mathcal{C}_2'[\mathcal{C}_2''[\cdot_2]]]$, by consecutive application of (T-PAR) and (T-AUTH) we have $\Delta' \vdash_\emptyset \mathcal{C}'[(a)P_1, (a)P_2\{b/x\}]$. Then, by consecutive application of (T-NEW) and (T-NEW-REP) we have

$$\Delta \vdash_\emptyset (\nu \tilde{c} : \tilde{\Omega})\mathcal{C}'[(a)P_1, (a)P_2\{b/x\}]$$

By Lemma 3.6.8 we conclude $\Delta \vdash_\emptyset Q$. □

Since errors involve redexes, the proof of Lemma 3.6.11 is intertwined with the proof of the error absence property. As a direct consequence of Lemma 3.6.11 we get the soundness of our typing analysis.

Proposition 3.6.12 (Type Soundness)**.** *If P is well-typed then P is not an error.*

We may also show that reduction also preserves typing.

Theorem 3.6.13 (Subject Reduction)**.** *If P is well-typed, $\Delta \vdash_\emptyset P$ and $P \to Q$ then $\Delta \vdash_\emptyset Q$.*

Proof. The proof follows by induction on the derivation of $P \to Q$. We have two base cases by rules (R-COMM) or (R-AUTH), both of which follow directly by Lemma 3.6.11. For the induction steps we have two cases.

- If the last applied rule is (R-NEWC) we again distinguish two cases.

  (*a*) $(\nu a : \mathbf{r})P' \to (\nu a : \mathbf{r})Q'$ is derived from $P' \to Q'$. Let $\Delta \vdash_\emptyset (\nu a : \mathbf{r})P'$. By Proposition 3.6.2 we conclude $(\nu a : \mathbf{r})Q'$ is well-formed, thus $\mathbf{r} \notin \mathsf{sym}(Q')$. By Lemma 3.6.5 we have that $\Delta', a : \{a\}(T) \vdash_\emptyset P'$, where $\mathbf{r} \notin \mathsf{sym}(P')$ and $\Delta' = \Delta\{a/\mathbf{r}\}$. By induction hypothesis $\Delta', a : \{a\}(T) \vdash_\emptyset Q'$, and by (T-NEW) we derive $\Delta \vdash_\emptyset (\nu a : \mathbf{r})Q'$.

  (*b*) $(\nu a : \kappa)P' \to (\nu a : \kappa)Q'$ is derived from $P' \to Q'$.Follows similar reasoning, by application of rule (T-NEW-REP).

- If the last applied rule is (R-STRUC) then $P \to Q$ is derived from $P' \to Q'$, where $P \equiv P'$ and $Q \equiv Q'$. Let $\Delta \vdash_\emptyset P$. By Lemma 3.6.8 we have $\Delta \vdash_\emptyset P'$. By induction hypothesis $\Delta \vdash_\emptyset Q'$ and again by Lemma 3.6.8 we conclude $\Delta \vdash_\emptyset Q$.

□

Combining Proposition 3.6.12 and Theorem 3.6.13 we may observe that a well-typed process is not an error and also that it never reduces to an error, which is the main result of this section.

Corollary 3.6.14 (Type Safety)**.** *If P is well-typed and $P \to^* Q$ then Q is not an error.*

3.6.5 Illustrating typing rules by examples

In this section, we further explain the principles behind our typing discipline by extending the example given in Section 3.6.2. Let us consider that the first process shown in Section 3.6.2 is composed with another process willing to send a name along *alice*, specifically

$$(alice)\,alice!exam.0 \mid (exam)(minitest)(alice)\,alice?x.x!value.0 \quad (3.5)$$

Considering assignments $alice : \{alice\}(\{exam, minitest\}(\{value\}(\emptyset)))$ and $exam : \{exam\}(\{value\}(\emptyset))$, by rule (T-OUT) we may observe that sending name *exam* on *alice* is safe, for the only replacement of *exam* given in its type (which is the name itself) is also specified as safe to be communicated in *alice* (since it is included in $\{exam, minitest\}$). To exemplify the symbolic representation of names in types, let us consider process (3.5) is placed in the context that restricts *exam*:

$$(\nu\, exam : \mathbf{r})((alice)\,alice!exam.0 \mid (exam)(minitest)(alice)\,alice?x.x!value.0) \quad (3.6)$$

The assignment given above for *alice* changes to $\{alice\}(\{\mathbf{r}, minitest\}(\{value\}(\emptyset)))$, by rule (T-NEW). The introduced symbol represents that a restricted

name can be communicated in *alice*, and that the restricted process cannot be composed with others that rely on contextual authorizations for names exchanged in *alice*. Nevertheless, the process in (3.6) can be composed with processes $(alice)alice?x.(x)x!value$ and $(alice)alice?x.alice(x)x!value$, in which the name received in *alice* is authorized directly. Let us now consider replicated process

$$!(license)license?x.(\nu exam : \mathbf{r})((x)x!exam.0 \mid (x)(exam)x?y.y!value.0) \tag{3.7}$$

that can serve as a model of a server repeatedly available to receive a name and then, on the received name to receive (in the left thread) and/or to send a fresh name (in the right thread). This process is rejected by our typing analysis since a symbol ($\mathbf{r}$) appears in the body of a replicated input (cf. (T-REP-IN)). In fact, this process can reduce to an error. For instance, the process in (3.7) can receive *alice* twice, activating two copies of the replicated process

$$\begin{aligned}&(\nu exam_1 : \mathbf{r})((alice)alice!exam_1.0 \mid (exam_1)(alice)alice?y.y!value.0)\\ &\mid (\nu exam_2 : \mathbf{r})((alice)alice!exam_2.0 \mid (exam_2)(alice)alice?y.y!value.0)\end{aligned} \tag{3.8}$$

The two "copies" of the restricted name *exam* are actually different names. Since both names can be sent on *alice* the error can be reached if the received name does not match the contextual authorization (e.g., $(exam_2)(alice)exam_1!value.0$).

Our typing analysis ensures that names created inside replicated input are distinguished with special symbol κ. The symbol represents that the associated name is never subject to contextual authorizations, not even within the restriction scope (cf. (T-NEW-REP)). Now, if we consider κ instead of $\mathbf{r}$ annotation in the process given in (3.7) we again obtain a process that cannot be typed, but now the reason can be found explicitly in the replicated process: we are trying to rely on the contextual authorizations for name marked with κ, which is not allowed. Hence, for names created in the body of replicated inputs we cannot rely on contextual authorizations in any part of the process.

Still, process

$$!(license)license?x.(\nu exam : \kappa)(alice)alice!exam.0 \qquad (3.9)$$

can be typed, e.g., using assumption $alice : \{alice\}(\kappa(\{value\}(\emptyset))$. Thus, the process can be composed with processes that do not rely on contextual authorizations for names communicated in *alice*. We may observe that the carried type $\{value\}(\emptyset)$ does not change regardless of type κ, and that channels communicated in *alice* can in turn only be used to communicate *value*.

In our model we can directly represent servers that allow for an infinite authorization generation. For instance, process

$$!(public)public?x.((x)public\langle x\rangle.0 \mid (public)public!x.0)$$

can delegate an unbounded number of authorizations for a received name, through delegation on channel *public*. Composing the above process in parallel with $(public)public!comm.0$ yields a process that can generate infinite number of copies of $(public)(comm)public\langle comm\rangle.0$. For this, anyone authorized to use *public* can also be authorized to use *comm*. Using assumption $public : \{public\}(\gamma(T))$, for some γ, we can type the authorization generator process, and assuming $comm \in \gamma$ we may also type the composition using $comm : \{comm\}(T)$.

3.6.6 Type-checking

The type analysis presented in the last section can single out processes that are authorization safe, i.e., are not errors and do not evolve to errors. However, we may notice that it raises some questions on the applicability of the induced type-checking procedure. This section presents a refined type system that deals with some of the issues and in Section 3.6.6.1 we show the two typing systems are equivalent in what concerns typability of processes.

The first problem we can observe is the inference of the type of a bound name in rules (T-NEW) and (T-NEW-REP). We solve this issue

in the usual way by adding the type information to name restrictions. Therefore, we now write

$$(\nu a : \mathbf{r}(T)) \text{ and } (\nu a : \kappa(T)),$$

instead of $(\nu a : \mathbf{r})$ and $(\nu a : \kappa)$, respectively. The second problem is efficiency of guessing how to split the multiset ρ to the branches in rule (T-PAR). We tackle this problem by refining the type discilpine following the idea of [110].

The idea is instead of dividing the multiset ρ in two randomly, we first pass the whole ρ to the left branch, after which the part of ρ that is unused by the left branch is passed to the right branch. Our first attempt to implement this idea is to extend the typing judgment $\Delta \vdash_\rho P$ to $\Delta \vdash_\rho P; \rho'$, where ρ' represents the multiset of names of authorizations that are not used by P, and that, hence, serves as the "output" of the algorithm. Following this intuition, let us try to type process $(a)((b)0 \mid P)$. We may observe that both authorizations (a) and (b) are scoping the left branch and that both are unused by the process there (process 0). Hence, the intuition says to consider names a and b as part of the output multiset when typing 0 and pass them on to the right branch (process P) in the verification. This leads us to a situation in which authorization (b) can be used for the verification of the process P without even scoping over the process. Therefore, we further refine the typing judgment by splitting the multiset ρ in two, obtaining $\Delta; \rho_1 : \rho_2 \vdash P; \rho'$, where ρ_1 represents a multiset of names of authorizations that can be considered to be passed as unused, and names of authorizations in ρ_2 cannot be passed (ρ' does not change). Following this intuition and considering again the same process $(a)((b)0 \mid P)$ we would have that when verifying process 0, name a is in ρ_1, while b is in ρ_2 (not to be passed), resulting in $\rho' = \{a\}$.

The third problem of the typing rules given in the last section is also connected to rule (T-PAR): we need to deduce if the two sets of symbols mentioned in the two branches are disjoint. We use similar

reasoning as before, and once again refine the typing judgment to

$$\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi',$$

where ξ is a set that incrementally collects all used symbols from $\mathcal{S}$ mentioned in the process, and ξ' is the output of the algorithm, which in this case is the set of used symbols. The idea is to use set ξ to collect used symbols, that are then passed in the output ξ'. Hence, when parallel composition is considered, starting set if symbols ξ should be first passed to the left branch. Then, all symbols discovered in the verification of the left branch are added to ξ and passed to the right branch in the output ξ'. We remark that there are no changes in the interpretation of the typing environment Δ with respect to typing discipline given in Section 3.6.

We noted that in the typing judgment multiset ρ_1 represents authorizations that can be passed to a process placed in parallel and typed afterwards, while ρ_2 represents authorizations that are not for passing. Consider now process $a!b.a!c.0$ is typed with $\rho_1 = \{a, a\}$ and $\rho_2 = \emptyset$. When typing the first output one authorization in ρ_1 should be considered as used, and hence, not to be considered as to be passed. However, the authorization should be considered as still available for the second output of the process (representing the confinement of the operational semantics). To this end, while typing the first output we need only to transfer one name from ρ_1 to ρ_2. Then, the second output is typed with $\rho_1 = a$ and $\rho_2 = a$, in which case no transfer is needed as one a is already in ρ_2 (representing the fairness). We formalize this idea by introducing an auxiliary operation, called *move*, dedicated for transferring names between multisets.

Similar to operator *drift* from Section 3.4, operator *move* also has a twofold meaning. On one hand, $move(\rho_1 : \rho_2, b)$ represents the transfer of name b from multiset ρ_1 to multiset ρ_2, provided the name is not already contained in ρ_2 in the first place. If the name is contained in ρ_2 the operation is idempotent. On the other hand, if the name is not present in both multisets, i.e., if $b \notin \rho_1 \uplus \rho_2$, the operator is undefined, signaling insufficient authorizations.

Definition 3.6.15 (Operator *move*)**.** *Operator move takes as arguments a pair of multisets $\rho_1 : \rho_2$ and a name b and is defined as follows*

$$move(\rho_1 : \rho_2, b) = \rho_1 \backslash (\{b\} \backslash \rho_2) : \rho_2 \uplus (\{b\} \backslash \rho_2), \text{ if } b \in \rho_1 \uplus \rho_2 \text{ (undefined otherwise).}$$

Operator move is directly generalized for the case of transferring a set of names γ, by transferring it name by name, namely if $\gamma = b, \gamma'$ then for arbitrary multisets ρ_1 and ρ_2, we write $move(\rho_1 : \rho_2, \gamma) = move(move(\rho_1 : \rho_2, b), \gamma')$.

We may now present the type-checking rules. Table 3.8 presents the rules for typing non-prefixing language constructs, explained next reading from conclusion to premises.

- In rule (A-STOP) multiset of names ρ_1 and the set of symbols ξ are passed as the output, while ρ_2 is discarded.
- In (A-AUTH) the name of authorization that scopes over process P, is added to multiset ρ_2 since after type-checking the process P the authorization should not be considered as available outside of this scope.
- In (A-PAR) multisets of names of authorizations ρ_1 and ρ_2 given for the parallel composition are passed to the left branch (process P_1) as authorizations that can be passed (to the right branch). The second multiset of names, which refers to authorizations exclusively scoping the left branch, is empty. In the same way, the starting set of symbols ξ is passed to process P_1. The part of $\rho_1 \uplus \rho_2$ that is unused in the verification of process P_1 is given in ρ_3 and is thus passed to the verification of the right branch (process P_2). Also, the set of symbols ξ', which is ξ enlarged with symbols from P_1, is passed to the verification of P_2. The output of the checking the parallel composition is: a multiset $\rho_1 \cap \rho_4$, which is a multiset of names of authorizations that are unused by both P_1 and P_2 obtained in the verification of the right branch (ρ_4),

and originally considered to be passed as unused for the parallel composition process (ρ_1); the set of symbols ξ'', obtained as the result of the verification of the right branch (ξ with symbols from P_1 and P_2).

- The only novelty in rule (A-NEW) is that the symbol associated with restricted name is added to the set of used symbols, and rule (A-NEW-REP) follows exactly the same lines as (T-NEW-REP) (except using the explicit types given in the syntax).

Table 3.9 introduces typing prefixes.

- In (A-OUT-1) the output process is typed if names of authorizations required for the output (a or γ), are contained in the multiset of names ρ_2, hence authorizations that are not considered to be passed. In this case, the continuation process P is typed under the unchanged conditions.

- In (A-OUT-2) the process is typed if the names of authorizations required for the output are not contained in ρ_2 (hence (A-OUT-1) cannot be applied), but are contained in $\rho_1 \uplus \rho_2$. This is the case when name a or (a subset of) names γ included in $\rho_1 \setminus \rho_2$, should not be considered as to be passed anymore (for they are used by the prefix) and thus must be transferred from ρ_1 to ρ_2 by operator *move*. This means that for $\beta = a$ the name is transferred from ρ_1 to ρ_2, and for $\beta = \gamma$ only names from γ that are in ρ_1 but are not in ρ_2 are transferred from ρ_1 to ρ_2.

- Rules (A-IN-1) and (A-IN-2) follow the same lines as rules (A-OUT-1) and (A-OUT-2).

- In rule (A-REP-IN) only one authorization for a is specified as available for the process P. The process P must contain no symbols, as the input and the output set of symbols are empty. In conclusion, the input multiset of names ρ_1 and the set of symbols ξ are both passed to the output directly.

- In both rules (A-DELEG-1) and (A-DELEG-2) the name of the delegated authorization must be present in $\rho_1 \uplus \rho_2$ by application of operator *move*. The operator moves b from ρ_1 to ρ_2 if not already present in the former. As the authorization is to be delegated away by the prefix, the name b is then taken out from the last multiset. After that, name(s) of authorization(s) required for the action on a is(are) manipulated following the same reasoning as in (A-OUT-1) and (A-OUT-2).

- Rules (A-RECEP-1) and (A-RECEP-2) follow the same lines as (A-DELEG-1) and (A-DELEG-2), except that the name b is not taken out but added to the multiset of names that are directly scoping the process, and hence, not considered to be passed (as in (A-AUTH)).

We remark that rule (A-OUT-2) is non-deterministic in the case when both a and also (a subset) of γ are contained in $\rho_1 \setminus \rho_2$, as both options of transferring via *move* are left open. Furthermore, the non-determinism appears only if also the name a is not considered as final in its type, i.e., if $\gamma \neq a$ and $\gamma \neq \kappa$. Notice that a similar condition ($a \notin \rho \Rightarrow \gamma \in \rho$) offering a choice is also present in the original system in rule (T-OUT), but here we actually must commit to one of them so as to "mark" which names are used. For the purpose of the induced verification procedure, we need to check both possibilities in every application of the rule, up to the point the verification is successful or all options have been explored.

Nevertheless, the type-checking procedure following rules given in Table 3.8 and Table 3.9 is more efficient than the one following the original rules shown in Table 3.7. We may observe that rule (T-PAR) requires the exploration of all possible decompositions of ρ, for which a direct implementation is exponential on the size of ρ. On the other hand, the rules (A-OUT-2), (A-IN-2), (A-DELEG-2) and (A-RECEP-2) also involve some exploration (because of premise $\beta = a \vee \beta = \gamma$). In these rules we have two options for β when *move* is defined for both a

and γ and we need to explore both of options to determine typability. In case *move* is defined only for one option no further exploration is necessary. Therefore, if a program is not typable we need to explore all such options, as for the decompositions of rule (T-PAR), for which the exponential complexity can be reached. Still, we can show the efficiency is improved by observing that:

- the two options for β in the mentioned rules are present only once for each name in between applications of the rule for parallel composition, since we can rely on rule (A-OUT-1) once a choice has already been taken, and
- the number of possible decompositions of the set $\{a\}\cup\gamma$ is smaller than the number of possible decompositions of ρ, since the former is necessarily contained in the latter.

Furthermore, moving away from the worst case analysis, since we are only interested in well-typed processes (see Definition 3.6.4), we have that at top level all assumptions in Δ are of the form $a : a(T)$ or $a : \kappa(T)$ and also that rules (A-NEW) and (A-NEW-REP) only introduce such assumptions. We also have that typing prefixes with subject names with types of the form $a : a(T)$ or $a : \kappa(T)$ do not involve such exploration (since $\beta = \gamma = a$ in one case and in the other case κ is not a valid option). Hence, only when typing a prefix with a variable x as a subject may originate the exploration, and only in very specific cases. In particular, only when construct (x) and/or $a(x)$ are present in addition to authorizations for all possible instantiations for x (given by the type), of which some are scoping over parallel composition(s). Consider also that even in the case when (x) and/or $a(x)$ and authorizations for all instantiations for x are present, but none of these are scoping over a parallel composition, the operator *move* is idempotent so no exploration is necessary.

Even more important than the argued improvement of the efficiency is that the rules of Table 3.8 and Table 3.9 pave the way for a polynomial type-checking procedure. The idea here is to change

the rules to consider only one option, regardless if both are available. We believe this would have a minimal affect on the expressiveness. However, this change would affect our result that the two type systems directly correspond, as shown in the next section. Informally, by changing the rules the complexity should be polynomial, since: the rules of Table 3.8 and Table 3.9 are syntax-directed (except for the alerted condition $\beta = a \vee \beta = \gamma$), where all elements in the premises are operationally obtained considering the elements in the conclusion and where rules are mutually exclusive. Any two typing rules for the same prefix are also mutually exclusive, e.g., $a \in \rho_2 \vee \gamma \subseteq \rho_2$ in rule (A-IN-1) and $a \notin \rho_2 \wedge \gamma \not\subseteq \rho_2$ in rule (A-IN-2). And finally, consider that all operations, such as environment access and update, (multi)set union, intersection and inclusion may be implemented with polynomial complexity.

3.6.6.1 Correspondence result

In this section, we show that the type system given in Section 3.6.3 is equivalent to the type system given in Section 3.6.6. In order to compare the two type systems, we first need to compare the two syntaxes considered, one only with symbols and the other with full type annotations in the name restriction constructs. To this end, we define function $erase(P)$ that removes the extra annotations.

Definition 3.6.16 (Erasing type annotations)**.** *Function erase is defined as a homomorphism except for*

1. $erase((\nu a : \mathbf{r}(T))P) = (\nu a : \mathbf{r})erase(P)$
2. $erase((\nu a : \kappa(T))P) = (\nu a : \kappa)erase(P)$

In order to show the correspondence of the two type systems, we first establish the correlation between the input and the output information when typing process with the refined type system. Namely, the output multiset of names is always contained in the multiset of

names that are considered as to be passed, while the output set of symbols always contains the input set.

Lemma 3.6.17 (Monotonicity)**.** *If* $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$ *then*

1. $\mathsf{sym}(P) = \xi' \setminus \xi$ *and* $\xi \subseteq \xi'$,
2. $\rho' \subseteq \rho_1$.

Proof. The proof is by induction on the depth of the type checking derivation.

1. We detail only when the last applied rules is (A-NEW) or (A-REP-IN).
   - *Case* (A-NEW): Let $\Delta'; \rho_1 : \rho_2; \xi \vdash (\nu a : \mathbf{r}(T))P; \rho'; \xi'$ be derived from $\Delta, a : \{a\}(T); \rho_1 : \rho_2; \xi \cup \mathbf{r} \vdash P; \rho'; \xi'$, where $\Delta' = \Delta\{\mathbf{r}/a\}$, $\mathbf{r} \notin \xi$ and $a \notin \rho_1, \rho_2, names(T)$. By induction hypothesis we have $\xi' \setminus (\xi \cup \mathbf{r}) = \mathsf{sym}(P)$ and $\xi \cup \mathbf{r} \subseteq \xi'$. Then, $\xi' \setminus \xi = \mathsf{sym}(P) \cup \{\mathbf{r}\} = \mathsf{sym}((\nu a : \mathbf{r}(T))P)$ and $\xi \subseteq \xi'$.
   - *Case* (A-REP-IN): Let $\Delta; \rho_1 : \rho_2; \xi \vdash !(a)a?x.P; \rho_1; \xi$ be derived from $\Delta, x : T; \emptyset : \{a\}; \emptyset \vdash P; \emptyset; \emptyset$, $\Delta(a) = \gamma(T)$ and $x \notin \rho_1, \rho_2, names(\Delta)$. By induction hypothesis $\mathsf{sym}(P) = \emptyset \setminus \emptyset = \emptyset$ and since $\mathsf{sym}(P) = \mathsf{sym}(!(a)a?x.P)$, we can conclude $\xi \setminus \xi = \emptyset = \mathsf{sym}(!(a)a?x.P)$.
2. We detail only when the last applied rule is (A-OUT-1) or (A-PAR).
   - *Case* (A-OUT-2): Let $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$ be derived from $\Delta; \rho_1' : \rho_2'; \xi \vdash P; \rho'; \xi'$, where $\rho_1' : \rho_2' = move(\rho_1 : \rho_2; \beta)$, $\Delta(a) = \gamma(\gamma'(T))$, $\Delta(b) = \gamma''(T)$, $a \notin \rho_2 \wedge \gamma \not\subseteq \rho_2$, $\beta = a \vee \beta = \gamma$, and $\gamma'' \subseteq \gamma'$. By induction hypothesis we have $\rho' \subseteq \rho_1'$. By the definition of the operator *move* we may conclude $\rho_1' \subset \rho_1$. Hence, $\rho' \subseteq \rho_1$.
   - Case (A-PAR): Directly, since $\rho_1 \cap \rho_4 \subseteq \rho_1$.

□

We may now show one direction of our correspondence result: if a process is typed using the refined system then the corresponding process obtained by the application of the *erase* function can also be typed with the original type system.

Lemma 3.6.18 (Typing correspondence: soundness)**.** *If* $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$*, then* $\Delta \vdash_\rho erase(P)$*, where* $\rho = (\rho_1 \uplus \rho_2) - \rho'$*.*

Proof. The proof is by induction on the depth of the type checking derivation. We detail only the base case obtained by rule (A-STOP), and, for the induction step, only when the last applied rule is (A-PAR) or (A-REP-IN).

- *Case* (A-STOP): Let $\Delta; \rho_1 : \rho_2; \xi \vdash 0; \rho_1; \xi$. By (T-STOP) we can directly derive $\Delta \vdash_{\rho_2} 0$.

- *Case* (A-PAR): Let $\Delta; \rho_1 : \rho_2; \xi \vdash P_1 \mid P_2; \rho_1 \cap \rho_4; \xi''$ be derived from

$$\Delta; \rho_1 \uplus \rho_2 : \emptyset; \xi \vdash P_1; \rho_3; \xi' \text{ and } \Delta; \rho_3 : \emptyset; \xi' \vdash P_2; \rho_4; \xi''$$

  By Lemma 3.6.17 we have $\mathsf{sym}(P_1) = \xi' \setminus \xi$ and $\xi \subseteq \xi'$ and $\rho_3 \subseteq \rho_1 \uplus \rho_2$, but also $\mathsf{sym}(P_2) = \xi'' \setminus \xi'$ and $\xi' \subseteq \xi''$ and $\rho_4 \subseteq \rho_3$. Hence, we may conclude that $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$. By induction hypothesis we have

$$\Delta \vdash_{(\rho_1 \uplus \rho_2) - \rho_3} erase(P_1) \quad \text{and} \quad \Delta \vdash_{\rho_3 - \rho_4} erase(P_2).$$

  Since $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$, by (T-PAR) we obtain $\Delta \vdash_{(\rho_1 \uplus \rho_2) - \rho_4} erase(P_1 \mid P_2)$. By Lemma 3.6.6, we conclude

$$\Delta \vdash_{(\rho_1 \uplus \rho_2) - (\rho_1 \cap \rho_4)} erase(P_1 \mid P_2).$$

- *Case* (A-REP-IN): Let $\Delta; \rho_1 : \rho_2; \xi \vdash !(a)a?x.P; \rho_1; \xi$ be derived from $\Delta, x : T; \emptyset : \{a\}; \emptyset \vdash P; \emptyset; \emptyset$. By Lemma 3.6.17 we have $\mathsf{sym}(P) = \emptyset$. By induction hypothesis we obtain $\Delta, x : T \vdash_{\{a\}} erase(P)$. Since $\mathsf{sym}(P) = \emptyset$, by (T-REP-IN) follows $\Delta \vdash_{\rho_2} erase(!(a)a?x.P)$.

□

To obtain the completeness result we first show a form of Weakening result for the refined type system (cf. Lemma 3.6.6).

Lemma 3.6.19 (Weakening)**.** *If* $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$ *then* $\Delta; \rho_1 \uplus \rho; \rho_2; \xi \vdash P; \rho' \uplus \rho; \xi'$.

Proof. The proof is by induction on the depth of the type checking derivation. We detail only the base case, given by (A-STOP), and the case of (A-PAR).

- *Case* (A-STOP): Let $\Delta; \rho_1 : \rho_2; \xi \vdash 0; \rho_1; \xi$. By the same rule we may also observe $\Delta; \rho_1 \uplus \rho : \rho_2; \xi \vdash 0; \rho_1 \uplus \rho; \xi$.
- *Case* (A-PAR): Let $\Delta; \rho_1 : \rho_2; \xi \vdash P_1 \mid P_2; \rho_1 \cap \rho_4; \xi''$ be derived from

  $$\Delta; \rho_1 \uplus \rho_2; \emptyset; \xi \vdash P_1; \rho_3; \xi' \text{ and } \Delta; \rho_3 : \emptyset; \xi' \vdash P_2; \rho_4; \xi''$$

  By induction hypothesis we have

  $$\Delta; \rho_1 \uplus \rho_2 \uplus \rho : \emptyset; \xi \vdash P_1; \rho_3 \uplus \rho; \xi' \quad \text{and} \quad \Delta; \rho_3 \uplus \rho; \emptyset; \xi' \vdash P_2; \rho_4 \uplus \rho; \xi''$$

  Then, by (A-PAR) we derive

  $$\Delta; \rho_1 \uplus \rho : \rho_2; \xi \vdash P_1 \mid P_2; (\rho_1 \uplus \rho) \cap (\rho_4 \uplus \rho); \xi''$$

  which concludes the proof since $(\rho_1 \uplus \rho) \cap (\rho_4 \uplus \rho) = (\rho_1 \cap \rho_4) \uplus \rho$.

□

The completeness result shows that if a process is typed with the original system, then it can also be typed with the refined one (again, up to the *erase* function).

Lemma 3.6.20 (Typing correspondence: completeness)**.** *If* $\Delta \vdash_\rho \mathit{erase}(P)$ *then for any* ρ_1, ρ_2 *multisets of names, and any* ξ, ξ' *sets of symbols from* $\mathcal{S}$*, such that* $\rho_1 \uplus \rho_2 = \rho$ *and* $\xi' \setminus \xi = \mathsf{sym}(P)$*, we have that* $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$*, for some* ρ'.

Proof. The proof is by induction on the derivation of the original type system. We detail the base case, induced by (T-STOP), and for the induction step we detail only the cases of (T-PAR) and (T-IN).

- *Case* (T-STOP): Let $\Delta \vdash_\rho 0$. By (A-STOP) we can derive $\Delta; \rho_1 : \rho_2; \xi \vdash 0; \rho_1; \xi$, for any ρ_1, ρ_2 and ξ, such that $\rho_1 \uplus \rho_2 = \rho$.

- *Case* (T-PAR): Let $\Delta \vdash_\rho \mathit{erase}(P_1 \mid P_2)$ be derived from $\Delta \vdash_{\rho'} \mathit{erase}(P_1)$ and $\Delta \vdash_{\rho''} \mathit{erase}(P_2)$ where $\rho' \uplus \rho'' = \rho$ and $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$. By induction hypothesis we can derive

$$\Delta; \rho' : \emptyset; \xi_1 \vdash P_1; \rho'_1; \xi_2 \text{ and } \Delta; \rho'' : \emptyset; \xi_2 \vdash P_2; \rho'_2; \xi_3$$

  for any ξ_1, ξ_2 and ξ_3, such that $\xi_2 \backslash \xi_1 = \mathsf{sym}(P_1)$, $\xi_3 \backslash \xi_2 = \mathsf{sym}(P_2)$, and where $\xi_1 \cap \mathsf{sym}(P_2) = \emptyset$. Using ξ_2 to type process P_2 is possible since $\xi_2 \cap \mathsf{sym}(P_2) = \emptyset$ holds by $\xi_1 \cap \mathsf{sym}(P_2) = \emptyset$ and the assumption $\mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset$. By Lemma 3.6.19 we derive

$$\Delta; \rho : \emptyset; \xi_1 \vdash P_1; \rho'_1 \uplus \rho''; \xi_2 \text{ and } \Delta; \rho'' \uplus \rho'_1 : \emptyset; \xi_2 \vdash P_2; \rho'_2 \uplus \rho'_1; \xi_3$$

  and by (A-PAR) we conclude

$$\Delta; \rho_1 : \rho_2; \xi_1 \vdash P_1 \mid P_2; \rho_1 \cap (\rho'_2 \uplus \rho'_1); \xi_3$$

- *Case* (T-OUT): Let $\Delta \vdash_\rho \mathit{erase}(a!b.P)$ be derived from $\Delta \vdash_\rho P$, where $\Delta(a) = \gamma(\gamma'(T))$, $\Delta(b) = \gamma''(T)$ and $\gamma'' \subseteq \gamma'$, and where $a \notin \rho$ implies $\gamma \in \rho$. By induction hypothesis

$$\Delta; \rho'_1 : \rho'_2; \xi \vdash P; \rho'; \xi'$$

  for any ρ'_1, ρ'_2 and ξ, ξ', such that $\rho'_1 \uplus \rho'_2 = \rho$ and $\xi' \setminus \xi = \mathsf{sym}(P)$. We now have to show that $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$ for any ρ_1 and ρ_2 such that $\rho_1 \uplus \rho_2 = \rho$. We distinguish four cases.

  (i) If $a \in \rho$ and $a \in \rho_2$ we chose $\rho'_2 = \rho_2$ and by (A-OUT-1) we derive $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$.

(ii) If $a \in \rho$ and $a \notin \rho_2$ we chose $\rho_2' = \rho_2 \uplus \{a\}$ and by (A-OUT-2) we derive $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$.

(iii) If $\gamma \in \rho$ and $\gamma \in \rho_2$ we chose $\rho_2' = \rho_2$ and by (A-OUT-1) we derive $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$.

(iv) If $\gamma \in \rho$ and $\gamma \notin \rho_2$ we chose $\rho_2' = \rho_2 \uplus (\gamma \setminus \rho_2)$ and by (A-OUT-2) we derive $\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'$.

□

We may now state the main result of this section.

Theorem 3.6.21 (Typing correspondence)**.** $\Delta \vdash_\rho \mathit{erase}(P)$ *if and only if* $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$*, for any* ρ_1, ρ_2 *and* ξ *such that* $\rho_1 \uplus \rho_2 = \rho$ *and* $\xi' \setminus \xi = \mathsf{sym}(P)$.

Proof.

($\Leftarrow$) Let $\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi'$. For $\rho = \rho_1 \uplus \rho_2$, by Lemma 3.6.18 we obtain $\Delta \vdash_{\rho - \rho'} \mathit{erase}(P)$. Then, by Lemma 3.6.6 we can conclude $\Delta \vdash_\rho \mathit{erase}(P)$.

($\Rightarrow$) Directly by Lemma 3.6.20.

□

Theorem 3.6.21 shows that the type-checking procedures induced by the original and the refined type rules perfectly correspond, while the latter rules offer for a more efficient implementation. For future work, we plan to make a precise complexity analysis so as to characterize the efficiency improvement. Also, we plan to make a precise characterization of the loss of expressiveness that results from the pragmatical (polynomial time) option.

3.7 Extended example

This section presents an example motivated by the *Bring Your Own License* (BYOL) notion, that should provide further insights in our model and our type discipline, and also to provide a link to a relevant domain in practice. The BYOL is a licensing model that is closely related to the cloud-based based computing. The cloud computing provides on-demand availability of resources, such as deploying and running applications, data storage, etc. Typically, these resources can be accessed by many users (over the internet), but are not directly maintained by users. The core of the BYOL model is that it provides flexibility to a user that is willing to deploy a software application in the cloud to also deploy his own license. Here, we can recognize the pattern of delegation, as the user can lose the ability to run the application elsewhere by deploying his license in the cloud.

To provide a proper mindset, let us consider an example in which a *Company* is willing to deploy a *query* service (that may require some exhaustive computations) in the cloud and to store the resulting data on a cloud database. For instance, *Company* may consider offers from two cloud service providers, Amazon Web Service (*AWS*) and IBM Cloud (*IBM*), and is licensed to use Microsoft Azure SQL Database (*SQL*). We may then model this scenario with

$$Company \mid AWS \mid IBM \mid SQL$$

where we consider processes can run concurrently. The communication protocol of the two providers *AWS* and *IBM* may be represented as

$$AWS = !(aws)aws?service.aws(service).service!data.0$$

$$IBM = !(ibm)ibm?service.ibm(service).service!data.0$$

specifying that the providers can repeatably receive a *service* name, and then to receive the respective authorization, which would allow for the provider to use *service* for sending some data. We abstract here

from the computation task itself, given that we focus exclusively on the communication pattern, so running the *service* amounts to sending the *data*.

We now model *Company* by specifying two managers and two workers that run concurrently. A manager should decide which service provider is to be used and a worker is responsible for interaction with the providers. Therefore, we define

$$Company = (query)^2(ibm)^2(aws)^2(Manager \mid Manager \mid Worker \mid Worker)$$

where the company owns two of each of authorizations to interact with services *IBM* and *AWS* and with the database center *SQL* (through *query*). Consider that a manager's decision of which cloud service should be used depends on the quality of service (*qos*), and that the manager notifies a worker about the decision via channel *choice*. For the purpose of modeling this process, we make a simple extension of our language with a conditional statement "if then else". Therefore,

$$Manager = (choice) \text{ if } (qos) \text{ then } choice!aws.0 \text{ else } choice!ibm.0$$

Once a manager has made the decision and notified the worker by sending the name of the chosen service provider, one of the workers receives the name and then sends *query* and the respective authorization to the provider

$$Worker = (choice)choice?csp.csp!query.csp\langle query\rangle.0$$

We may notice that the authorization for *query* is at first implicitly available for the whole *Company* and that a *Worker* will grab and explicitly delegate the authorization in order to allow the provider to use *query*. Also, two of each of authorizations for *aws* and *ibm* are originally available to all processes in the domain of the company, hence the two *Manager*s can choose the two providers can be used zero, one or two times. In any of the four cases, the *query* is used exactly twice (by each of the two *Worker*s).

We model the cloud database so as to be able repeatably to receive a request along *query*, hence

$$SQL = !(query)query?x.0$$

For the sake of simplicity, we abstract away from possible later interactions, since these may also require isolation (e.g., via dedicated private channels).

We now observe possible interactions in the system. Consider that one of the *Manager*s makes a decision to use *ibm* service provider and sends the name to one of the *Worker*s. If we assume a usual extension of the LTS with rules to deal with the "if then else" statement by considering the evolution, via a τ transition, to the *then* branch when the condition is true, and otherwise to the *else* branch, together with (L-OUT) and (L-SCOPE-EXT) we may derive

$$Manager \xrightarrow{\tau} \xrightarrow{choice!ibm} (choice)0$$

Then, one of the *Worker*s performs the (authorized)reception by rules (L-IN) and (L-SCOPE-EXT)

$$Worker \xrightarrow{choice?ibm} Worker'$$

where $Worker' = (choice)ibm!query.ibm\langle query\rangle.0$. The two dual actions along *choice* can then synchronize when the two processes are composed in parallel. Thus, by rule (L-COMM) we have

$$Manager \mid Worker \xrightarrow{\tau}\xrightarrow{\tau} (choice)0 \mid Worker'$$

Now, the decision has been made, and $Worker'$ can send the request to the chosen provider by rule (L-OUT)

$$Worker' \xrightarrow{(ibm)ibm!query} Worker''$$

where $Worker'' = (choice)(ibm)ibm\langle query\rangle.0$. We may observe that the action is not authorized for name *ibm*, that is confined to $Worker''$

as the result of the transition. However, it becomes authorized at the level of the company where the authorization is floating

$$(query)^2(ibm)^2(aws)^2(Manager \mid (choice)0 \mid Worker' \mid Worker) \xrightarrow{ibm!query} (query)^2(ibm)(aws)^2(Manager \mid (choice)0 \mid Worker'' \mid Worker)$$

by rules (L-SCOPE-EXT), (L-PAR) and (L-SCOPE). Following the same reasoning, the company, via process $Worker''$, can delegate authorization for *query* on channel *ibm*, as by rule (L-OUT-A)

$$Worker'' \xrightarrow{(query)ibm\langle query\rangle} (choice)(ibm).0$$

and by rules (L-PAR), (L-SCOPE-EXT) and (L-SCOPE), we may derive

$$(query)^2(ibm)(aws)^2(Manager \mid (choice)0 \mid Worker'' \mid Worker) \xrightarrow{ibm\langle query\rangle} (query)(ibm)(aws)^2(Manager \mid (choice)0 \mid (choice)(ibm)0 \mid Worker)$$

causing the *Company* to lose one authorization for *query*.

The *IBM* process first receives name *query*, and then receives the authorization, by the rules (L-IN-REP), (L-IN-A), (L-SCOPE-EXT) and (L-PAR)

$$IBM \xrightarrow{ibm?query} \xrightarrow{ibm(query)} IBM',$$

where $IBM' = IBM \mid (query)(ibm)query!data.0$. Finally, the service provider IBM' is connected with *SQL* through channel *query* and also authorized to use the channel. Hence, IBM' can send the *data* to *SQL*.

We may now attest that the above scenario contains no errors by applying our typing analysis. We assume an additional typing rule that deals with the conditional statement in a standard way: both branches and the whole conditional should have the same type. Let us consider the type assignments collected in a typing environment

$$\begin{array}{rcl} \Delta & = & aws : \{aws\}(T), \\ & & ibm : \{ibm\}(T), \\ & & query : T, \\ & & choice : \{choice\}(\{aws, ibm\}(T)) \end{array}$$

where $T = \{query\}(\{data\}(\emptyset))$. We can then check

$$\Delta \vdash_{\emptyset} Company \mid AWS \mid IBM \mid SQL$$

Thus, the system is well-typed and owns enough authorizations, so it never gets stuck.

We may now derive

$$\Delta \vdash_{\rho} Worker$$

where $\rho = \{query, ibm, aws\}$. Let us recall

$$Worker = (choice)choice?csp.csp!query.csp\langle query\rangle.0.$$

Since rule (T-STOP) claims that the inactive process can be typed by any assumptions, we may observe

$$\Delta, csp : \{aws, ibm\}(T) \vdash_{\rho_1} 0$$

where $\rho_1 = \{choice, ibm, aws\}$. Then, by rule (T-DELEG), we may derive

$$\Delta, csp : \{aws, ibm\}(T) \vdash_{\rho_1, \{query\}} csp\langle query\rangle.0$$

since all replacements for csp given in its type ($\{aws, ibm\}$) are contained in ρ_1, to which we add the name of the delegated authorization in the conclusion. The added name ensures that the process must be placed in a context that provides all the necessary authorizations. If we now apply rule (T-OUT), we observe

$$\Delta, csp : \{aws, ibm\}(T) \vdash_{\rho_1, \{query\}} csp!query.csp\langle query\rangle.0$$

where we again have the same check for csp (now considering $\rho_1, \{query\}$), and we also check that all possible replacements for $query$, which is only $query$ itself, are contained in the carried type of csp, which holds since $T = \{query\}(\{data\}(\emptyset))$. Then, by (T-IN)

$$\Delta \vdash_{\rho_1, \{query\}} choice?csp.csp!query.csp\langle query\rangle.0$$

since *choice* is in $\rho_1, \{query\}$ and the type of *csp* is equal to the carried type of *choice*. Finally, by (T-AUTH) we conclude

$$\Delta \vdash_{\rho} Worker$$

where ρ is obtained from $\rho_1, \{query\}$ by removing *choice*. Hence, each *Worker* is safe if placed in a context that provides authorizations for names *query*, *ibm*, and *aws*. We remark that authorizations for *ibm* and *aws* will not be both actually needed since only one of them can be received in *choice*. However, to ensure any possible evolution of the system is safe, both authorizations (for *ibm* and *aws*) must be provided.

3.8 Towards applications

This section provides an intuition of how our model can lead to developments in a practical setting. Specifically, we consider an extension of a programming language that embeds the principles presented in this work. We remark that authorizations and many other security concerns are often handled separately with respect to the application layer. However, it may be the case that the application domain and the security concerns are closely entangled. Thus, establishing the correspondence between application requirements and security layer guarantees, in general, may not be a trivial task. For instance, a resource management can be tightly related to the application due to service contracts that are subject to optimization by the application business logic. Providing programming language direct support for the resource management in such cases might be an important solution. Following these lines, we believe introducing programming language support for our notion of floating authorizations can produce some benefits in practice.

To illustrate the idea let us consider Figure 3.1 that presents the code of a program written in Go. Here, the language of Go is endowed with a construct `auth( )` that represents our authorization scoping

construct. The construct is used in lines 12, 17, 18, 22, 29, 34, and 38, and marked with the comment `// Language extension`). The program represents the *Company* process described in Section 3.7, simplified here by considering there is only one manager thread that sends a channel to two worker threads. For simplicity, we only use anonymous goroutines for which the intended authorization scoping is given directly by the syntax.

First, the program creates three channels `choice`, `aws`, and `ibm` with the proper type assignments (lines 6-8). Channel `choice` can be used to communicate messages of type `chan string`, meaning that `choice` is used to communicate channel endpoints that can be used to communicate strings. Types of `aws` and `ibm` specify that these two channels can be used to communicate strings. After that, in lines 11-15, a goroutine, that represents a manager process, is defined and directly invoked so as to spawn a thread that will run the code given in the body of the routine, in lines 12-14. Specifically, in line 12 an authorization for channel `choice` is created, after which in lines 13 and 14 channel endpoints for `aws` and `ibm`, respectively, are sent via channel `choice`. We may observe that the manager thread is self-sufficient in terms of authorizations as both communications are carried out using channel `choice`, for which the respective authorization is present.

The main thread continues to lines 17 and 18 where the authorizations for `aws` and `ibm` are created. Then, in lines 21-25 and 28-32 the goroutines for two worker threads the defined and invoked. The code for the workers specifies the creation of an authorization for channel `choice` in lines 22 and 29, after which in lines 23 and 30 the reception of a channel on channel `choice` is performed, and finally the emission of a text message on the received channel is conducted in lines 24 and 31. The main thread then carries out the receptions of the text messages in lines 35 and 39, for which the respective authorizations are provided in lines 34 and 38, and finally, their onscreen display in lines 36 and 40.

We may observe that the two workers receive the `aws` and `ibm`

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          choice := make(chan chan string)
7          aws := make(chan string)
8          ibm := make(chan string)
9
10         // MANAGER
11         go func() {
12                 auth(choice) // Language extension
13                 choice <- aws
14                 choice <- ibm
15         }()
16
17         auth(ibm) // Language extension
18         auth(aws) // Language extension
19
20         // WORKER 1
21         go func() {
22                 auth(choice) // Language extension
23                 csp := <-choice
24                 csp <- "worker1"
25         }()
26
27         // WORKER 2
28         go func() {
29                 auth(choice) // Language extension
30                 csp := <-choice
31                 csp <- "worker2"
32         }()
33
34         auth(aws) // Language extension
35         msg1 := <-aws
36         fmt.Println("aws", msg1)
37
38         auth(ibm) // Language extension
39         msg2 := <-ibm
40         fmt.Println("ibm", msg2)
41
```

endpoints on channel `choice` sent by the manger. Thus, the workers output the text messages (lines 24 and 31) on the `aws` and `ibm` channels. Notice that the two authorizations for `aws` and `ibm` are created in the main thread (lines 17-18). Hence, these are floating over both worker threads spawned by the goroutine invocation, and each one of the authorizations can be confined to either one of the threads. Also, we may observe that each worker thread receives a different channel and which one is not prescribed by the code. Therefore, `"worker1"` can be sent on `aws` and `"worker2"` on `ibm`, or `"worker2"` can be sent on `aws` and `"worker1"` on `ibm`. In both cases, the program can be considered as authorization safe since the respective floating authorization can be confined to the thread at the moment it uses the channel.

We may now consider refinements of the resource management in the context of this simple example. If channel endpoints are subject to a rigorous accounting, for instance, according to an established service contract (for instance, with third parties, such as Amazon or IBM, where the number of active channel endpoints may be restricted), it is important to have language mechanisms that give direct support to conform to the service contract. The creation of the authorizations in lines 17-18 gives a precise operational specification of the resources available. Notice that we could also reason on authorization reception replacing the creation in lines 17-18 so as to, for instance, rely on a communication with the service provider to establish the authorizations. Furthermore, the floating authorizations provide the flexibility for the resources to be accessible among concurrent threads while keeping track of the resource control.

We can also build on our typing analysis to devise static verification techniques that can ensure programs like the one shown never get stuck due to lacking authorizations. However, notice that a direct application of our typing discipline would exclude the example given in Figure 3.1, since in the carried type of channel `choice` both `aws` and `ibm` names would have to be mentioned, as both indeed can be communicated in the channel. Following these lines, to ensure that both

workers do not lack authorizations we would require two authorizations for each `aws` and `ibm`. We conceive that our typing analysis can be extended so as to address this configuration and, moreover, that our language principles can already be used as the basis for runtime verification techniques that would allow to transparently run the example program. We also conceive that our language principles can be embedded in a programming language by means of a specialized API, instead of a proper extension, relying on the appropriate library calls for authorization creation and resource usage, up to some ingenuity for authorization scoping. We do retain the necessity of considering specialized language constructs for the sake of the dedicated theoretical investigation presented in this thesis.

(L-OUT) $a!b.P \xrightarrow{(a)a!b} (a)P$

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

(L-OUT-A) $a\langle b\rangle.P \xrightarrow{(a)(b)a\langle b\rangle} (a)P$

(L-IN-A) $a(b).P \xrightarrow{(a)a(b)} (a)(b)P$

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

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

(L-RES) $$\frac{P \xrightarrow{\alpha} Q \quad a \notin \mathsf{n}(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)Q}$$

(L-OPEN) $$\frac{P \xrightarrow{(a)^i a!b} Q \quad a \neq b}{(\nu b)P \xrightarrow{(a)^i(\nu b)a!b} Q}$$

(L-SCOPE-INT) $$\frac{P \xrightarrow{\tau_{\omega(a)}} Q}{(a)P \xrightarrow{\tau_\omega} Q}$$

(L-SCOPE-EXT) $$\frac{P \xrightarrow{(a)\sigma_a} Q}{(a)P \xrightarrow{\sigma_a} Q}$$

(L-SCOPE) $$\frac{P \xrightarrow{\alpha} Q \quad \tau_{\omega(a)} \neq \alpha \neq (a)\sigma_a}{(a)P \xrightarrow{\alpha} (a)Q}$$

(L-COMM) $$\frac{P \xrightarrow{(a)^i a!b} P' \quad Q \xrightarrow{(a)^j a?b} Q' \quad \omega = (a)^{i+j}}{P \mid Q \xrightarrow{\tau_\omega} P' \mid Q'}$$

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

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

Table 3.2: LTS rules.

$$\text{(SC-PAR-INACT)}\quad P \mid 0 \equiv P \qquad \text{(SC-PAR-COMM)}\quad P \mid Q \equiv Q \mid P \qquad \text{(SC-PAR-ASSOC)}\quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$\text{(SC-RES-INACT)}\quad (\nu a)0 \equiv 0 \qquad \text{(SC-RES-SWAP)}\quad (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$$

$$\text{(SC-RES-EXTR)}\quad P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin \mathsf{fn}(P) \qquad \text{(SC-REP)}\quad !(a)a?x.P \equiv !(a)a?x.P \mid (a)a?x.P$$

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

Table 3.3: Structural congruence.

$$\text{(C-END)}\quad \mathit{drift}(\,\cdot\,; \emptyset; \rho') = \cdot \qquad \text{(C-REM)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot]; \rho; \rho' \uplus \{a\}) = \mathcal{C}'[\cdot]}{\mathit{drift}((a)\mathcal{C}[\cdot]; \rho \uplus \{a\}\,; \rho') = \mathcal{C}'[\cdot]}$$

$$\text{(C-SKIP)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot]; \rho; \rho') = \mathcal{C}'[\cdot] \quad a \notin \rho'}{\mathit{drift}((a)\mathcal{C}[\cdot]; \rho; \rho') = (a)\mathcal{C}'[\cdot]} \qquad \text{(C-PAR)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot]; \rho; \rho') = \mathcal{C}'[\cdot]}{\mathit{drift}(\mathcal{C}[\cdot] \mid R; \rho; \rho') = \mathcal{C}'[\cdot] \mid R}$$

Table 3.4: *drift* on contexts with one hole

$$\text{(C2-SPL)}\quad \frac{\mathit{drift}(\mathcal{C}_1[\cdot_1];\rho_1;\rho_1') = \mathcal{C}_1'[\cdot] \quad \mathit{drift}(\mathcal{C}_2[\cdot_2];\rho_2;\rho_2') = \mathcal{C}_2'[\cdot]}{\mathit{drift}(\mathcal{C}_1[\cdot_1] \mid \mathcal{C}_2[\cdot_2];\rho_1;\rho_2;\rho_1';\rho_2') = \mathcal{C}_1'[\cdot_1] \mid \mathcal{C}_2'[\cdot_2]}$$

$$\text{(C2-REM-L)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2;\rho_1' \uplus \{a\};\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}((a)\mathcal{C}[\cdot_1,\cdot_2];\rho_1 \uplus \{a\};\rho_2;\rho_1';\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2]}$$

$$\text{(C2-REM-R)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2;\rho_1';\rho_2' \uplus \{a\}) = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}((a)\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2 \uplus \{a\};\rho_1';\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2]}$$

$$\text{(C2-SKIP)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2;\rho_1';\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2] \quad a \notin \rho_1' \uplus \rho_2'}{\mathit{drift}((a)\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2;\rho_1';\rho_2') = (a)\mathcal{C}'[\cdot_1,\cdot_2]}$$

$$\text{(C2-PAR)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];\rho_1;\rho_2;\rho_1';\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2]}{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2] \mid R;\rho_1;\rho_2;\rho_1';\rho_2') = \mathcal{C}'[\cdot_1,\cdot_2] \mid R}$$

Table 3.5: *drift* on contexts with two holes.

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

$$\text{(R-AUTH)}\quad \frac{\mathit{drift}(\mathcal{C}[\cdot_1,\cdot_2];a,b;a) = \mathcal{C}^-[\cdot_1,\cdot_2]}{\mathcal{C}[a\langle b\rangle.P, a(b).Q] \to \mathcal{C}^-[(a)P,(a)(b)Q)]}$$

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

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

Table 3.6: Reduction rules.

$$\frac{}{\Delta \vdash_\rho 0}\ (\text{T-STOP}) \qquad \frac{\Delta \vdash_{\rho_1} P_1 \quad \Delta \vdash_{\rho_2} P_2 \quad \mathsf{sym}(P_1) \cap \mathsf{sym}(P_2) = \emptyset}{\Delta \vdash_{\rho_1 \uplus \rho_2} P_1 \mid P_2}\ (\text{T-PAR})$$

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

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

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

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

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

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

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

Table 3.7: Typing rules.

$$\text{(A-STOP)}\quad \Delta;\rho_1:\rho_2;\xi \vdash 0;\rho_1;\xi \qquad \text{(A-AUTH)}\quad \frac{\Delta;\rho_1:\rho_2 \uplus \{a\};\xi \vdash P;\rho';\xi'}{\Delta;\rho_1:\rho_2;\xi \vdash (a)P;\rho';\xi'}$$

$$\text{(A-PAR)}\quad \frac{\Delta;\rho_1 \uplus \rho_2:\emptyset;\xi \vdash P_1;\rho_3;\xi' \quad \Delta;\rho_3:\emptyset;\xi' \vdash P_2;\rho_4;\xi''}{\Delta;\rho_1:\rho_2;\xi \vdash P_1 \mid P_2;\rho_1 \cap \rho_4;\xi''}$$

$$\text{(A-NEW)}\quad \frac{\Delta,a:\{a\}(T);\rho_1:\rho_2;\xi \cup \mathbf{r} \vdash P;\rho';\xi' \quad \Delta' = \Delta\{\mathbf{r}/a\} \quad \mathbf{r} \notin \xi \quad a \notin \rho_1,\rho_2,\mathit{names}(T)}{\Delta';\rho_1:\rho_2;\xi \vdash (\nu a:\mathbf{r}(T))P;\rho';\xi'}$$

$$\text{(A-NEW-REP)}\quad \frac{\Delta,a:\kappa(T);\rho_1:\rho_2;\xi \vdash P;\rho';\xi' \quad a \notin \rho_1,\rho_2,\mathit{names}(T,\Delta)}{\Delta;\rho_1:\rho_2;\xi \vdash (\nu a:\kappa(T))P;\rho';\xi'}$$

Table 3.8: Type-checking rules (part 1).

(A-OUT-1)
$$\frac{\Delta; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(\gamma'(T)) \quad \Delta(b) = \gamma''(T) \quad \gamma'' \subseteq \gamma' \quad a \in \rho_2 \vee \gamma \subseteq \rho_2}{\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'}$$

(A-OUT-2)
$$\frac{\Delta; \rho_1' : \rho_2'; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(\gamma'(T)) \quad \Delta(b) = \gamma''(T) \quad \gamma'' \subseteq \gamma' \quad a \notin \rho_2 \wedge \gamma \not\subseteq \rho_2 \quad \beta = a \vee \beta = \gamma \quad \rho_1' : \rho_2' = move(\rho_1 : \rho_2, \beta)}{\Delta; \rho_1 : \rho_2; \xi \vdash a!b.P; \rho'; \xi'}$$

(A-IN-1)
$$\frac{\Delta, x : T; \rho_1 : \rho_2; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad x \notin \rho_1, \rho_2, names(\Delta) \quad a \in \rho_2 \vee \gamma \subseteq \rho_2}{\Delta; \rho_1 : \rho_2; \xi \vdash a?x.P; \rho'; \xi'}$$

(A-IN-2)
$$\frac{\Delta, x : T; \rho_1' : \rho_2'; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad x \notin \rho_1, \rho_2, names(\Delta) \quad a \notin \rho_2 \wedge \gamma \not\subseteq \rho_2 \quad \beta = a \vee \beta = \gamma \quad \rho_1' : \rho_2' = move(\rho_1 : \rho_2, \beta)}{\Delta; \rho_1 : \rho_2; \xi \vdash a?x.P; \rho'; \xi'}$$

(A-REP-IN)
$$\frac{\Delta, x : T; \emptyset : \{a\}; \emptyset \vdash P; \emptyset; \emptyset \quad \Delta(a) = \gamma(T) \quad x \notin \rho_1, \rho_2, names(\Delta)}{\Delta; \rho_1 : \rho_2; \xi \vdash !(a)a?x.P; \rho_1; \xi}$$

(A-DELEG-1)
$$\frac{\Delta; \rho_1' : \rho_2'; \xi \vdash_\rho P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad \rho_1' : (\rho_2' \uplus \{b\}) = move(\rho_1 : \rho_2, b) \quad a \in \rho_2' \vee \gamma \subseteq \rho_2'}{\Delta; \rho_1 : \rho_2; \xi \vdash a\langle b\rangle.P; \rho'; \xi'}$$

(A-DELEG-2)
$$\frac{\Delta; \rho_1' : \rho_2'; \xi \vdash_\rho P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad \rho_1'' : (\rho_2'' \uplus \{b\}) = move(\rho_1 : \rho_2, b) \quad a \notin \rho_2'' \wedge \gamma \not\subseteq \rho_2'' \quad \beta = a \vee \beta = \gamma \quad \rho_1' : \rho_2' = move(\rho_1'' : \rho_2'', \beta)}{\Delta; \rho_1 : \rho_2; \xi \vdash a\langle b\rangle.P; \rho'; \xi'}$$

(A-RECEP-1)
$$\frac{\Delta; \rho_1 : \rho_2 \uplus \{b\}; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad a \in \rho_2 \vee \gamma \subseteq \rho_2}{\Delta; \rho_1 : \rho_2; \xi \vdash_\rho a(b).P; \rho'; \xi'}$$

(A-RECEP-2)
$$\frac{\Delta; \rho_1' : \rho_2' \uplus \{b\}; \xi \vdash P; \rho'; \xi' \quad \Delta(a) = \gamma(T) \quad a \notin \rho_2 \wedge \gamma \not\subseteq \rho_2 \quad \beta = a \vee \beta = \gamma \quad \rho_1' : \rho_2' = move(\rho_1 : \rho_2, \beta)}{\Delta; \rho_1 : \rho_2; \xi \vdash_\rho a(b).P; \rho'; \xi'}$$

Table 3.9: Type-checking rules (part 2).

Chapter 4

Conclusion

In this chapter, we summarize the contributions, present related work, and give some initial ideas for future work.

4.1 Summary of contributions

In this thesis, we have presented two formal models: one modeling confidential information passing via restricting forwarding, and the other modeling controlled usages of resources via floating authorizations.

In Chapter 2, we have presented a model for confidential name passing, called Confidential π-calculus (C_π). Our model, which is a simple fragment of the π-calculus [102], was previously introduced in [89]. The C_π-calculus disables forwarding of received names directly at the syntax level by restricting the π-calculus feature that input variables can appear as objects of the output prefixes. To the best of our knowledge, this is the first process model based on the π-calculus that represents the controlled name passing by constraining and not extending the original syntax. We have defined the non-forwarding property, which claims that a process cannot forward any of its received names, and as a sanity check we have shown that C_π processes satisfy this property. We have also shown that a π process

respects the non-forwarding property when it can be related to a C_π process via strong bisimilarity relation. The same relation is then used to show that in C_π-calculus one can directly represent the creation of closed domains for channels.

We have provided some insight on the usefulness of our model by presenting examples that show the C_π can be used to model restricted information passing, authentication, closed and open-ended groups. The encoding presented in this thesis, which simplifies the one presented in [89], shows that the π-calculus processes can be represented in the C_π-calculus. We have proved the correctness of the encoding in the form of an operational correspondence result (Completeness in Corollary 2.6.6 and Soundness in Corollary 2.6.12).

In Chapter 3, we have presented a model of floating authorizations, which was previously introduced in [80, 92]. We took advantage of already existing work on authorizations [44] to directly adopt the syntax presented there. Hence, our aim was to investigate the accounting principle associated to floating authorizations, by changing an existing model in the minimal necessary way. We defined the semantics for our model in terms of a labeled transition system and also a reduction relation, and we showed these two indeed represent alternatives to each other via a harmony result (Corollary 3.4.13). We motivated our work by showing that the starting process model [44] directly conflicts with our notion of accounting, as it allows to change the number of authorizations in the system directly, e.g., by rewriting rules of the structural congruence. We also defined error processes as undesired configurations that cannot reduce due to lacking authorizations. We have shown there is an alternative way to define errors by using the labeled transition system.

The thesis also provides a preliminary investigation of the behavioral semantics of our authorization model. We have used the strong bisimilarity relation to show some fundamental properties and to validate our design principles, but also to provide insight on the difficulty of obtaining a normal form characterization of processes. To statically

single out processes that are not errors and that never reduce to errors, we devised a typing analysis that addresses contextual authorizations. We have proved a soundness result (Corollary 3.6.14) for our typing discipline. We also presented a refinement of our typing discipline that lead to a more efficient type-checking procedure, and we showed a correspondence result (Theorem 3.6.21) for the two type systems. Finally, we presented an extended example showing a scenario that involves the notion of Bring Your Own License, and we exploit this example to provide an insight on a possible application of our model in programming language design.

Apart from the work reported in this thesis, during his PhD studies the candidate was also involved in research in the field of multiple-valued logic, specifically in the encodings of threshold functions. The results thus far include showing that the well-known results of Chow and Nomura can be generalized to the case of generalized multi-layer S-threshold functions [91, 90], and a characterization of multiple-valued threshold function in the Vilenkin-Chrestenson basis [88, 87].

4.2 Related work

In past, a plethora of approaches have been proposed both for controlling name sharing and name usages. We first address work related to our calculus for confidential name passing, presented in Chapter 2.

Confidentiality and secrecy has been extensively studied in the field of process calculi. We found the process models based on the π-calculus, such as [26, 30, 46, 49, 61, 111], as the most related to our C_π-calculus. Building on the π-calculus, Cardelli et al. [26] introduce an additional language construct that represents the group creation. Groups are then associated to channels as types. The semantics of the model disables the scope extrusion of groups, and their devised typing discipline ensures that grouped names are never communicated on open channels, hence preventing the leakage of protected channels. The work of [26] is used by Kouzapas and Philippou [61] to extend the

model with groups with constructs that permit reasoning about the private data in information systems.

The work of Giunti et. al. [46] considers the π-calculus with an additional operator, called hide. The hide operator resembles the name restriction, as it binds the name specified inside, but is, in a sense, more constraining than name restriction since it blocks extrusion of the name. Hence, the name specified inside the hide operator has closed scope, which again prevents that the name leaks outside its originally defined scope. Vivas and Yoshida [111] have introduced an operator called filter. Filter is statically associated to a process and allow interaction of the process with its environment only on names that are contained in the (polarized) filter, while blocking any other actions of the process. We also mention [30, 49] where the types associate the security levels to channels. In the latter work, the security level of a channel can be downgraded via special declassified input and output prefix constructs.

All models mentioned so far share one property: to be able to reason on a specific aspect of secrecy in a proper way they extend the π-calculus with additional language constructs and/or introduce a typing discipline. In contrast, the C_π-calculus does not extend but uses only a fragment of the π-calculus. For this, we believe that many aspects of secrecy can be modeled and studied in a more canonical way when using our model. As a first step to strengthen this claim, we plan to make a precise representation of group creation [26] in the C_π-calculus, following the intuition provided in Section 2.5.2.

Studying fragments of the π-calculus is not a new idea by itself. We find several proposals following these lines. The most famous probably is the asynchronous π-calculus, proposed by Honda and Takoro [55] and by Boudol [18] independently. The asynchronous π puts a constrain on the π-calculus syntax that only an inactive process can be specified as the continuation of an output prefix. Thus, the output does not block any continuation as it is always performed independently of the rest of the process, which allows to modeling

asynchronous communications. Merro and Sangiorgi proposed the Localized π-calculus [67], which restricts the input capability for the received names (and does not consider the matching operator). Hence, similarly to the C_π-calculus, the Localized π-calculus also puts a restriction on the input variable, but now such a variable cannot appear as a subject of an input prefix (and can appear as the object of an output action).

Sangiorgi has also proposed the Private π-calculus [99], that restricts the π-calculus syntax so that objects of output prefixes are always bound. This induces a simplification in the theory of the Private π, coming from the fact that there is now symmetry between the output and the input prefixes (both bind object of the prefix) and that substitution rising in a synchronization can be considered as α-conversion (renaming of a bound name). A similarity between the C_π and the Private π is that in both forwarding of names is not possible. A significant difference is that in the Private π each name can be sent only once, hence our notion of channel handlers does not seems to be directly representable there. The common goal of all these models is investigating specific notions in a dedicated way, without requiring the introduction of specialized primitives, instead by considering a suitable fragment of the π-calculus.

We now turn to comment on work related to our calculus for floating authorizations, presented in Chapter 3. We find many approaches that address controlling resource usage, such as locks (for mutual exclusion in critical code blocks) and communication protocols (e.g., token ring), just to name a few. A number of type systems have been developed to this end, such as [34, 48, 106], where the types specify capabilities over resources. We believe our model provides more flexibility than those considering type systems, as it directly separates resource and capability. Thus, in our model one can communicate a resource without granting the capabilities, that are provided separately (cf. example with the "unauthorized" brokers given in Section 3.1). We point out that in [34] the types specify the number of messages

that may be exchanged, therefore related to the accounting notion investigated here.

We also find a number of models that introduce capabilities as first class entities. The work on which ours directly relies is the one that introduced authorizations [43, 44]. A detailed comparison is given by a number of examples throughout Chapter 3, basically showing that the original work on authorizations directly conflicts with our notion of accounting. Papers such as [14, 46, 60, 111] address usage of channels and of resources as communication objects. Among these are already mentioned models that consider constructs hide and filter that restrict the behaviors allowed on channels [46, 111]. Both models differ from ours as their constructs are static and therefore not able to capture our floating resource capabilities.

We also mention models that specify usages in a (binding) name scope construct [60], and authorization scopes for resources based on given access policies [14]. In [60], the usage specification that corresponds to a type is directly inserted in the model in a binding scoping construct, which contrasts with our non-binding authorization scoping. Also, [14] provides detailed usage policies that are associated with the authorization scopes for resources. We believe both models [14, 60] are less adequate to represent our notion of floating authorizations, since there access is granted explicitly and controlled via the usage/policy specification. This leads us to think that our notion of confinement cannot be directly represented in these two models in a direct way.

4.3 Future work

Our work on the C_π-calculus is at an early stage and leaves many open questions. For instance, our initial work on testing forwarding of the π processes using the C_π processes, presented in Section 2.4.2, has already opened the question of providing an algorithmic procedure for deriving C_π processes considered in Proposition 2.4.10. Another question is a representation of models with groups and hiding [26, 46]

in our calculus, for which the initial ideas are already presented in Section 2.5.2. Also, one direction for the future work is considering the C_π for modeling and analyzing some already existing protocols, such as OAuth2.0 [77], where we can also find restricted information sharing in the context of an authentication scheme.

Regarding our work on floating authorizations, we also find several directions for possible future work. Our model shows a way to deal with accountable resources that can be accessed in a shared way. This general idea is then made concrete with our design choice to confine authorizations when they are used, this way (permanently) restricting their scope. It would be interesting to consider non-consumptive authorizations, that return to their original scope after (complete) use.

We believe the principles of our work can be applied when considering also "one-shot" authorizations. By one-shot authorizations we think of authorizations that can be used only once, i.e., only for a single action. Considering such authorizations the induced model would be able to make a precise accounting on how many times channels can be used. We remark that such a model would rely on the same technical machinery as the work presented here. For instance, we would use the same reduction rules (relying on the *drift* operator), with a simple twist that the deleted authorizations are not reintroduced as they are in our model (for the confinement purposes). Along these lines, we can consider building our calculus for floating authorizations by considering the C_π-calculus instead of the π-calculus as the underlying model, since the obtained model would combine control over usages with the control over sharing.

By endowing authorizations with type information we would get another layer in the control of usages of channels. For instance, this can be achieved by considering input/output types, where the authorization can be used only for input or output, or session types, where the authorization is to be used according to a specified protocol.

Our type system, presented in Section 3.6, can also be subject to possible improvements and extensions. For instance, 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. Our model can be extended also by some form of usage specifications like the ones mentioned above [14, 60], and by endowing authorizations as indicated in the previous paragraph. 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 [41] to carry out our typing analysis, given that our types can be seen to some extent as refinements on the domain of names. Our typing analysis can be improved also to address names created in the body of replicated input in a uniform way, by considering the approach presented in [40] that provides the unique identification of private names even in the presence of replicated processes.

Even though we have presented a theoretical investigation in this document, we believe our principles developed for floating authorizations can be conveyed to more practical settings. One such setting is the licensing domain (as already mentioned in the Introduction), where we already have identified patents [6, 10, 38] for the purpose of certifying license usage. Another is the setting of recently introduced permissioned blockchain systems such as Hyperledger Fabric [4]. We believe it would be interesting to provide high-level descriptions and prove properties of the Membership Service Provider (MSP), the part of the Fabric system which is responsible for issuing node credentials (used for authorization and authentication). We intend to pursue this idea, starting by aiming at the formal verification of smart contracts (chaincode) in Fabric, exploiting recently introduced developments for smartcontracts [7, 12] in the context of contract-oriented programming [11]. In particular, we have already identified notions of *active contract* [12], which encompasses a running *balance*, and of *authorizations* to perform operations, hence where we expect to also find the

dimensions of domain, accounting and delegation. And the last setting we would like to mention is again protocol OAuth2.0, where we believe some of our principles can be used to reason on Access Token manipulation, since there we also find the notions of domain and delegation.

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115. ACM, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[3] L. Acciai and M. Boreale. Spatial and behavioral types in the pi-calculus. *Information and Computation*, 208(10):1118–1153, 2010.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In R. Oliveira, P. Felber, and Y. C. Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.

[5] W. J. Armstrong, N. Nayar, and K. P. Stamschror. Management of a concurrent use license in a logically-partitioned computer, Oct. 25 2005. US Patent 6,959,291.

[6] W. J. Armstrong, N. Nayar, and K. P. Stamschror. Management of a concurrent use license in a logically-partitioned computer, Oct. 25 2005. US Patent 6,959,291.

[7] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. Sok: Unraveling bitcoin smart contracts. In L. Bauer and R. Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 217–242. Springer, 2018.

[8] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[9] J. C. M. Baeten and D. Sangiorgi. Concurrency theory: A historical perspective on coinduction and process calculi. In J. H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 399–442. Elsevier, 2014.

[10] P. Baratti and P. Squartini. License management system, June 3 2003. US Patent 6,574,612.

[11] M. Bartoletti and R. Zunino. A calculus of contracting processes. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 332–341. IEEE Computer Society, 2010.

[12] M. Bartoletti and R. Zunino. Bitml: A calculus for bitcoin smart contracts. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 83–100. ACM, 2018.

[13] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[14] C. Bodei, V. D. Dinh, and G. L. Ferrari. Checking global usage of resources handled with local policies. *Science of Computer Programming*, 133:20–50, 2017.

[15] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.

[16] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Informatica*, 35(5):353–400, 1998.

[17] M. Boreale and D. Sangiorgi. Some congruence properties for pi-calculus bisimilarities. *Theoretical Computer Science*, 198(1-2):159–176, 1998.

[18] G. Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[19] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[20] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[21] M. Bugliesi, S. Calzavara, and R. Focardi. Formal methods for web security. *Journal of Logical and Algebraic Methods in Programming*, 87:110–126, 2017.

[22] L. Caires, F. Pfenning, and B. Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[23] L. Caires and H. T. Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.

[24] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In F. van Breugel and M. Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2008.

[25] M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[26] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[27] A. Church. A formulation of the simple theory of types. *Jurnal of Symbolic Logic*, 5(2):56–68, 1940.

[28] L. Chwistek. *The theory of constructive types*. University Press, 1925.

[29] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[30] S. Crafa and S. Rossi. Controlling information release in the pi-calculus. *Information and Computation*, 205(8):1235–1273, 2007.

[31] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[32] S. Dal-Zilio and A. D. Gordon. Region analysis and a pi-calculus with groups. *Journal of Functional Programming*, 12(3):229–292, 2002.

[33] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. *Information and Computation*, 256:253–286, 2017.

[34] A. Das, J. Hoffmann, and F. Pfenning. Work analysis with resource-aware session types. In A. Dawar and E. Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 305–314. ACM, 2018.

[35] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[36] R. De Nicola. Behavioral equivalences. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 120–127. Springer, 2011.

[37] J. Dedeić, J. Pantović, and J. A. Pérez. On compensation primitives as adaptable processes. In S. Crafa and D. Gebler, editors, *Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015.*, volume 190 of *EPTCS*, pages 16–30, 2015.

[38] J. M. Ferris and G. E. Riveros. Offering additional license terms during conversion of standard software licenses for use in cloud computing environments, June 9 2015. US Patent 9,053,472.

[39] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. Jocaml: A language for concurrent distributed and mobile programming.

In J. Jeuring and S. L. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002.

[40] A. Francalanza, M. Giunti, and A. Ravara. Pointing to private names. *EasyChair Preprint no. 439*, EasyChair, 2018.

[41] T. S. Freeman and F. Pfenning. Refinement types for ML. In D. S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991.

[42] S. J. Gay. A framework for the formalisation of pi calculus type systems in isabelle/hol. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2001.

[43] S. Ghilezan, S. Jakšić, J. Pantović, J. A. Pérez, and H. T. Vieira. A typed model for dynamic authorizations. In S. Gay and J. Alglave, editors, *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015.*, volume 203 of *EPTCS*, pages 73–84, 2015.

[44] S. Ghilezan, S. Jakšić, J. Pantović, J. A. Pérez, and H. T. Vieira. Dynamic role authorization in multiparty conversations. *Formal Aspects of Computing*, 28(4):643–667, 2016.

[45] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

[46] M. Giunti, C. Palamidessi, and F. D. Valencia. Hide and new in the pi-calculus. In B. Luttik and M. A. Reniers, editors, *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012.*, volume 89 of *EPTCS*, pages 65–79, 2012.

[47] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.

[48] D. Gorla and R. Pugliese. Dynamic management of capabilities in a network aware coordination language. *Journal of Logic and Algebraic Programming*, 78(8):665–689, 2009.

[49] M. Hennessy. The security pi-calculus and non-interference. *Journal of Logic and Algebraic Programming*, 63(1):3–34, 2005.

[50] M. Hennessy. *A distributed Pi-calculus.* Cambridge University Press, 2007.

[51] D. Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 1997.

[52] D. Hirschkoff and D. Pous. A distribution law for CCS and a new congruence result for the π-calculus. *Logical Methods in Computer Science*, 4(2), 2008.

[53] D. Hirschkoff and D. Pous. On bisimilarity and substitution in presence of replication. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 454–465. Springer, 2010.

[54] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[55] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[56] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems, 7th European Symposium on Programming, ESOP 1998, Proceedings*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[57] W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[58] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016.

[59] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[60] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the p-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[61] D. Kouzapas and A. Philippou. Privacy by typing in the π-calculus. *Logical Methods in Computer Science*, 13(4), 2017.

[62] I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, 2011.

[63] Z. Luo. *Computation and reasoning - a type theory for computer science*. Oxford University Press, 1994.

[64] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[65] D. Medić and C. A. Mezzina. Static VS dynamic reversibility in CCS. In S. J. Devitt and I. Lanese, editors, *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, volume 9720 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2016.

[66] L. G. Meredith and M. Radestock. A reflective higher-order calculus. *Electronic Notes in Theoretical Computer Science*, 141(5):49–67, 2005.

[67] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[68] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[69] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[70] R. Milner. The polyadic pi-calculus (abstract). In R. Cleaveland, editor, *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, volume 630 of *Lecture Notes in Computer Science*, page 1. Springer, 1992.

[71] R. Milner. Higher-order action calculi. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*, volume 832 of *Lecture Notes in Computer Science*, pages 238–260. Springer, 1993.

[72] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[73] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[74] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[75] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.

[76] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[77] OAuth Working Group. RFC 6749: OAuth 2.0 Framework. https://tools.ietf.org/html/rfc6749.

[78] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[79] C. Palamidessi, V. A. Saraswat, F. D. Valencia, and B. Victor. On the expressiveness of linearity vs persistence in the

asychronous pi-calculus. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 59–68. IEEE Computer Society, 2006.

[80] J. Pantović, I. Prokić, and H. T. Vieira. A calculus for modeling floating authorizations. In C. Baier and L. Caires, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10854 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2018.

[81] J. Parrow. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, 2008.

[82] R. Perera and J. Cheney. Proof-relevant π-calculus: a constructive account of concurrency and causality. *Mathematical Structures in Computer Science*, 28(9):1541–1577, 2018.

[83] C. A. Petri. *Kommunikation mit automaten*. PhD thesis, Universitat Hamburg, DE, 1962.

[84] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[85] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[86] B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000.

[87] I. Prokić. Characterizations of multiple-valued threshold functions in the Vilenkin-Chrestenson basis. *Journal of Multiple-Valued Logic and Soft Computing*, (accepted).

[88] I. Prokić. Characterization of quaternary threshold functions in the Vilenkin-Chrestenson basis. In *48th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2018, Linz, Austria, May 16-18, 2018*, pages 13–18. IEEE Computer Society, 2018.

[89] I. Prokić. The Cpi-calculus: a model for confidential name passing. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *Electronic Proceedings in Theoretical Computer Science*, pages 115–136. Open Publishing Association, 2019.

[90] I. Prokić and J. Pantović. Nomura parameters for s-threshold functions. In *47th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2017, Novi Sad, Serbia, May 22-24, 2017*, pages 248–253. IEEE Computer Society, 2017.

[91] I. Prokić and J. Pantović. Characterization of generalized s-threshold functions by Nomura parameters. *Journal of Multiple-Valued Logic and Soft Computing*, 33:271 – 290, 2019.

[92] I. Prokić, J. Pantović, and H. T. Vieira. A calculus for modeling floating authorizations. *Journal of Logical and Algebraic Methods in Programming*, 107:136 – 174, 2019.

[93] F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society*, 2(1):338–384, 1926.

[94] W. Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[95] G. Rosen. Security update. https://newsroom.fb.com/news/2018/09/security-update/.

[96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[97] R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[98] D. Sangiorgi. *Expressing mobility in process algebras : first-order and higher-order paradigms*. PhD thesis, University of Edinburgh, UK, 1993.

[99] D. Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1&2):235–274, 1996.

[100] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems*, 31(4):15:1–15:41, 2009.

[101] D. Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[102] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[103] A. Scalas and N. Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

[104] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4-5):547–612, 2007.

[105] D. J. Solove. A taxonomy of privacy. *University of Pennsylvania Law Review*, 154:477, 2005.

[106] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In A. D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 529–549. Springer, 2010.

[107] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. http://homotopytypetheory.org/book, 2013.

[108] R. K. Thiagarajan, A. K. Srivastava, A. K. Pujari, and V. K. Bulusu. BPML: A process modeling language for dynamic business models. In *Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02), Newport Beach, California, USA, June 26-28, 2002*, pages 239–241. IEEE Computer Society, 2002.

[109] M. C. Tschantz and J. M. Wing. Formal methods for privacy. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

[110] V. T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[111] J. Vivas and N. Yoshida. Dynamic channel screening in the higher order pi-calculus. *Electronic Notes in Theoretical Computer Science*, 66(3):170–184, 2002.

[112] P. Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[113] P. H. Welch and F. R. M. Barnes. Communicating mobile processes. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer, 2004.

[114] A. F. Westin. Social and political dimensions of privacy. *Journal of social issues*, 59(2):431–453, 2003.

[115] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910–1913.

[116] Y. Zhao. *Behavioural access control in distributed environments*. PhD thesis, University of York, 2013.